Space Potaters

***Detailed Design Document***

Team Ramrod*:*  
Anderson, Travis  
Birrenkott, Chris  
Tan, Joshua  
Wako, Shambel

CSCI 161  
Section 1

December 7, 2011

**Abstract:**

This purpose of this document is to give a detailed description of the design for the Space Potaters software. This document should be used by anyone who requires a more comprehensive understanding of the Space Potaters software than is presented in the Space Potaters Software Requirements Specification (SRS), including, but not limited to, software developers, project managers, and software testers. This document explains the design of the Space Potaters software at various levels of abstraction.

**Table of Contents:**

|  |  |  |
| --- | --- | --- |
| *Section* | *Topic* | *Page(s)* |
| **1** | **Introduction** | 1 |
| 1.1 | Purpose of System | 2 |
| 1.2 | Design Methodology Used | 2 |
| 1.3 | Definitions, Acronyms, and Abbreviations | 2 |
| 1.4 | Overview of Document | 2 |
| 1.5 | References | 3 |
| **2** | **Proposed Software Architecture** | 3 |
| 2.1 | Overview | 3 |
| 2.2 | Subsystem Decomposition | 4 |
| 2.3 | Persistent Data Management | 5 |
| **3** | **Class & Object Design** | 6 |
| 3.1 | Overview | 6 |
| 3.2 | Detailed Class Design | 8 |
| 3.3 | Object Interaction | 16 |
| **4** | **Glossary** | 18 |
| **5** | **Appendix** | 18 |
| 5.1 | Appendix A – Use Case Diagrams | 18 |
| 5.2 | Appendix B – Detailed Class Diagrams | 22 |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason For Changes** | **Version** |
| Team Ramrod | 11/04/2011 | Initial document. | 1.0 |
| Team Ramrod | 12/07/2011 | Updated minimal class diagram, updated game model package diagram, removed Figure 14 (collision example sequence diagram, reason: obsolete), updated Appendix B. | 1.1 |

**1. Introduction:**

The Space Potaters software is being created as a group project for the NDSU Fall 2011 course CSCI 161. An emphasis is placed on the proper use of a Software Requirements Specification (SRS) and Detailed Design Document (DDD) to aid in the design of the software, over the complete implementation of the design. This is due to the time constraint of a single semester to complete the project.

The SRS and DDD were created using a “wish-list” mindset, in which the design created represents what the complete software should look like, given the time to complete it. Despite this, the specification and design for a software that presented both a functional web-based game and a downloadable offline-based game had to be reduced to a functional downloadable offline-based game only – again, due to time constraints. However, the design for the Space Potaters software allows it to be easily upgraded to include this functionality, since the game model and logic would remain mostly unchanged in the transition.

**1.1. Purpose of System:**

The purpose of the Space Potaters software is to provide an entertaining experience to the user. It will allow the user to play the Space Potaters game, a variation of the classic Space Invaders game. The Space Potaters variation implements a different theme than the original, while maintaining the original game logic and design. It also implements an arcade-like record system, in which the top ten scores achieved by users are recorded. This will increase the replay value of the game and allow for different users to compete against each other within the Space Potaters game.

**1.2. Design Methodology Used:**

Although a strict software design methodology is not employed in the development of the Space Potaters software, the development utilizes an iterative and incremental approach to the software. In this way, higher priority software components are completed before lower priority ones. This is important, since due to the limiting time constraint, not all software components may be fully implemented by the time of the software delivery date.

The design uses UML models to aid in the communication of particular design elements. The UML models used include use case diagrams, a package diagram, class diagrams, a state chart diagram, and sequence diagrams.

**1.3. Definitions, Acronyms, Abbreviations:**

*GUI* – Abbreviation for “graphic user interface”.

*MVC*- Model–view–controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering.

**1.4. Overview of the Document:**

This document can be divided into two primary sections, varying in the level of abstraction used to describe the design. The first section, “Proposed Software Architecture”, explains the design from a top down approach. It begins with an overview of the package divisions and a brief explanation of the subsystem that each package encompasses. The major subsystems are then described in greater detail, with references to any associated use cases (located in Appendix A). It concludes with description of the techniques employed in the management of persistent data.

The second primary section explains the software on a more detailed scale, including a description of classes and interfaces used, as well as the relationships among them. A map of these relationships is provided using minimal class diagrams (stripped of class fields and methods).The purpose of each class and its general use in the Space Potaters software in then described, with references to the corresponding detailed class diagram located in Appendix B. The section ends with sequence diagrams and a state chart diagram to illustrate object interaction for critical and/or complex functionality in the software.

A glossary is located at the end of the document to define any uncommon or unclear terminology, as are the aforementioned Appendix A and Appendix B, containing the use case diagrams and detailed class diagrams, respectively.

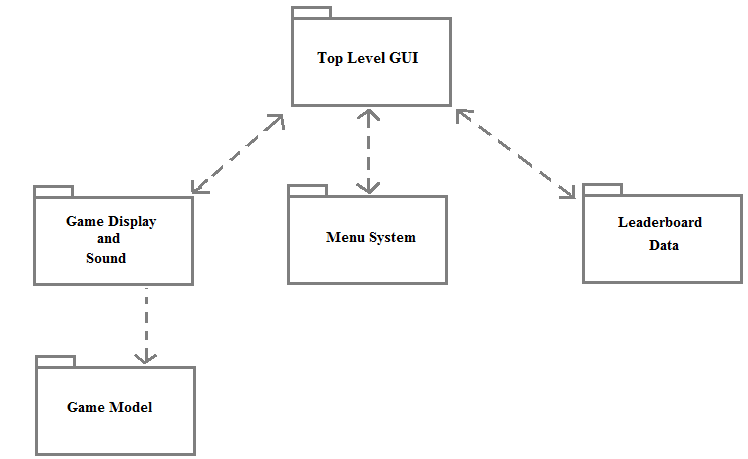
**1.5. References:**

1. CS 161 Class Notes/Lecture Slides and Design Document (DD) Template, Dr. S. Abufardeh
2. UML diagram information : http://www.ibm.com/developerworks/rational/library/
3. Java Serialization API: http://java.sun.com/developer/technicalArticles/Programming/serialization/
4. MVC Defintion: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
5. Anderson, J., & Franceschi, H. (2012). *Java illuminated: An active learning approach* (3rd ed.). Sudbury, MA: Jones & Bartlett Learning.
6. Davison, A. (2005). *Killer game programming in Java*. Cambridge, MA: O’Reilly Media.
7. Glass, G. (n.d.). Space invaders 101 – An accelerated Java 2D tutorial. Retrieved from http://www.cokeandcode.com/info/tut2d.html.

