**Interstellar Game Systems, Inc**

**Space Wars: Gravity X (Game)**

**Spring 2016**

**Designed by**

**Guangzheng Wu**



**DEPARTMENT OF COMPUTER SCIENCE**

**GEORGIA STATE UNIVERSITY**

Atlanta, Georgia

April 21th, 2016

Table of Contents

WSD……………………………………………………………………………... 1

Gantt Chart………………………………………………………………………. 2

Problem Statement………………………………………………………………. 3

RTM……………………………………………………………………………... 6

Prototype……………………………………………………………………….... 8

Use Cases………………………………………………………………………... 9

Sequence Diagrams…………………………………………………………….. 27

Category Interaction Diagram………………………………………………….. 45

Function Point Cost Analysis…………………………………………………... 51

COCOMO…………………………………………………………………….... 52

FPCA&COCOMO Comparison……………………………………………...... 57

Object Design………………………………………………………………….. 58

Test Cases…………………………………………………………………….... 64

Dictionary……………………………………………………………………… 67

Software Architecture & Database…………………………………………….. 68

Rationale……………………………………………………………………….. 69

Project Legacy………………………………………………………………….. 72

User Guide……………………………………………………………………… 73

Resume…………………………………………………………………………. 74

**WSD** **for Space Wars: Gravity X Game Development**

Test Cases, Ship Implementer

Object in the Application,

Rationale for Object

Rationale

Object in the Application,

Rationale for object

Software Architecture

Database, Horizontal Prototype

Interstellar Game Systems, Inc

Developer Coordinator:

Eduardo López

Problem Statement,

Dictionary,

Rationale.

WSD, Gantt Chart,

Point Cost Analysis

Title, RTM, WSD, Gantt Chart,

Interaction Diagram, RTM

Rationale Test Cases,

GravityX & Planet Implementer

Title, WSD, Dictionary, Rationale

Gantt Chart, WSD, Category Interaction Diagram

Developer:

Huang Li

Client:

Guangzheng Wu

Developer:

Jackie Oung

Dictionary,

Gantt Chart

Developer:

Minh-Nhat Bui

Function Point Cost Analysis

RTM, Rational

Rationale for object

RTM, Title

Gantt Chart, WSD, COCOMO, Controller Implementer

Gantt Chart, WSD

Category Interaction Diagram, Gantt Chart

Ratonale

COCOMO,

Projectile Implementer

Rationale

**Problem Statement**

1.0 INTRODUCTION:

Space Wars : Gravity X is a horizontal scrolling space game where the player would have to go as far as possible in the game avoiding enemies and obstacles in order to get the highest score possible.

2.0 GRAPHICS/GAME INTERFACE

The game shall have four main objects interacting with each others :  
- The player  
- The player’s bullets/projectile  
- The planets  
- The enemies

The game shall be played horizontally and the player’s spaceship shall be traveling from the left to the right. The different obstacles (planets and enemies) shall appear from the right side of the screen moving the the left.

3.0 GAME INSTRUCTION

The player shall be able to move his spaceship around the screen by using the keyboard. The player’s spaceship shall explode if it gets in contact with a planet. The player’s spaceship shall also explode if it gets in contact with an enemy’s spaceship.  
The player shall be able to avoid enemies and find himself a path to the end of the level by destroying the planets and enemies. The player shall be able to destroy them by shooting bullets at them. The player shall also be able to upgrade the power, amount of bullets and rate of fire of the spaceship’s bullet by picking up upgrades appearing on the screen.

3.1 GAME MODE

The game shall start with the option of selecting either a single player mode or a multiplayer mode.

3.1.1 ONE PLAYER MODE

In one player mode, the player shall play the game by himself/herself.

3.1.2 TWO PLAYERS MODE

In two players mode, the players shall play in cooperative mode. They shall have to help each other in order to go as far as possible in the game. Since there would be more bullets on the screen, the planets and enemies shall be stronger and harder to destroy in two players mode.

3.2 CONTROLLER

In one player mode, the first player shall use W,A,S and D to move the spaceship and use H to shoot bullets.  
In two player mode, the second player shall use the arrow keys to move his/her spaceship around and press ‘2’ on the numpad to shoot bullets.

