**Project 11 – Hangman / Software Development**

**Name(s)** Ozaner, Armen **Due Date**:

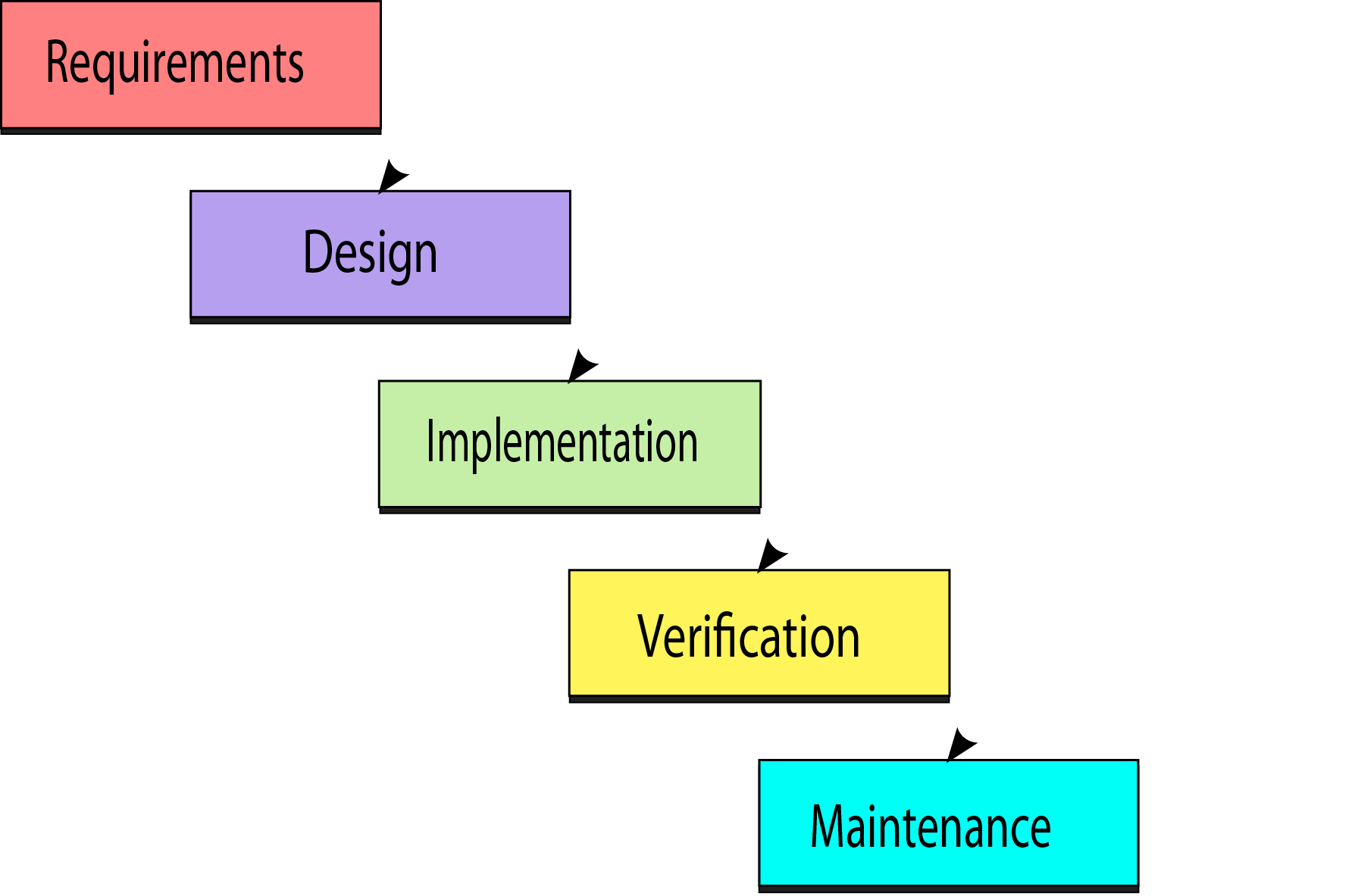
**Description:**

**Problem**: In this project, we will be developing a Hangman game that uses the MVC pattern, object-oriented design, and a graphical user interface (GUI).

Too often, programmers rush to start writing code before they have properly nailed down the requirements and settled on a solid object-oriented design. In this multi-part project, you will work with a partner, and you will do the project in phases. The overall goals of the project are to create a Hangman game and to become familiar with the software development lifecycle in a more realistic context.