**2. Proposed Software Architecture:**

**2.1. Overview:**

The Space Potaters software is divided into five packages, representing the major subsystems involved.

 **Figure 1:** Package Diagram

Each package represents a collection of classes that are functionally related to each other. In the case of the Top Level GUI package, the collection consists of a single class, since this class cannot be grouped with any other package in a meaningful way.

The Top Level GUI package serves as the base user interface and manages the GUI components for the Leaderboard Data package, Menu System package, and Game Display and Sound package. The Leaderboard Data package manages record data for user scores. The Menu System package is used to navigate the user around the software. The Game Display and Sound package and the Game Model package manage the presentation and logic of the game, respectively.

**2.2. Subsystem Decomposition:**

Top Level GUI Package

The Top Level GUI package is comprised of a single class (the SpacePotaters class) and is concerned with managing the different GUI components to present a single unified user interface to the user. It also monitors the current status of the window in which the game is being played, and will perform different processes for different states. The three main actions performed on the main GUI window are minimizing, maximizing, and exiting. *Figure A.1* (located in Appendix A) shows the use case diagram for these different actions, as well as their associated processes.

Leaderboard Data Package

The Leaderboard Data package consists of two classes: Leaderboard and Record. This package deals with saving, retrieving, and storing user score records. The only way in which the user interacts with this package is if he ends a Space Potaters game with a score that ranks in the top ten for recorded scores. In this case, a new record would be created, which would involve recording the name to be stored with that record. This is illustrated in the use case diagram *Figure A.2*.

Menu System Package

The Menu System package is comprised of three classes: MainMenu, ChooseDifficulty, and InstructionView. The function of this package is to present a menu of options to the user and branch out to other parts of the software based on the user input. The different use cases in the menu correspond to the user clicking on the different menu buttons: “Quickstart”, “Start Game (Custom Difficulty)”, “View Leaderboards”, “View Instructions”, and “Exit”. These use cases are illustrated in the use case diagram *Figure A.3*.

Game Display and Sound Package

The Game Display and Sound package contains all those classes that relate to the visual or audile presentation of the game. This includes the library classes for the sound effects and sprites. This package will regularly monitor the Game Model package and use the information retrieved to determine its current behaviors. The ways in which the user directly interacts with this package are by manually pausing /exiting the game or by submitting requests to act on Gollum. In addition to pausing the game by minimizing the application window (handled by the Top Level GUI package), the user can pause the game by pressing a key. Exiting the current game and returning to the main menu can also be accomplished by pressing a key. *Figure A.4* shows these two abilities.

Game Model Package

The Game Model package consists of all the various components related to the game logic. Classes representing the different game entities are located in this package. Objects of these classes interact with each other according to the defined game logic. One thing to note is the unidirectional association between the Game Display and Sound package and the Game Model package. This was designed intentionally, so as to keep the game-viewing components of the game as separate as possible from the game model, keeping with the model-view-controller (MVC) architectural pattern.

Changes to the Game Model package are accomplished indirectly by the user pressing keys (handled via GameView). Besides pausing and exiting the game with the ‘P’ and ‘Esc’ keys, respectively, the other in-game actions the user can perform include shooting with Gollum and moving Gollum. Use case diagram *Figure A.5* illustrates these actions.

**2.3. Persistent Data Management:**

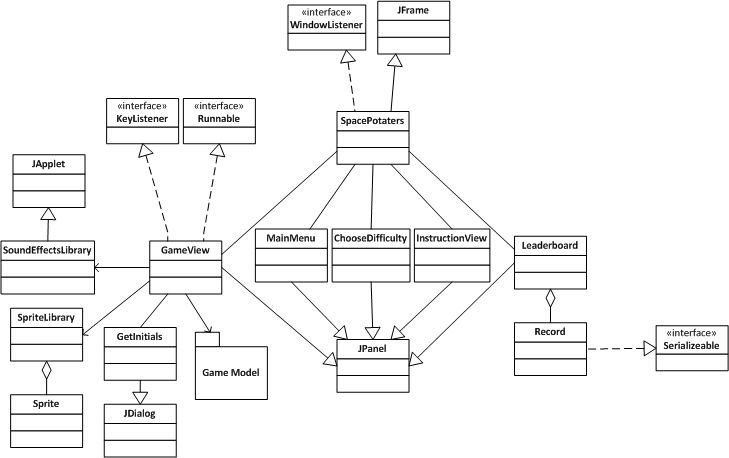
The only data that is to be stored in between sessions of the application are the leaderboard records for the top ten scores achieved by users. Since this data is relatively simple and small in size, this software will implement persistent objects using the Java Serialization API. To accomplish this, the Record class implements the Java interface Serializeable. The Record objects, which will be stored in an ArrayList, can then be easily serialized and deserialized, since ArrayList also implements Serializeable. Although this approach is implementation specific, i.e. the records will not be compatible if the structure for the Record class is changed, it is still an acceptable solution in this case. The structure of the Record class should not change from its initial design.

Since the only personal information the records store is a name (which the user is free to pick), there are no security requirements for the persistent data.