4.0 PHYSICS  
Different sizes planets shall be generated throughout the game. Each planet shall have its own pull strength and each planet shall modify the trajectory of the ship. Two or more planets shall attract each other as well and if two planets collide, they shall explode into multiple small fragments. The bullets of the spaceship shall not interact with the planets. The enemies shall have a predefined pattern to follow on the screen and they shall not be affected by the gravity of the planets nor shall they explode may they come in contact with a planet.

5.0 GAME DATA

The game shall keep track of the score of both players. The player score shall increase gradually over time. Also, every time a player destroy a planet or enemy, the player’s score shall increase.   
At the end of the game, the screen shall display the final score of the player(s) and record the score into the game database. If the player’s score is among the top 10 score of the leaderboard, the screen shall display a congratulation message displayer the rank of the player.

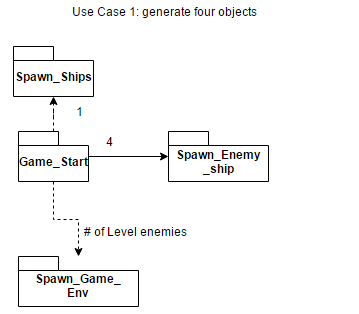
6.0 STORY

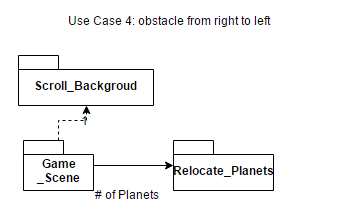
To make the game more interesting, the game shall display a little story either at the beginning or when the player loses to create an effect of surprise for the player.

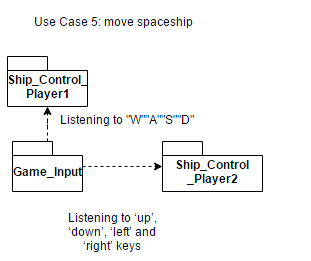
**RTM**

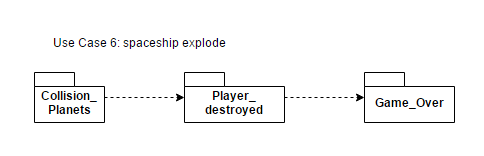
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entry# | Para# | HCC Requirement Traceability Matrix | Type | Use Case Name |
| 1 | 2.0 | The game shall have four main objects: the player, planets, enemies and the player’s bullets/projectile | SW | UC1 SW generate four objects |
| 2 | 2.0 | The game shall be played horizontally | SW | n/a |
| 3 | 2.0 | the player’s spaceship shall be traveling from the left to the right. | sw | n/a |
| 4 | 2.0 | The different obstacles (planets and enemies) shall appear from the right side of the screen moving to the left. | SW | UC4 SW obstacle from right to left |
| 5 | 3.0 | The player shall be able to move his spaceship around the screen by using the keyboard. | SW | UC5 SW move spaceship |
| 6 | 3.0 | The player’s spaceship shall explode if it gets in contact with a planet. | SW | UC6 spaceship explode |
| 7 | 3.0 | The player’s spaceship shall also explode if it gets in contact with an enemy’s spaceship. | SW | n/a |
| 8 | 3.0 | The player shall be able to avoid enemies and find himself a path to the end of the level by destroying the planets and enemies. | SW | UC8 SW destroy planets and enemies |
| 9 | 3.0 | The player shall be able to destroy them by shooting bullets at them. | SW | UC9 SW shooting bullets |
| 10 | 3.0 | The player shall also be able to upgrade the power, amount of bullets and rate of fire of the spaceship’s bullet by picking up upgrades appearing on the screen. | SW | UC10 SW spaceship upgrade |
| 11 | 3.1 | The game shall start with the option of selecting either a single player mode or a multiplayer mode. | SW | UC11 SW game mode |
| 12 | 3.1.1 | In one player mode, the player shall play the game by himself/herself | SW | UC12 SW Single player mode |
| 13 | 3.1.2 | In two players mode, the players shall play in cooperative mode. | SW | UC13 SW Double player mode |
| 14 | 3.1.2 | They shall have to help each other in order to go as far as possible in the game | SW | UC14 SW Cooperation |
| 15 | 3.1.2 | Since there would be more bullets on the screen, the planets and enemies shall be stronger and harder to destroy in two players mode. | SW | n/a |
| 16 | 3.2 | In one player mode, the first player shall use W,A,S and D keys to move the spaceship and use the spacebar key to shoot bullets. | HW | n/a |
| 17 | 3.2 | In two player mode, the second player shall use the arrow keys to move his/her spaceship around and press ‘0’ on the numpad to shoot bullets. | HW | n/a |
| 18 | 4.0 | Different sizes planets shall be generated throughout the game. | SW | n/a |
| 19 | 4.0 | Each planet shall have its own pull strength | SW | UC19 SW planet info |
| 20 | 4.0 | Each planet shall modify the trajectory of the ship. | SW | n/a |
| 21 | 4.0 | Two or more planets shall attract each other as well | SW | UC21 SW planet attraction |
| 22 | 4.0 | If two planets collide they shall explode into multiple small fragments. | SW | UC22 SW planet fragment collision |
| 23 | 4.0 | The bullets of the spaceship shall not interact with the planets. | SW | n/a |
| 24 | 4.0 | The enemies shall have a predefined pattern to follow on the screen | SW | UC24 SW enemies creation |
| 25 | 4.0 | they shall not be affected by the gravity of the planets | SW | n/a |
| 26 | 4.0 | nor shall they explode may they come in contact with a planet. | SW | n/a |
| 27 | 5.0 | The game shall keep track of the score of both players. | SW | UC27 SW game score |
| 28 | 5.0 | The player score shall increase gradually over time. | SW | n/a |
| 29 | 5.0 | Every time a player destroy a planet or enemy, the player’s score shall increase. | SW | n/a |
| 30 | 5.0 | At the end of the game, the screen shall display the final score of the player(s) and record the score into the game database. | SW | n/a |
| 31 | 5.0 | If the player’s score is among the top 10 score of the leaderboard, the screen shall display a congratulation message displayer the rank of the player. | SW | UC31 SW display and record score |
| 32 | 6.0 | The game shall display a little story either at the beginning or when the player loses | SW | UC32 SW display story |