**The Software Development Lifecycle – the Waterfall model**

In some ways, the software development process is similar to other product development processes. It has remained notoriously hard, however, to turn it into a generally predictable activity. The computer science subfield of software engineering, which studies software development, has proposed the Waterfall model as one useful way to conceptualize the software development lifecycle. The stages include gathering requirements, proposing a software design, coding an implementation according to that design that satisfies the requirements, testing/verifying that implementation in isolation and under anticipated usage conditions, and maintaining the software (responding to bug reports, modifications required by new usage scenarios, etc.).



**Some Gifts**

To help you make a bit quicker progress, I am supplying you with a **HangmanGraphics** class and a **GFace** helper class that provide a simple graphical view of the hangman gallows and figure that is nicely scalable. You can incorporate these classes into your thinking about requirements, design and implementation. If you get done early with the project, you may want to create your own replacement for the **HangmanGraphics** class.

The **HangmanGraphics** class implements the **ProgressivelyDrawable** interface. This interface (also provided) defines what public methods a graphical model must supply to be used by the **Hangman** class. The **ProgressivelyDrawable** interface allows the **Hangman** class to be relatively independent of the particular graphics class used for the hangman.

/\*\*

\* The graphical interface implemented by HangmanGraphics.

\* **@author** Mark Jones

\*

\*/

**public** **interface** ProgressivelyDrawable {

**void** drawNextPart(); // draw the next part of the hangman

**void** reset(); // reset the hangman

**int** getMaxParts(); // return the maximum number of parts supported by the graphics

}

We have not yet discussed file I/O, so I am also giving you a **Dictionary.java** class that can read a dictionary from a file. I also supply sample dictionary files (**dictionary.txt** and **yawl.txt**), but you should also develop a dictionary of your choosing. Another extension for those done early would be to allow the dictionary to be selectable from the GUI.

**Requirements (Project Part I, 10 pts)**

When you first set out to develop an application, you need to develop a ***requirements*** document. For a game, requirements are typically driven by the marketplace and the realities of the gaming platform – what would users want and what are we capable of providing.

In its simplest form, the requirements document is a list of the things that your application must do.

I have started you off with a short list of my requirements for your project. Note that good requirement writers use the words **must (not)**, **should (not),** and **may** very carefully. (See [http://www.ietf.org/rfc/rfc2119.txt](http://www.ietf.org/rfc/rfc2119.txt%20) for more detail on how these terms are technically used in writing requirements. See <http://www.w3.org/TR/soap12-part1/> for an example of an Internet requirements document that I was a contributor to as a member of the W3C XML Protocol Working Group.) Good requirement writers are also careful not to stray into the realm of design.

Add your own requirements to this list; you may also strengthen (but not weaken) any of my requirements. You should have at least a dozen solid requirements. Feel free to add features such as statistics on the total number or percentage of games won/lost, current winning/losing streak, saving the stats in a file to preserve them across sessions, etc. Requirements often include ***accessibility*** considerations – how big should the window be, what are the minimum font sizes for readability, etc.

Save your requirements in a separate file called “Hangman Requirements (<your last names>).doc” (or docx). When you have completed your requirements, have me check them over before proceeding. Then submit your requirements to your Google Drive Hand-In folder.

My requirements:

1. The application **must** implement the standard rules of Hangman. (See the appendix below.)
2. The application **must** provide a GUI game environment rather than a console.
3. Users **must** be able to play multiple rounds of Hangman.
4. The game **should** remain responsive (millisecond response times).
5. The game **should** work on Windows and Mac OS.
6. The application **may** be resizeable.

**Design (Project Part II, 10 points)**

The next activity, following requirements, is ***design***. Begin a document called “Hangman Design (<your last names>).doc” (or docx). This document should progressively describe your design choices at different levels of abstraction. The design activity is extremely important. If done well, it can save a tremendous amount of wasted coding energy by creating the right project structure from the outset. It helps sharpen the requirements and associate them with particular areas of the design. It guides the scheduling of programming effort on the implementation.