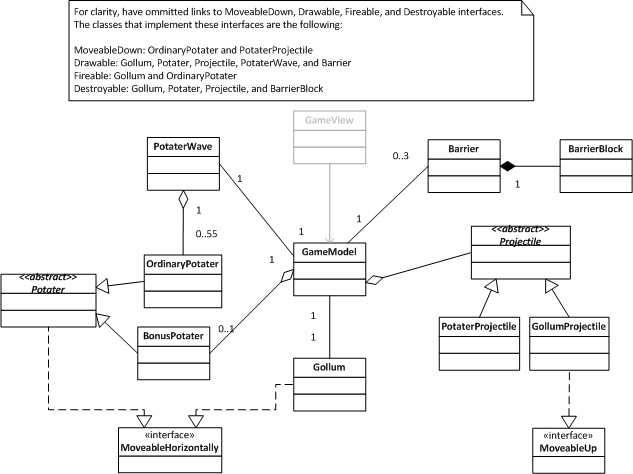
**3. Class & Object Design**

**3.1. Overview:**

The following minimal class diagrams show the relationships between classes used in this software. The Game Model package is abbreviated in *Figure 2* for the sake of clarity. The Game Model package is expanded upon in *Figure 3*, where its classes and their relationships are illustrated.



**Figure 2:** Minimal Class Diagram (Game Model abbreviated)



**Figure 3:** Class Diagram for Game Model package

The SpacePotaters class acts as the top-level GUI and window event handler. The MainMenu class serves as the starting and returning point for the user interface. Functionality from the Leaderboard class is displayed when the user chooses to view the game leaderboards from the main menu.

The actual Space Potaters game is also accessed via the MainMenu class. The Space Potaters game is divided into a view component and a model component. The view component consists of the GameView class, which houses the game animation loop, and the library classes – SoundEffectsLibrary and SpriteLibrary. SoundEffectsLibrary serves as the sound storage class, as well as sound player, while SpriteLibrary stores the game’s Sprite objects, i.e. the graphical representations of the game components.

The backbone of the model component is the GameModel class. GameModel monitors the interactions between the following classes: PotaterWave, Potater, OrdinaryPotater, BonusPotater, Gollum, Projectile, PotaterProjectile, GollumProjectile, Barrier, and BarrierBlock. The OrdinaryPotater and BonusPotater classes are subclasses of Potater and represent the enemy. PotaterWave represents the group of Potaters invading. Gollum is the user-controlled shooter, who is protected by Barriers. A Barrier is subdivided into BarrierBlocks to facilitate the simulation of incremental damage. The PotaterProjectile and GollumProjectile classes are subclasses of Projectile and represent the bullets/missiles shot between the two sides.

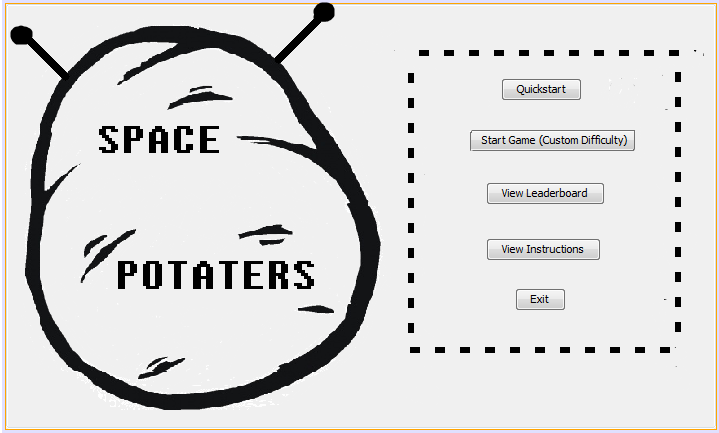
**3.2. Detailed Class Design:**

SpacePotaters (*Figure B.1)*:

SpacePotaters extends JFrame and implements WindowListener. It is the top-level GUI for the Space Potaters software. It contains references to the JPanel components it needs to switch between (GameView, MainMenu, Leaderboard, ChooseDifficulty, and InstructionView) and the corresponding switching methods. It also contains the WindowListener interface methods for pausing/exiting on different window events.

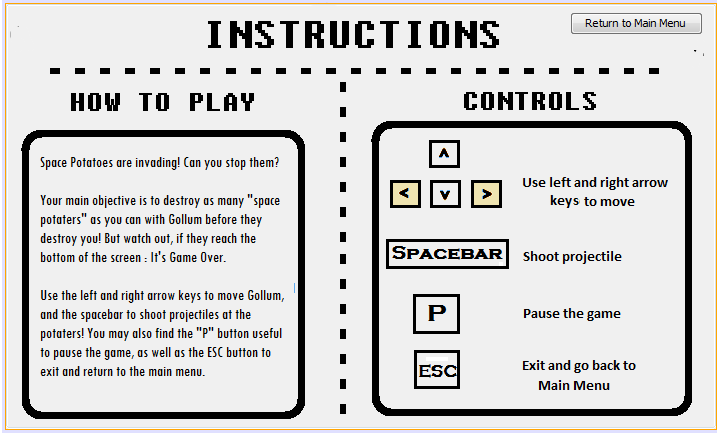
MainMenu (*Figure B.2*):

MainMenu extends JPanel and is visible inside SpacePotaters whenever the user is viewing the main menu. It consists of JButton components corresponding to the different menu options the user can choose between, as well as a reference to the top-level SpacePotaters object, so it can call its switching methods when needed.



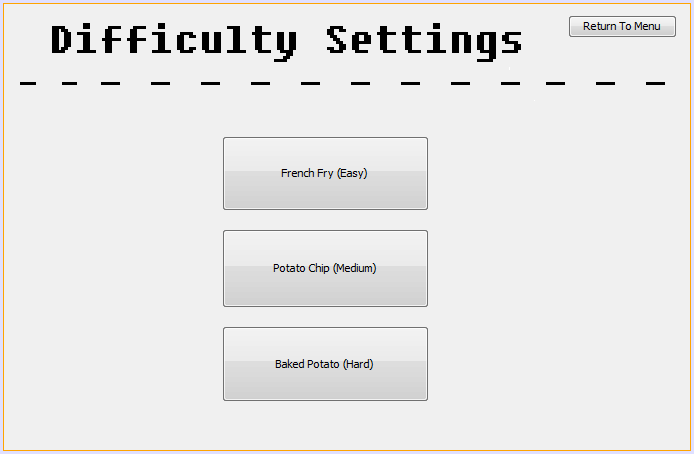
**Figure 4:** Screenshot of Main Menu

*Figure 4* shows a screenshot of the GUI generated by the MainMenu class.