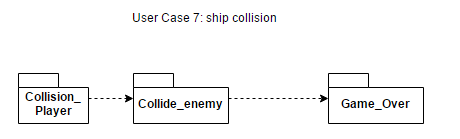
**Category Interaction Diagram**

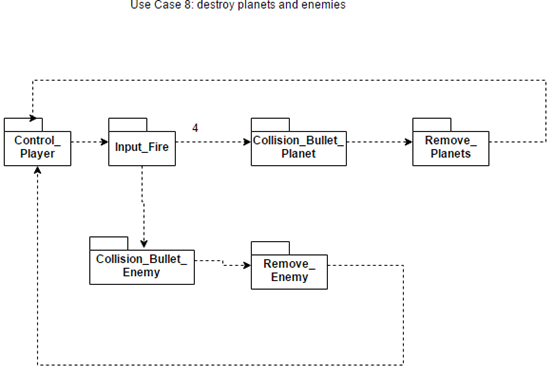


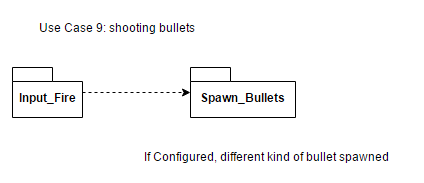


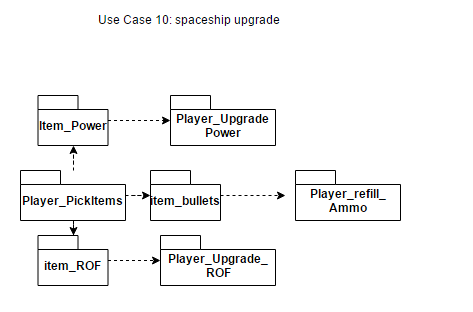


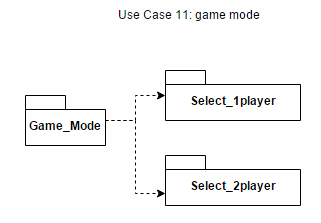


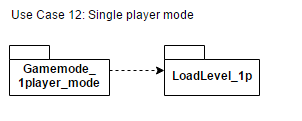


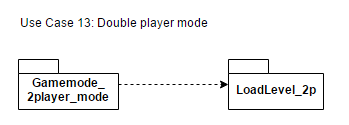


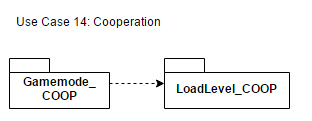


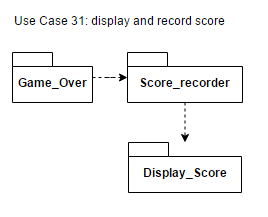
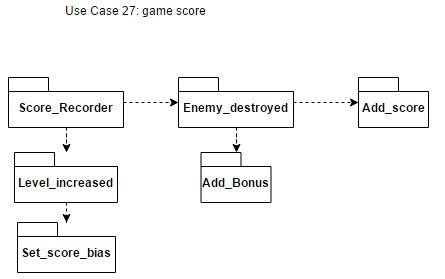
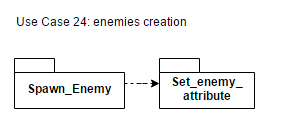
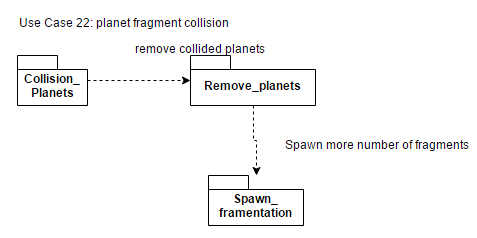
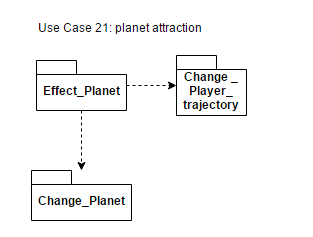
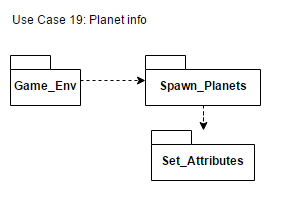












**Dictionary**

RTM, Requirements Traceability Matrix:  
Requirements tracing, a process of documenting the links between the requirements and the work products developed to implement and verify those requirements. The RTM captures all requirements and their traceability in a single document delivered at the conclusion of the life cycle.

Gantt Chart:

A Gantt chart is a type of bar chart that illustrates a [project schedule](https://en.wikipedia.org/wiki/Schedule_(project_management)). A Gantt charts illustrates the start and finish dates of the terminal elements and summary elements of a [project](https://en.wikipedia.org/wiki/Project).

**Rationale**

With the increased popularity of video games in the last few years, we decided for a our product to be a game. Several studies show that most of the money spent on Apps is on the video games category, that is the reason why we decided not to sell our product to a single client, instead we want to reach as many people as possible with our product.

We really like the concept of scrolling shooting games, but we didn’t want to make it a typical shooting game. That is why we decided to add interesting features like gravity and planets. We believe that our use of physics, especially gravitational laws, will make our game highly entertaining and differentiate it from other shooting games available. Our idea is to make enough obstacles to maintain the user immersed in the game.