**High-level design**

* You are required to use the Model-View-Controller design pattern, similar to what we did in the Blackjack code. Describe how this works in the ACM Graphics world.
* Remember that the Model includes only information about the domain model – Hangman in this case. The most important consideration is that no direct information about the user interface (UI) should be present in the model.
* The View represents the knowledge about how to display the model objects in a user interface. It will offer visual representations of the gallows, the letters already guessed, the word being guessed (in progress), etc., but it will not know about the rules of the game. It is permissible to pass data model objects to the View, but only for the purpose of accessing their properties.
* There are different possible designs for the communications between Models, Views and Controllers. In our Blackjack project, the model notified the view of important events. The view updated the UI (a graphical view of the blackjack state) and communicated important start/end-of-game state to the controller. Elaborate your chosen communication strategy in your design document.

**Class design**

* Describe the classes that you anticipate using for Hangman. Classify them as part of the Model, View or Controller.
* For each class, describe its public interface. What methods (messages) will it respond to? What state does it maintain?

**UI design**

* For user interfaces classes, describe the user interface experience. For example, is it a console game or a GUI? Some design documents actually include mockups of the interface screens and user interactions, so that customer feedback can be gauged early on, before lots of effort goes into coding.

**Technology choices**

* Design documents may also include a discussion of the anticipated techniques and technology to be used in the implementation. What graphics libraries will be used? Will there be any file I/O? Databases? Are there any challenging algorithms which need to be developed?  
  When you have completed your design document, have me check it over before proceeding. Then submit your design document to your Google Drive Hand-In folder.

**Coding / Implementation / Testing (Project Part III, 25 points)**

Using your design document as your guide, use the following implementation strategy to write your code:

1. Divide the implementation effort up. One of you can concentrate on the model. The other one can concentrate on the user interface. This will help you appreciate the amount of communication often required in a real development effort.
2. Implement a bare bones version that roughs in the classes that you need and provides stubs for all of the public methods that have the right signature but no real implementation. Data can be “hardwired” into the stubs to satisfy the compiler and to check that the pieces are calling each other properly. It is at this point, before you commit to writing and debugging lots of code that you want to catch any obvious flaws in your GUI design, your user experience, your class architecture, etc.
3. When you have everything roughed in, you can start integrating the pieces together and completing the coding.
4. Documentation forms an important part of the implementation phase. It is extremely valuable during the testing and maintenance phases, which may be carried out by people other than the original coders. At the very least, make sure to include complete Javadoc documentation for each class, method, and global variable. Judicious programmer comments inside your methods can be helpful also.

**Testing**

Along the way, you will be debugging your code and unit testing parts of it. A **unit test** just exercises a small part (one or a small collection of methods) of a larger system.

There are several other types of verification or testing that need to be carried out. Software engineering has development some formal verification techniques (like proofs!), but these haven’t proven practical enough for most programming activities. Typically programmers try to test a variety of different possible situations, particularly including “edge cases” – situations at the extremes or boundaries. What if the secret hangman word contains spaces or puctuation? What if the user types a non-letter? What if the user repeats an already guessed letter? Can the user interface accommodate very long words?

In a large system, **integration tests** are broader tests that check to see that the pieces (which have been individually tested) fit together properly. **System tests under load** are tests that ensure that the system works under real world conditions. In complex systems, build (and testing!) the test harness itself can take an immense effort all by itself.

The first usable versions of a system are often called **alpha versions**. More mature versions of a system are called **beta versions**. A **release candidate** is a version very near release. **Released versions** of a system are “finished” versions. There can be major or minor releases that follow the original release (which is often designated version 1.0).

Enlist some friends or classmates in the class to be your alpha or beta testers. Try running your code in different environments (on different machines).

When you have completed your coding and testing:

1. Submit your source code, copied and pasted at the end of this Word document as usual. Name the document “Hangman Code (<your last names>).doc” (or docx). Start each class listing on a new page.
2. At the end of the document also append several screen shots that illustrate your game during play.
3. Submit your Word doc to your Google Drive Hand-In folder.

***Hangman:***

**package** unit9;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Font;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.KeyEvent;

**import** java.util.ArrayList;

**import** javax.swing.JButton;

**import** javax.swing.JLabel;

**import** acm.graphics.GImage;

**import** acm.graphics.GLabel;

**import** acm.graphics.GLine;

**import** acm.graphics.GPoint;

**import** acm.program.GraphicsProgram;

/\*\*

\* Project 11 - Hangman<br>

\* Ozaner Hansha, Armen<br>

\* Dr. Jones<br>

\* AP Computer Science<br>

\* April 4th, 2016<br>

\*/

@SuppressWarnings("serial")

**public** **class** Hangman **extends** GraphicsProgram **implements** HangmanView {

/\*\*

\* Size of the window containing the application.