**Figure 5:** Screenshot of Instructions Display

Clicking the “View Instructions” button will display the screen shown above in *Figure 5*.

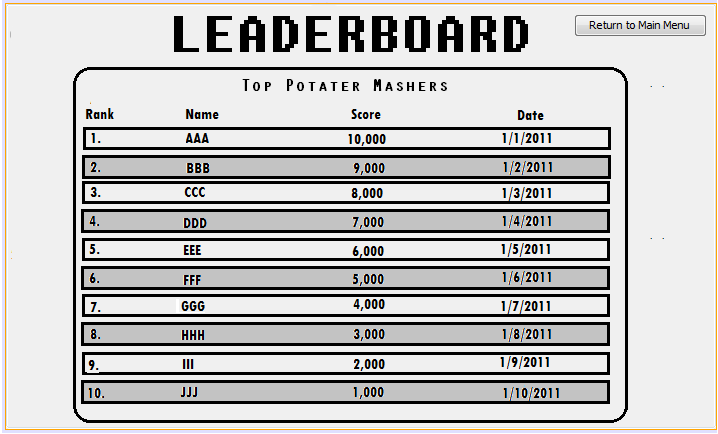
The user can launch into the game in two ways, as explained in Use Case Diagram A.3. The first is to click “Quickplay” from the main menu. This will start the game with the default difficulty. The other way is to click “Start Game (Custom Difficulty)”, which will display a screen similar to *Figure 6* before launching into the game.



**Figure 6:** Screenshot of Difficulty Settings Display

Leaderboard (*Figure B.3*):

This Leaderboard class extends JPanel and is displayed when the user chooses to view the leaderboards. It is also in control of saving and loading the record file. When the user finishes a game, the getLowestScore() method will be called to determine if a new record should be created. If so, then the appropriate methods of Leaderboard will be called to create a record, including the saveToFile() method.



**Figure 7:** Screenshot of Leaderboard Display

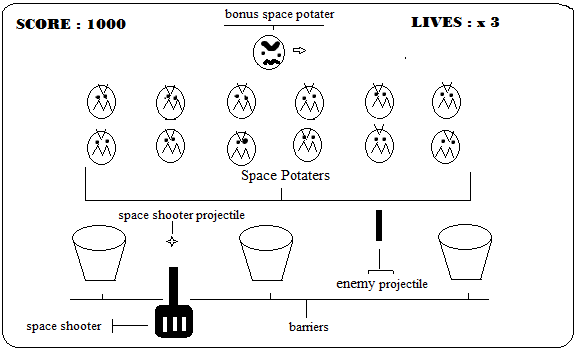
The display associated with the Leaderboard class is shown above in *Figure 7*.

Record (*Figure B.4*):

A Record consists of three fields to store the user’s initials, the date the record was created, and the score of the record. Record implements Serializeable so that it can be saved to and loaded from a file.

GameView (*Figure B.5*):

The GameView class is the visual aspect of the Space Potaters game. It implements KeyListener to monitor if the user pauses/ exits the game or requests to move/shoot with Gollum and implements Runnable for its animator Thread. It uses NO\_DELAYS\_PER\_YIELD, MAX\_FRAME\_SKIPS, and DEFAULT\_FPS to run the animation thread at a near constant frame rate (or update rate). It also uses dbGraphics and dbImage to allow for the technique of double buffering, which helps to avoid a jittery/jumpy animation loop. The run() method of GameView consists of three primary components : update the game model, render the game model to the screen, and sleep for a variable amount of time.



**Figure 8:** In-game display

*Figure 8* shows the general layout of the in-game display generated by GameView.

When the game ends, one of two cases can occur: the user scored high enough to create a new record, or the user did not score high enough to create a new record. *Figure 9* (below) shows the view displayed when the user did not score high enough, and *Figure 10* (below) shows the view displayed when the user has scored high enough.



**Figure 9:** Screenshot of Game Over Display



**Figure 10:** Screenshot of New Record Display

SoundEffectsLibrary (*Figure B.6*):

This class is used to store a static instance of the soundStorage, so audio files need to be loaded only once. It extends JApplet to allow it to play AudioClip’s.

SpriteLibrary (*Figure B.7*):

Similar to SoundEffectsLibrary, this class is used to store a static instance of a SpriteLibrary, so image files need to be loaded only once. It contains a HashMap that maps String names to Sprites.

Sprite (*Figure B.8*):

A Sprite is a wrapper class for an Image. It provides additional methods to determine the height and width of the sprite image.

GameModel (*Figure B.9*):

GameModel is the class representing the Space Potaters game logic. It contains references to the various game model components (PotaterWave, Gollum, Barrier, Projectile, etc.), as well as fields to track the current game statistics, such as lives and score.

The update( ) method of GameModel simulates a single loop/step of gameplay. Inside this method, entities may be moved, a projectile collision may be detected, a potater may randomly shoot, etc. It uses a brute force method to check every projectile for any collision.

Gollum (*Figure B.10*):

The Gollum class represents the user-controlled space shooter. It implements the following interfaces: Destroyable, MoveableHorizontally, Fireable, and Drawable. The main methods of Gollum are those to move and shoot.

Note: The methods of Gollum should all be synchronized, since the Gollum object might be updated by the event thread of the GameView (the key listener methods) while the animator thread of GameView is examining or drawing it.

Potater (*Figure B.11*):

Potater is an abstract class representation of a generic potater enemy. Different potaters may have different point values. Potater implements the Destroyable, MoveableHorizontally, and Drawable interfaces.

OrdinaryPotater (*Figure B.12*):

OrdinaryPotater is a subclass of Potater and represents a potater that moves down the screen towards Gollum. Unlike the BonusPotater and Gollum, which move in a continuous fashion, an OrdinaryPotater moves in discrete steps and moves once every waitPeriod. OrdinaryPotater objects can also randomly shoot at Gollum. OrdinaryPotater implements the Fireable and MoveableDown interfaces.