The last decision we had to take was whether we wanted our game to be a single player game or a multiplayer game. After several hours of discussion, we decided that in order to achieve the most fun for our customers we should make our game a cooperative multiplayer game with a single player option. In single player mode, the player would have to try to go as far as possible by himself. However in multiplayer mode, the obstacles and enemies are going to be harder to destroy and therefore, they will have to cooperate and make sure to combine and coordinate a strategy in order to create themselves a path to the end.

**Rationale for Use Case**

After analysing the problem statement, we found 31 differents entries in the Requirement Traceability Matrix. All of those entries were very distinct from each others, however we noticed that some of them could be regrouped together while doing the Use Cases. Indeed, after a closer look at all the different entries that we had, we noticed that it was unecessary to create a specific Use Case for every single entries. Let’s look at the entries #14, #15, #16 and #17 for example. The entries #14 and #15 only give general information about how the players need to help each other in multiplayer modes and how the planets would be harder to destroy while entries #16 and #17 give more precise and specific details about the keys the users need to press in order to control their spaceship. It wouldn’t really make sense to create a Use Case just to mention that both players have to help each other. Therefore we decided to regroup those four entries into a single Use Case. Those four entries are not the single case where we used a single Use Case to regroup multiple entries. After analysing all the entries, we managed to regroup a couple of entries and ended up with 17 different Use Cases out of the 31 entries.