\* **@see** #init()

\*/

**public** **static** **final** Dimension ***WINDOW\_SIZE*** = **new** Dimension(1000,800);

/\*\*

\* The coordinates of the {@link #graphics}

\*/

**public** **static** **final** GPoint ***GRAPHICS\_COORDS*** = **new** GPoint(50,70);

/\*\*

\* Font letters will be displayed in.

\*/

**public** **static** **final** Font ***LETTER\_FONT*** = **new** Font("Times New Roman",Font.***PLAIN***,50);

/\*\*

\* Font notifications will be displayed in.

\*/

**public** **static** **final** Font ***NOTIFICATION\_FONT*** = **new** Font("Papyrus", Font.***BOLD***, 20);

/\*\*

\* Constants relating to the lines used in displaying the letters.

\* **@see** #gameStartNotification(String)

\*/

**public** **static** **final** **int** ***LINE\_LENGTH*** = 60,***LINE\_Y*** = 600,***LINE\_GAP*** = 30;

/\*\*

\* The graphics object for this HangmanView.

\*/

**private** GHangman graphics = **new** GHangman();

/\*\*

\* The new round button.

\*/

**private** JButton newGameButton = **new** JButton("New Round");

/\*\*

\* Displays notifications about the game in the top.

\*/

**private** JLabel notifications = **new** JLabel();

/\*\*

\* The corresponding {@link HangmanModel} for method calls.

\*/

**private** HangmanModel model = **new** HangmanModel(**this**,graphics.getMaxParts()-1);

/\*\*

\* Stores letter spaces for later reference (removal).

\*/

**private** ArrayList<GLine> lineBuffer = **new** ArrayList<GLine>();

/\*\*

\* Stores letter objects for letter reference (removal/visibility).

\*/

**private** ArrayList<GLabel> letterBuffer = **new** ArrayList<GLabel>();

/\*\*

\* Initializes the GUI.

\* **@see** acm.program.GraphicsProgram#init()

\*/

@Override

**public** **void** init() {

setSize(***WINDOW\_SIZE***);

setTitle("Hangman Simulator 2016");

GImage bg = **new** GImage("src/unit9/bg.jpg");

bg.scale(.6);

add(bg,-100,0);

graphics.drawNextPart(); //Adds Gallows

add(graphics,***GRAPHICS\_COORDS***);

getGCanvas().requestFocus();

add(notifications,***NORTH***);

notifications.setFont(***NOTIFICATION\_FONT***);

notifications.setFocusable(**false**);

add(newGameButton,***SOUTH***);

newGameButton.setFocusable(**false**);

addActionListeners();

addKeyListeners();

model.newRound();

}

/\*\*

\* Catches button press events.

\* **@see** acm.program.Program#actionPerformed(java.awt.event.ActionEvent)

\*/

@Override

**public** **void** actionPerformed(ActionEvent e) {

**if**(e.getActionCommand().equals("New Round"))

model.newRound();

}

/\*\*

\* Catches enter key events.

\* **@see** acm.program.Program#keyPressed(java.awt.event.KeyEvent)

\*/

@Override

**public** **void** keyPressed(KeyEvent e) {

**if**(e.getKeyCode() == KeyEvent.***VK\_ENTER***)

model.newRound();

}

/\*\*

\* Catches keyboard key events.

\* **@see** acm.program.Program#keyTyped(java.awt.event.KeyEvent)

\*/

@Override

**public** **void** keyTyped(KeyEvent e) {

model.guess(Character.*toUpperCase*(e.getKeyChar()));

}

/\*\*

\* Sets up the given word for this round.

\* **@see** unit9.HangmanView#gameStartNotification(java.lang.String)

\*/

@Override

**public** **void** gameStartNotification(String word) {

notifications.setText("Welcome to Hangman! Type to guess a letter.");

reset();

**for**(**int** x = 0; x < word.length(); x++) {

**if**(word.charAt(x) != ' ') { //does not add a line for space characters

//Adds Line dependent on word size.