BonusPotater (*Figure B.13*):

BonusPotater is a subclass of Potater. A BonusPotater occasionally floats across the top of the game screen and represents an opportunity for the user to shoot it down and gain extra points. Unlike the OrdinaryPotater objects, a BonusPotater object moves in a continuous manner. BonusPotater implements the Drawable interface.

PotaterWave (*Figure B.14*):

PotaterWave is used to track the group of OrdinaryPotaters present at any time in a game. It contains general methods used to move the potaters as a group. It also can check to see if the potaters are at the left or right edge of the screen, or if they have successful invaded Gollum’s planet. PotaterWave implements the Drawable interface.

Projectile (*Figure B.15*):

Projectile is an abstract class that represents a generic projectile in the game. It has a method used to check if it collides with an object implementing the Destroyable interface. Projectile implements the Drawable and Destroyable interface.

PotaterProjectile (*Figure B.16*):

PotaterProjectile extends Projectile. It represents a projectile shot from an OrdinaryPotater and contains a reference to its shooter. It implements the MoveableDown interface.

GollumProjectile (*Figure B.17*):

GollumProjectile extends Projectile. It represents a projectile shot from Gollum and implements the MoveableUp interface.

Barrier (*Figure B.18*):

The Barrier class represents the protective barrier that shields Gollum from enemy fire (or possibly potaters from Gollum’s fire). It is modeled as an array of BarrierBlock objects, which help to simulate incremental damage on the barriers as they get shot. Barrier implements the Drawable interface.

BarrierBlock (*Figure B.19)*:

BarrierBlock objects represent the internal structure of a Barrier. They contain an isActive field to keep track of whether they have been destroyed or not (i.e. whether a portion of the barrier has been destroyed or not). BarrierBlock implements the Destroyable interface.

MoveableHorizontally (*Figure B.20*):

The MoveableHorizontally interface contains methods to move left or move right.

MoveableUp (*Figure B.21*):

The MoveableUp interface contains a single method to move up.

MoveableDown (*Figure B.22*):

The MoveableDown interface contains a single method to move down.

Drawable (*Figure B.23*):

The Drawable interface contains a single draw method and should be implemented by any classes which need to be drawn to the screen.

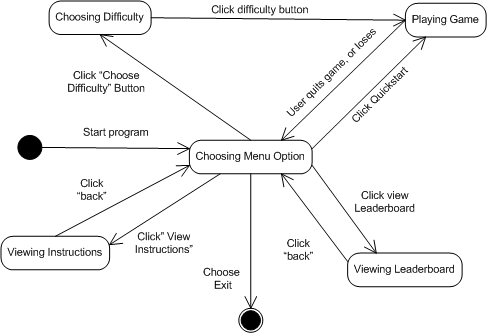
Fireable (*Figure B.24*):

The Fireable interface contains a single method to shoot and is implemented by OrdinaryPotaters and Gollum.

Destroyable (*Figure B.25*):

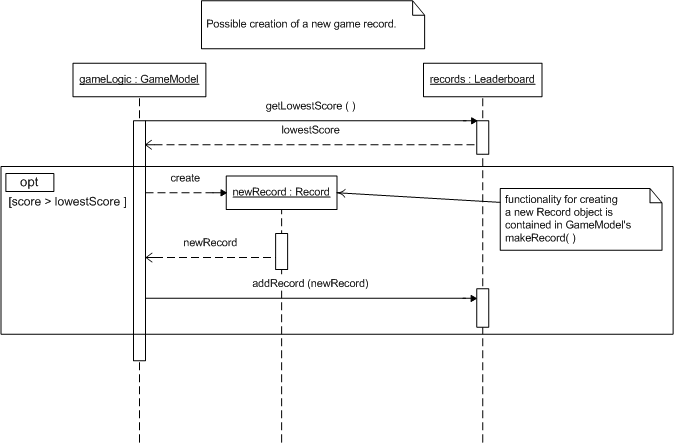
The Destroyable interface contains a method returnRectRep(), which return a Rectangle representation of an object. This Rectangle is used with a Projectile’s Rectangle to check if an intersection occurs. If it does, this signals a collision.

**3.3. Object Interaction:**



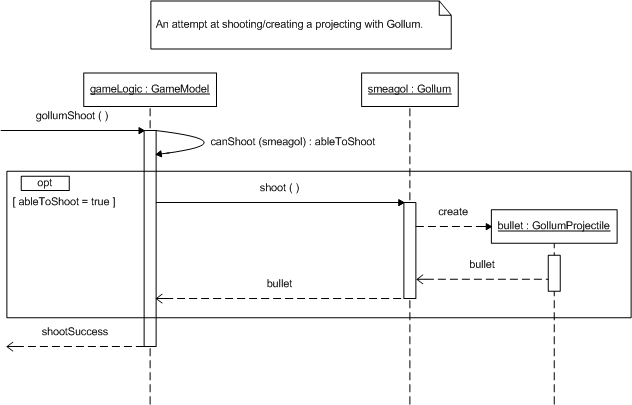
**Figure 11:** Main Menu State Chart Diagram

The different states and state transitions of the main menu are show above, in *Figure 11*. This diagram corresponds to Use Case Diagram A.3 in Appendix A.



**Figure 12:** New Record Sequence Diagram

*Figure 12* shows the sequence diagram for a possible new game record. Based on whether the user’s score is higher than the lowest score in the leaderboard, gameLogic will create and store a new record. If the record array is sorted, getLowestScore( ) can be simplified to getting the last entry. Use Case Diagram A.2 outlines this basic functionality.



**Figure 13:** Gollum Shooting Sequence Diagram

The sequence diagram for attempting to shoot with Gollum is shown in *Figure 13*. The events inside the ‘opt’ combined fragment will only execute if Gollum is able to shoot (i.e. does not already have an active projectile in the game).