**Rationale for the Objects**

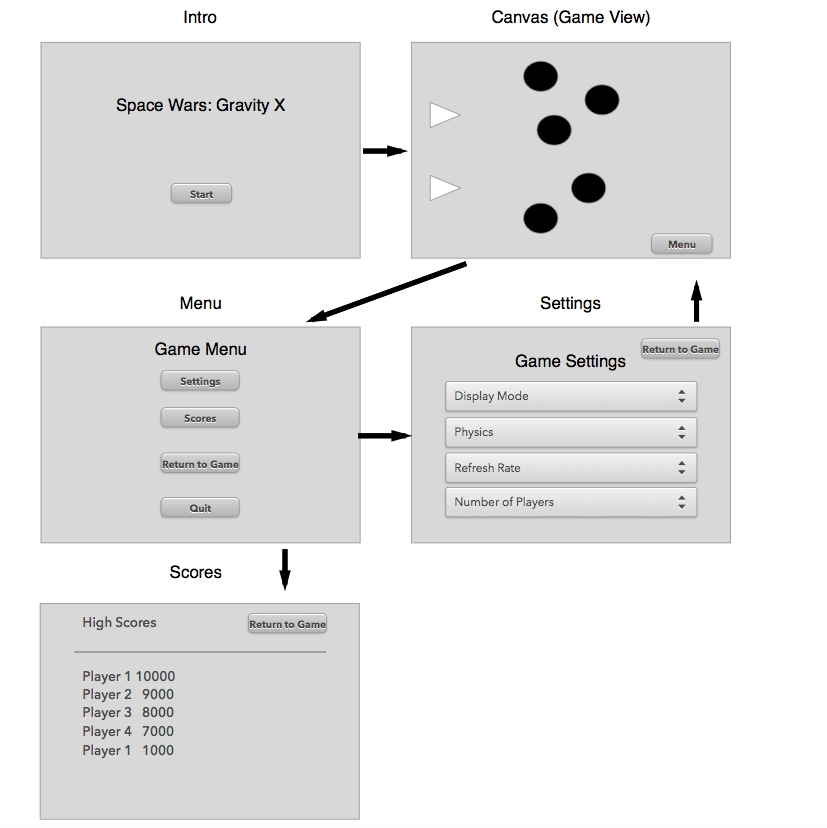
We decided to use the Model/View/Controller architecture because, as a game, our application is highly reliant on the User Interface. MVC helps separate the input logic, the game logic and the UI (rendering). The usefulness is quickly noticeable in the early stages of any game development project because it allows to change things quickly without too much rework of code in all layers of the application.   
  
 For this application the View is a Java FX canvas where we draw all the different sprites that are shown as the user interface. The Controller is composed of several classes that communicate with the model and the view. The controller includes the logic that controls what is displayed on the view. In the controller there is a Draw() class that checks the current state of all initialized sprites and draws them on the canvas.   
  
 To achieve movement and change of state for our game we use the Tick() class. The way Tick() works is by deleting everything on the canvas and calling Draw() several times per second. To allow the user influence the state of the game we use the Control() class. This class creates an event that changes the state of the PlayersShip() sprite. Control() also lets the user navigate the menu, created using the Menu() class.  
The menu lets the user view and change the state of system wide features. To view high-scores the Scores() class is used and to change the state of the game the Settings() class is used.  
  
 The last piece is the Model, which contains all the application logic. The classes included in the model cannot interact directly with the view, they change the state of the view indirectly using the controller. For this game we need to keep track of the state and position of all the sprites in the Canvas. This responsibility is assigned to the Positions() class, which also includes a very important method used to detect collisions between sprites. To create all of the different sprites we use five different classes: PlayersShip(), Enemy(), Planet(), Projectile() and PowerUp(). PlayersShip() initializes and stores the state of the sprite that represents the user. Enemy() initializes and controls the behavior of the enemy objects. Same for Planet(), Projectile() and PowerUp() with different characteristics appropriate for their functions. The PhysicsEngine() class is used for calculating the attraction between sprites. The last feature of the game is the Database using for saving restoring the games state, that is implemented by the Database() class.

**Rationale for System Analysis**

Doing the interaction diagrams helped us visualizing the different interaction and behaviour of the system. Thanks to those interaction diagrams, we were able to see how a group of objects collaborate in some behavior, especially for a single use-case. Instead of using words, those diagrams made it really clear and easy to see how objects and messages are passed between other objects within the use-case. We were able to describe the message flow in the system, capture the dynamic behaviour of it, describe its structural organization of the objects as well as describing the interaction among objects.

For example for our Use Case 21 for the planet attraction, we could better understand how the planets and spaceships would interact with each others and how the program would need to recalculate the position of the different objects and how it would need to erase the previous canvas to draw a new canvas with the new position of the different objects on screen.