**int** length = word.length()\*(***LINE\_LENGTH***+***LINE\_GAP***)-***LINE\_GAP***;

**int** offset = (**int**)((getSize().getWidth()-length)/2);

lineBuffer.add(**new** GLine(0,0,***LINE\_LENGTH***,0));

add(lineBuffer.get(lineBuffer.size()-1),x\*(***LINE\_LENGTH***+***LINE\_GAP***)+offset,***LINE\_Y***);

//Adds letter above line and makes invisible.

GLabel label = **new** GLabel(""+word.charAt(x));

label.setFont(***LETTER\_FONT***);

label.setVisible(**false**);

letterBuffer.add(label);

add(label,x\*(***LINE\_LENGTH***+***LINE\_GAP***)+(***LINE\_LENGTH***-label.getWidth())/2+offset,***LINE\_Y***);

}

}

}

/\*\*

\* Displays winning message.

\* **@see** unit9.HangmanView#gameWonNotification()

\*/

@Override

**public** **void** gameWonNotification() {

notifications.setText("You won.");

}

/\*\*

\* Displays losing message.

\* **@see** unit9.HangmanView#gameLostNotification()

\*/

@Override

**public** **void** gameLostNotification() {

notifications.setText("You lost.");

graphics.drawNextPart();

**for**(GLabel g: letterBuffer) {

**if**(!g.isVisible()) {

g.setVisible(**true**);

g.setColor(Color.***RED***);

}

}

}

/\*\*

\* Resets the GUI.

\*/

**public** **void** reset() {

**for**(GLine g: lineBuffer)

remove(g);

**for**(GLabel l: letterBuffer)

remove(l);

lineBuffer.clear();

letterBuffer.clear();

graphics.reset();

}

/\*\*

\* Displays correct notification and shows the correct letter.

\* **@see** unit9.HangmanView#correctNotification(char)

\*/

@Override

**public** **void** correctNotification(**char** c) {

notifications.setText("Nice Job!");

**for**(GLabel g: letterBuffer)

**if**(g.getLabel().equals(""+c))

g.setVisible(**true**);

}

/\*\*

\* Displays incorrect notification.

\* **@see** unit9.HangmanView#incorrectNotification(char)

\*/

@Override

**public** **void** incorrectNotification(**char** c) {

notifications.setText("That letter was incorrect. Try again.");

graphics.drawNextPart();

}

/\*\*

\* Displays already guessed notification.

\* **@see** unit9.HangmanView#alreadyGuessedNotification(char)

\*/

@Override

**public** **void** alreadyGuessedNotification(**char** c) {

notifications.setText("That letter has already been guessed. Try again.");

}

/\*\*

\* Starts the program.

\* **@param** args - No arguments expected.

\*/

**public** **static** **void** main(String[] args) {

**new** Hangman().start();

}

}

***HangmanView***

**package** unit9;

/\*\*

\* Interface for all Hangman games.

\* **@author** Ozaner Hansha

\*/

**public** **interface** HangmanView {

/\*\*

\* Starts the game with the given word.

\* **@param** word - current word of the round.

\*/

**void** gameStartNotification(String word);

/\*\*

\* Tells view game has been lost.

\*/

**void** gameLostNotification();

/\*\*

\* Tells view game was won.

\*/

**void** gameWonNotification();

/\*\*

\* Notifies the view of a correct guess.

\* **@param** c - the guessed letter.

\*/

**void** correctNotification(**char** c);

/\*\*

\* Notifies the view of an incorrect guess.

\* **@param** c - the guessed letter.

\*/

**void** incorrectNotification(**char** c);

/\*\*

\* Notifies the view of a letter that has already been guessed.

\* **@param** c - the guessed letter.

\*/

**void** alreadyGuessedNotification(**char** c);

}

***HangmanModel:***

**package** unit9;

**import** java.util.HashSet;

**import** java.util.Set;

/\*\*

\* Model for {@link Hangman} game. Stores state of game

\* & notifies {@link HangmanView} of events.

\* **@author** Ozaner Hansha

\*/

**public** **class** HangmanModel {

/\*\*

\* Regular expression for Alphabetic characters.

\*/

**public** **static** String *VALID\_CHARACTERS* = "[a-zA-Z]";

/\*\*

\* Reference to corresponding {@link HangmanView}.

\*/

**private** HangmanView view;

/\*\*

\* Amount of guesses until loss.

\*/

**private** **int** maxGuesses, guesses;

/\*\*

\* Used to obtain a random word.

\*/

// private Dictionary dictionary = new Dictionary("src/unit9/yawl.txt");

**private** Dictionary dictionary = **new** Dictionary();

/\*\*

\* The current word in play.