**4. Glossary:**

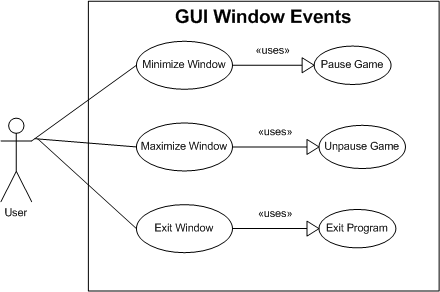
*Gollum* - The antihero of the Space Potaters game. Gollum is the potater-hating character from J.R.R Tolkien’s *The Lord of the Rings*. The user controls Gollum by moving him left or right and shooting at the oncoming evil potaters.

*Potaters* – The enemy of the Space Potaters game. Potaters are evil forms of potatoes that are trying to invade Gollum’s home planet (Middle Earth). They also shoot projectiles that can damage Gollum’s protective barriers or kill Gollum himself. They must be destroyed before they reach Gollum or else the game is ended.

**5. Appendix:**

**5.1. Appendix A – Use Case Diagrams:**

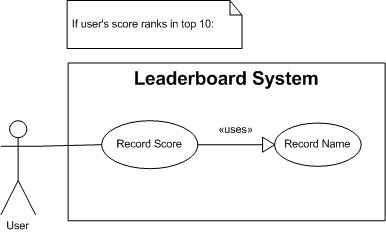
Use Case Diagram A.1 :



**Figure A.1:** GUI Window Events

This use case diagram shows the different actions the user can take while playing Space Potaters, in relation to the GUI window, as well as the processes associated with these actions. At any point in the game, the user has the ability to minimize, maximize, or exit the window containing the Space Potaters gameplay. The GUI will respond differently to each of these actions. Whenever the user minimizes the window, the game will automatically pause, saving the current state. Similarly, whenever the user maximizes the window, the game will automatically unpause. This will allow for a seamless transition whenever the user needs to take a break from the game. Lastly, exiting the window should trigger the obvious result in the program, namely to terminate.

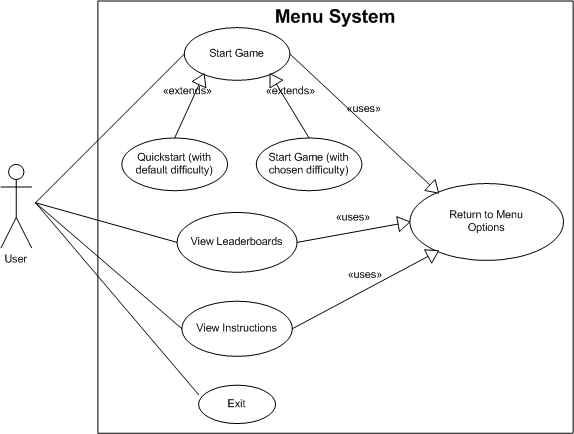
Use Case Diagram A.2:



**Figure A.2:** Leaderboard System

Use Case A.2 demonstrates the action performed by the user in the case that his score ranks in the top ten of those recorded. Whenever a score is a recorded into the leaderboard system, a name is also recorded to be put on file with that recorded score.

Use Case Diagram A.3:

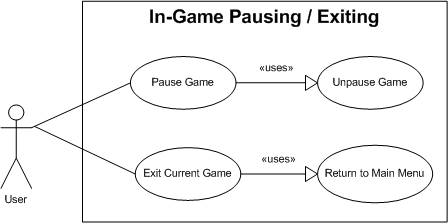


**Figure A.3:** Menu System

The different menu options include starting a game (with default or custom difficulty), viewing the leaderboards, viewing the gameplay instructions, and exiting the program. This use case diagram shows each of these. Quickstart and starting a game with a chosen difficulty are really just two specific ways in which to start a game. If the user does not choose to exit the game and instead chooses another option, he will eventually end up at the main menu again, before exiting the game.

Note: The user could end the game elsewhere, without returning to the main menu, by closing the application window. The reason this is not explicitly shown is because it is treated as an exception by the software. The desirable way for the user to end a game is through the main menu, as this will also ensure that any persistent data is saved correctly.

Use Case Diagram A.4:

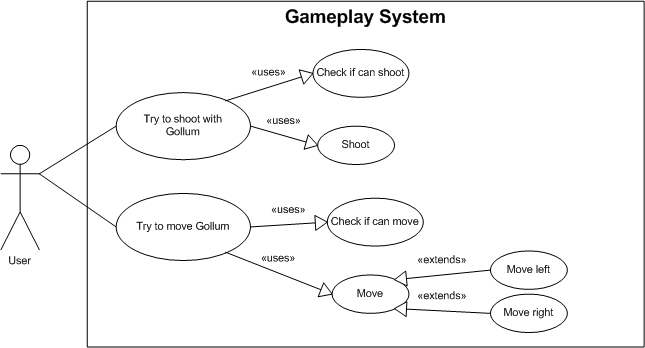


**Figure A.4:** In-Game Pausing/Exiting

Within a gameplay session of Space Potaters, the user can pause or exit the game by pressing the ‘P’ or ‘Esc’ key, respectively. These use cases are handled by the Game Display and Sound package. If the user exits the game by pressing the ‘Esc’ key, then he will be transported to the main menu.

Note: If the user exits the game in this way, his score will not be allowed to be entered in the leaderboards, if applicable.

Use Case Diagram A.5:



**Figure A.5:** Gameplay System

The two primary actions the user can perform to influence the game are shooting with Gollum and moving Gollum. Gollum is restricted to moving left and right within the bounds of the Space Potaters game window.

When the user presses the spacebar with the intention of shooting, the gameplay system first checks to see if Gollum is allowed to shoot. Gollum is allowed to shoot only if he does not have a current active projectile already in the game. If he doesn’t, then he shoots a new projectile.

When the user presses the left or right arrow with the intention of moving in that direction, the gameplay system first checks to see if moving left or right would move Gollum outside the display window of the game. If it does not, then the system will update the game model to move Gollum accordingly.