**Horizontal Prototype**



**Database**

Our project will use a XML file to store data. This file will store User names, high scores, and settings.

**Software Architecture**

We decided to use the Model/View/Controller architecture, because, as a game, our application is highly reliant on the User Interface.

**Object Design**

/\*\*

\* We decided to use the Model View Controller architecture for the development of our \* application. For our application, as with any MVC application, our code is going to be \* divided into three functional categories; each category being composed of several \* classes. Below you can find a description of all of the classes organized inside the \* three categories mentioned before.

\*/

public class Game{

/\*

\* The number of times per second the draw function is called.

\*/

private int fps;

/\*\*

\* View Subsystems

\* The following class is responsible for creating the user interface.

\*/

}

public class Canvas(){...}

/\*

\* Creates a Java FX UI canvas where the User interface is rendered.

\*/

/\*

\* Controller Subsystems

\* The following classes are responsible for managing the sequence of interactions with

\* the user.

\*/

}

public static void main() {…}

/\*\*

\* Loads all the subsystems for our application and sets constant values for some \* properties of the game.

\*/

public class Tick(int fps) {…}

/\*\*

\* Runs instance of Draw several times per second to create the animation.

\*/

public class Draw() {…}

/\*

\* Draws each sprite in the canvas in the location and with the dimensions given by an \* instance of the Positions class.

\*/

public class Controls() {…}

/\*

\* Listens to I/O. When a control key is pressed creates an event.

\*/

public class Menu() {

/\*

\* When called by the class Controls instance stops the tick function and draws the menu \* on the canvas.

\*/

public int resume() {…}

/\*

\* Resumes the tick function, returns 0 if successful.

\*/

public void quit() {…}

/\*

\* Sends a save event to database and terminates all processes.

\*/

}

public class Settings() {

/\*

\* Draws the settings menu on the canvas.

\*/

Public int displayMode() {…}

/\*

\*Changes the canvas mode from window to full screen or vice versa.

\* Returns 0 if successful.

\*/

Public void Physics() {…}

/\*

\* Changes the behavior of the physics engine.

\*/

Public void adjustFrameRate() {…}

/\*

\* Sets the number of times per second the Draw function is called by the Tick \* object.

\*/

Public void mutiplayerMode() {…}

/\*

\* Sets the number of PlayersShip objects that are created.

\*/

Public void backToMenu() {…}

/\*

\* Calls the Menu class.

\*/

public class Scores() {…}

/\* Displays the High-scores from the database on the canvas.

back() {…}

/\* Public void backToMenu() {…}

Description: Calls the Menu class.

/\*

\* 3. Model Subsystems

\* The following classes are responsible for maintaining domain knowledge. All the

\* behind the scenes processing is going to be done by them.

\*/

public class Positions() {

/\* Keeps track of where in the canvas each sprite is placed and of it’s dimensions. Sets new positions based on the gravity influence and previous positions and directions.

public bool collision() {…}

/\* When two sprites intersect on the canvas creates an event. \*Returns true if there is a collisions, else returns false.

\*/

public class PlayersShip() {…}

/\*

\* Sets a players sprite and behavior.

\*/

}

public class Enemy() {

/\*

\* Sets an enemy sprite and controls its behavior.

\*/

public void intelligence() {…}

/\*

\* Adapts the behavior of the enemy object according to the state of the other \* sprites.

\*/

}

public class Planet() {…}

/\*

\* Sets the properties for a planet object.

\*/

public void collisionEvent() {…}

/\*

\* If hit by a projectile decides wether to be destroyed or split, and sets \* new properties.

\*/

}

public class Projectile() {…}

/\*

\*Triggered by controls or Enemy objects AI. Creates a sprite with projectile properties.

\*/

public class PowerUp() {…}

/\*

\*Creates a sprite. Stores a property value for a PlayersShip object.

\*/

public class Physics() {…}

/\*

\* Calculates the gravity forces for all sprites in the Canvas.

\*/

public class Database() {

/\*

\* Stores and loads game properties from an XML file.

\*/

public int load() {…}

/\*

\* Sets all global variables to the ones saved on an XML file.

\* Returns 0 if successful.

\*/

public int save() {…}

/\*

\* Saves all global variables to an XML file.

\* Returns 0 if successful.

\*/

}