\*/

**private** String currentWord;

/\*\*

\* A set of guessed letters.

\*/

**private** Set<Character> guessedLetters = **new** HashSet<Character>();

/\*\*

\* Whether the game is over or not.

\*/

**private** **boolean** gameOver = **false**;

/\*\*

\* Constructor for {@link HangmanModel}.

\* **@param** view - A corresponding {@link HangmanView}.

\*/

**public** HangmanModel(HangmanView view, **int** maxGuesses) {

**this**.view = view;

**this**.maxGuesses = maxGuesses;

}

/\*\*

\* Resets variables and starts new round.

\*/

**public** **void** newRound() {

guesses = 0;

guessedLetters.clear();

gameOver = **false**;

view.gameLostNotification();

**do** {

currentWord = dictionary.getRandomWord().toUpperCase();

} **while**(currentWord.length() > 11);

view.gameStartNotification(currentWord);

}

/\*\*

\* Returns whether or not the game has been won.

\* **@return** - Win status of the game.

\*/

**public** **boolean** gameWon() {

//If the set of characters in the currentWord is a subset of the guessed letters.

//Then game has been won.

Set<Character> s = **new** HashSet<Character>();

**for**(**char** c: currentWord.toCharArray()) {

s.add(c);

}

s.remove(' ');

**return** guessedLetters.containsAll(s);

}

/\*\*

\* Checks if guess is valid and guesses if so.

\* Ends game if guesses exceed {@link #maxGuesses} or {@link #gameWon()} is true.

\* **@param** c - Letter being guessed.

\*/

**public** **void** guess(**char** c) {

**if**(!gameOver) {

**if**(!guessedLetters.contains(c)) {

**if**(isValidGuess(c)) {

guessedLetters.add(c);

**if**(currentWord.indexOf(c) != -1) {

view.correctNotification(c);

**if**(gameWon()) {

view.gameWonNotification();

gameOver = **true**;

}

}

**else** {

guesses++;

**if**(guesses >= maxGuesses) {

view.gameLostNotification();

gameOver = **true**;

}

**else**

view.incorrectNotification(c);

}

}

}

**else**

view.alreadyGuessedNotification(c);

}

}

/\*\*

\* Checks whether a guess is valid (i.e alphabetic).

\* **@param** c - a character.

\* **@return** The validity of a guess.

\*/

**public** **boolean** isValidGuess(**char** c) {

**return** (""+c).matches(*VALID\_CHARACTERS*);

}

}

***GHangman:***

**package** unit9;

**import** acm.graphics.GCompound;

**import** acm.graphics.GImage;

/\*\*

\* **@author** Ozaner Hansha

\*

\*/

**public** **class** GHangman **extends** GCompound **implements** ProgressivelyDrawable {

/\*\*

\* Number of parts this object has.

\*/

**public** **static** **final** **int** ***MAX\_PARTS*** = 7;

/\*\*

\* Array of Hangman images used.

\*/

**public** **static** **final** GImage[] ***HANGMAN\_IMAGES*** = **new** GImage[7];

**static** { //Initializes HANGMAN\_IMAGES

**for**(**int** x = 0; x < ***MAX\_PARTS***; x++) {

***HANGMAN\_IMAGES***[x] = **new** GImage("src/unit9/hangman" + x + ".png");

}

}

/\*\*

\* The current index of the compound.

\*/

**private** **int** currentPart = 1;

/\*\*

\* Assembles this object.

\*/

**public** GHangman() {

**for**(GImage g: ***HANGMAN\_IMAGES***) {

add(g);

}

scale(.5);

}

/\*\*

\* Draws the next part in this object.

\*/

@Override

**public** **void** drawNextPart() {

**if**(currentPart < ***MAX\_PARTS***) {

***HANGMAN\_IMAGES***[currentPart].setVisible(**true**);

currentPart++;

}

}

/\*\*

\* Resets this object.

\*/

@Override

**public** **void** reset() {

currentPart = 1;

**for**(GImage g: ***HANGMAN\_IMAGES***) {

g.setVisible(**false**);

}

***HANGMAN\_IMAGES***[0].setVisible(**true**);

}

/\*\*

\* Returns the max amount of parts this object contains.

\*/

@Override

**public** **int** getMaxParts() {

**return** ***MAX\_PARTS***;

}

}

***Screenshots:***