**5.2. Appendix B – Detailed Class Diagrams:**

|  |
| --- |
| **SpacePotaters** |
| - WINDOW\_HEIGHT : int - WINDOW\_WIDTH : int - gameView : GameView - mainMenu : MainMenu - leaderboard : Leaderboard - cLayout : CardLayout - difficulty : ChooseDifficulty - instructions : InstructionView - panelViews : JPanel |
| + switchToGameView ( ) : void + switchToMainMenu ( ) : void + switchToLeaderboard ( ) : void + viewInstructions ( ) : void + chooseDifficulty ( ) : void + windowActivated (e : WindowEvent) : void + windowClosed (e : WindowEvent) : void + windowClosing (e : WindowEvent) : void + windowDeactivated (e : WindowEvent) : void + windowDeiconified (e : WindowEvent) : void + windowIconified (e : WindowEvent) : void + windowOpened (e : WindowEvent) : void |

**Figure B.1:** SpacePotaters Class Diagram

|  |
| --- |
| **MainMenu** |
| - MAIN\_MENU\_IMAGE\_LOCATION : String - mainMenuBackground : Image - spTop : SpacePotaters  - quickStartButton : JButton  - chooseDifficultyButton : JButton - viewInstructionsButton : JButton - viewLeaderboardsButton : JButton - exitButton : JButton |
| + paintComponent(g : Graphics) : void |

**Figure B.2:** MainMenu Class Diagram

|  |
| --- |
| **Leaderboard** |
| - DEFAULT\_SAVE\_LOCATION : String - LEADERBOARD\_IMAGE\_LOCATION : String - PLACEHOLERS\_FILE : String - records : ArrayList <Record> - leaderboardBackground : Image - returnButton : JButton - spTop : SpacePotaters |
| + addRecord (newRecord : Record) : void - copyPlaceholders( ) : void + isNewRecord(playerScore : int) : Boolean - loadFromFile ( ) : boolean - saveToFile ( ) : boolean + loadFromFile ( ) : boolean + paintComponent (g : Graphics ) : void + paintLeaders (g : Graphics) : void |

**Figure B.3:** Leaderboard Class Diagram

|  |
| --- |
| **Record** |
| - userInitials : String - date : Date - score : int |
|  |

**Figure B.4:** Record Class Diagram

|  |
| --- |
| **GameView** |
| - spTop : SpacePotaters - LIVES\_DISPLAY\_SPACER : int - NO\_DELAYS\_PER\_YIELD : int - MAX\_FRAME\_SKIPS : int - DEFAULT\_FPS : int  - PANEL\_WIDTH : int - PANEL\_HEIGHT : int - game : GameModel - animator : Thread - gameRunning : boolean - gameOver : boolean - dbGraphics : Graphics - dbImage : Image - period : long  - startXLivesDisplay : int |
| - drawLives (g : Graphics) : void - drawScore(g : Graphics) : void + startGame ( ) : void + stopGame ( ) : void + run ( ) : void + addNotify ( ) : void + gameRender ( ) : void + printGameOver ( ) : void + paintScreen ( ) : void + resumeGame ( ) : void + pauseGame ( ) : void + getUserInitials ( ) : String + keyPressed (e : KeyEvent) : void + keyReleased( e : KeyEvent) : void + keyTyped(e : KeyEvent) : void + printGameOver ( ) : void + setUpGame ( diff : Difficulty) : void |

**Figure B.5:** GameView Class Diagram

|  |
| --- |
| **SoundEffectsLibrary** |
| - soundStorage : SoundEffectsLibrary - nameToSound : HashMap <String, AudioClip> |
| + playSound (name : String) : void |

**Figure B.6:** SoundEffectsLibrary Class Diagram

|  |
| --- |
| **SpriteLibrary** |
| - storage : SpriteLibrary - nameToSprite : HashMap <String, Sprite> |
| + getSprite (name : String) : Sprite |

**Figure B.7:** SpriteLibrary Class Diagram

|  |
| --- |
| **Sprite** |
| - spriteImage : Image |
| + getHeight ( ) : int + getWidth ( ) : int |

**Figure B.8:** Sprite Class Diagram

|  |
| --- |
| **GameModel** |
| + DEFAULT\_BULLET\_SPEED : int + DEFAULT\_NUM\_LIVES : int + NUM\_OF\_BARRIERS : int - barriers : ArrayList<Barrier> - bonus : BonusPotater - bonusTopSpacer : int - diffSetting : Difficulty - firePressed : boolean - gameOver : boolean - gamePaused : boolean - leftBoundary : int - leftPressed : boolean - lives : int - pWave : PotaterWave - projectiles : ArrayList<Projectile> - rand : Random - rightBoundary : int - rightPressed : boolean - score : int - smeagol : Gollum - topBoundary : int - windowHeight : int - windowWidth : int |
| + canShoot(shooter : Fireable) : boolean + checkProjectiles ( ) : void + collides(bullet : Projectile) : Destroyable +completedLevel( ) : boolean + destroyOvertakenBarrier (tater : OrdinaryPotater) : void + gameIsOver ( ) : boolean + gameUpdate( ) : void + getAllotted Lives(diif : Difficulty) : int + invasionSuccessful ( ) : boolean + nextLevel ( ) : void + outOfRange (proj : Projectile) : boolean + outOfrange (tater : BonusPotater) : boolean + randomDeployBonus ( ) : void + setFirePressed (firePressed : boolean) : void + setLeftPressed(leftPressed : boolean) : void + setRightPressed(rightPressed : boolean) : void + terminateGame ( ) : void + togglePause ( ) : void |

**Figure B.9:** GameModel Class Diagram

|  |
| --- |
| **Gollum** |
| - game : GameModel - speed : int - location : int [ ] - sprite : Sprite |
| + moveLeft ( ) : void + moveRight ( ) : void + resetPosition ( ) : void + shoot ( ) : Projectile + getRectRep ( ) : Rectangle + draw ( g : Graphics) : void |

**Figure B.10:** Gollum Class Diagram

|  |
| --- |
| ***<<abstract>> Potater*** |
| - speed : int - location : int [ ] - pointValue : int - sprite : Sprite |
| + moveLeft ( ) : boolean + moveRight ( ) : boolean + draw (g :Graphics) : void + getRectRep ( ) : Rectangle |

**Figure B.11:** Potater Class Diagram

|  |
| --- |
| **OrdinaryPotater** |
| - isSpriteV1 : boolean - rank : PotaterRank |
| + getPointFromRank(rank : PotaterRank ) : int + getSpriteFromRank(rank : PotaterRank , isSpriteV1 : boolean) : Sprite + moveDown ( ) : void + moveLeft ( ) : void + moveRight ( ) : void + shoot ( ) : Projectile + switchSprite ( ) : void |

**Figure B.12:** OrdinaryPotater Class Diagram

|  |
| --- |
| **BonusPotater** |
| - BONUS\_POINTS : int - BONUS\_SPEED : int - movingLeft : boolean |
| + move ( ) : void + moveLeft ( ) : void + moveRight ( ) : void |

**Figure B.13:** BonusPotater Class Diagram

|  |
| --- |
| **PotaterWave** |
| - DEFAULT\_OP\_STEP\_SIZE : int - difficultySetting : Difficulty - game : GameModel - haveWaited : int - initialLowestPoint : int - leftDirection : boolean - potaters : ArrayList<OrdinaryPotater> - randomShot : Random - reached25Percent : boolean  - reached50Percent : boolean - reached75Percent : boolean - slowWaitPeriod : int - speedModifer : int - topSpacer : int - waitPeriod : int |
| - addPotater(tater : OrdinaryPotater) : void + alternateSprites ( ) : void + atEdge ( ) : boolean + draw (g : Graphics) : void + hasInvaded ( ) : boolean + isEmpty ( ) : boolean + isTimeToMove ( ) : boolean + killPotater (casualty : OrdinaryPotater) : boolean + lowestPoint ( ) : int + move ( ) : void + potaterOnBottom (tater : Potater) : boolean + projectileHitPotater (proj : Projectile) : OrdinaryPotater + randomShootBottoms ( ) : void + yCoordWhenPercentTravaled (fractionTraveled : double) : int |

**Figure B.14:** PotaterWave Class Diagram

|  |
| --- |
| ***<<abstract>> Projectile*** |
| - speed : int - location : int [ ] - sprite : Sprite - shooter : Fireable |
| + draw ( g : Graphics) : void + getRectRep ( ) : Rectangle |

**Figure B.15:** Projectile Class Diagram

|  |
| --- |
| **PotaterProjectile** |
|  |
| + moveDown ( ) : void |

**Figure B.16:** PotaterProjectile Class Diagram

|  |
| --- |
| **GollumProjectile** |
|  |
| + moveUp ( ) : void |

**Figure B.17:** GollumProjectile Class Diagram

|  |
| --- |
| **Barrier** |
| - BLOCK\_SIDE\_LENGTH : int - blocks : ArrayList <BarrierBlock> - sprite : Sprite - location : int [ ] |
| + projectileHitBlock (proj : Projectile, block) : BarrierBlock + removeBlock (block : BarrierBlock) : void + draw (g : Graphics) : void + drawDamage(g : Graphics) : void |

**Figure B.18:** Barrier Class Diagram

|  |
| --- |
| **BarrierBlock** |
| - width : int - height : int - isActive : boolean - location : int [ ] |
| + getRectRep ( ) : Rectangle + drawDamage (g : Graphics) : void + setInactive ( ) : void |

**Figure B.19:** BarrierBlock Class Diagram

|  |
| --- |
| **<<interface>> MoveableHorizontally** |
|  |
| + moveLeft ( ) : void + moveRight ( ) : void |

**Figure B.20:** MoveableHorizontally Interface Diagram

|  |
| --- |
| **<<interface>> MoveableUp** |
|  |
| + moveUp ( ) : void |

**Figure B.21:** MoveableUp Interface Diagram

|  |
| --- |
| **<<interface>> MoveableDown** |
|  |
| + moveDown ( ) : void |

**Figure B.22:** MoveableDown Interface Diagram

|  |
| --- |
| **<<interface>> Drawable** |
|  |
| + draw (g : Graphics) : void |

**Figure B.23:** Drawable Interface Diagram

|  |
| --- |
| **<<interface>> Fireable** |
|  |
| + shoot ( ) : Projectile |

**Figure B.24:** Fireable Interface Diagram

|  |
| --- |
| **<<interface>> Destroyable** |
|  |
| + getRectRep ( ) : Rectangle |

**Figure B.25:** Destroyable Interface Diagram

|  |
| --- |
| **ChooseDifficulty** |
| - DIFFICULT\_IMAGE\_LOCATION : String - difficultyBackground : Image - easyButton : JButton - hardButton : JButton - mediumButton : JButton - returnButton : JButton - spTop : SpacePotaters |
| + paintComponent(g : Graphics) : void |

**Figure B.26:** ChooseDifficulty Class Diagram

|  |
| --- |
| **InstructionView** |
| - INSTRUCTIONS\_IMAGE\_LOCATION : String - instructionBackground : Image - returnButton : JButton - spTop : SpacePotaters |
| + paintComponent(g : Graphics) : void |

**Figure B.27:** InstructionView Class Diagram

|  |
| --- |
| **GetInitials** |
| - jButton1 : JButton - jLabel1 : JLabel - jLabel2 : JLabel - jPanel1 : JPanel - jTextField1 : JTextField - name : String |
| - enterPressed (evt : KeyEvent) : void - getEneteredInitials( ) : String - handleAction ( ) : void - initComponents ( ) : void - jButton1ActionPerformed(evt : ActionEvent) : void |

**Figure B.28:** GetInitials Class Diagram

|  |
| --- |
| **<<enumeration>> Difficulty** |
| EASY MEDIUM HARD |

**Figure B.29:** Difficulty Enumeration Diagram

|  |
| --- |
| **<<enumeration>> PotaterRank** |
| BOTTOM MIDDLE TOP |

**Figure B.30:** PotaterRank Enumeration Diagram