Software Architecture Document

AMBroSIA

Version 1.0

Nikolaos Bukas, Anthony Chin, Meong Hye Seo, Michael Smith, Haisin Yip

Ambrosia Games Ltd.
March 1, 2013

1.0 System Overview

1.1 Purpose

The purpose of the document is to present a detailed description of all classes, objects and associated methods used in the game. The document is intended for current developers working on the project, future developers and the client. It is to be used as a point of reference when writing the game itself.

It should be noted that this document is subject to change, and may be modified by any intended party at any time if circumstances require it (i.e. it is discovered that intended functionality is impossible to implement as described in this document).

1.2 Scope

The piece of software to be written will be a game called “AMBroSIA”, modeled after the arcade-style game “Asteroids”. The game can be played by one or two people, and will feature a 2D graphical user interface in which the player will be able to control a spaceship and destroy approaching asteroids. As a game, the software will have an entertainment value to those who play it, with the development team’s goal being to maximize the entertainment value possible.

This design document presents an overview of how the game will be structured, without delving deeply into implementation details. It follows from the Software Requirements Specification

2.0 Views

2.1 UML Diagram

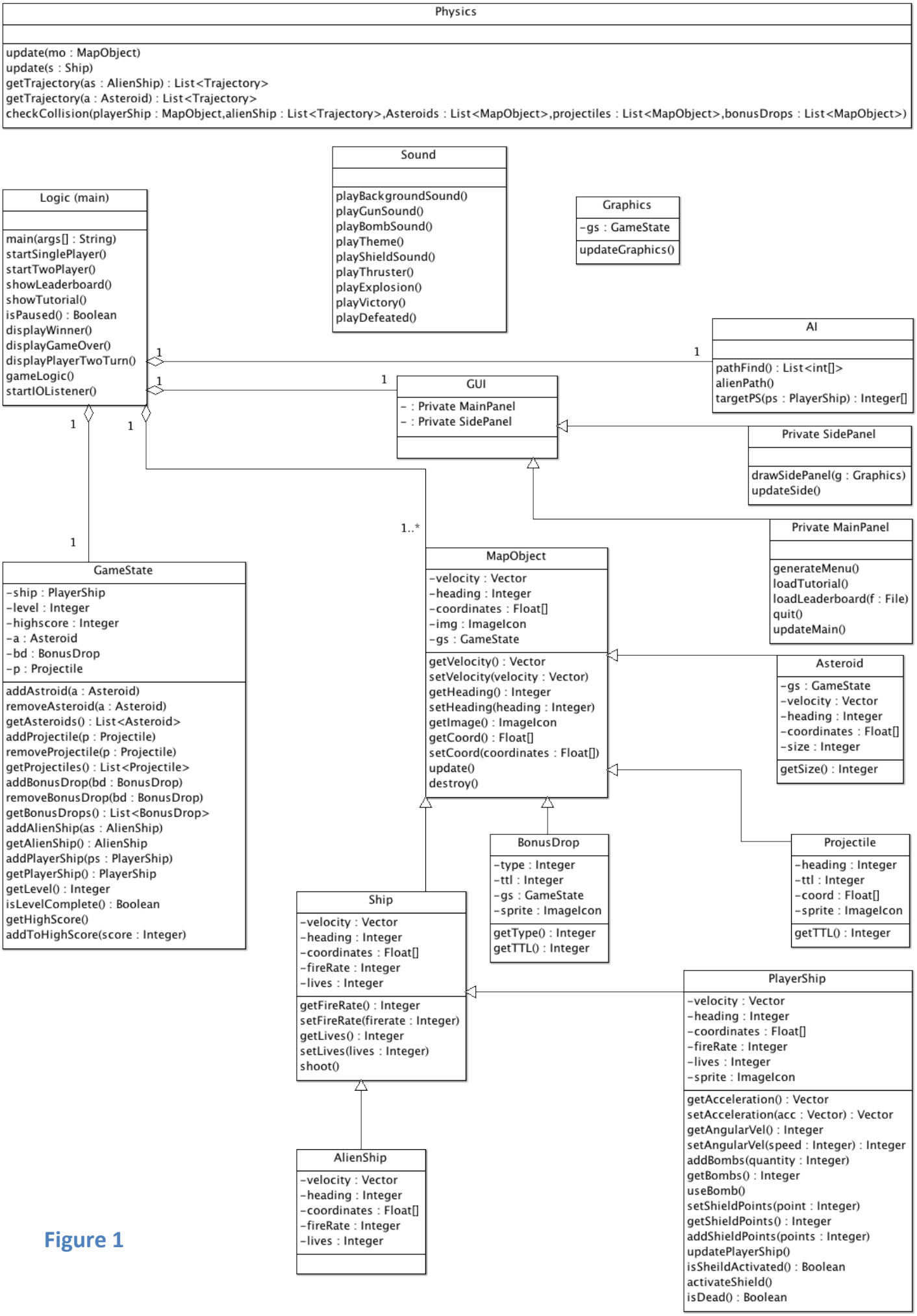
Physics
update(mo : MapObject)
update(s : Ship)
getTrajectory(as : AlienShip) : List<Trajectory>
getTrajectory(a : Asteroid) : List<Trajectory>
checkCollision(playerShip : MapObject,alienShip : List<Trajectory>,Asteroids : List<MapObject>,projectiles : List<MapObject>,bonusDrops : List<MapObject>)
Sound
playBackgroundSound()
playGunSound()
playBombSound()
playTheme()
playShieldSound()
playThruster()
playExplosion()
playVictory()
playDefeated()
Logic (main)
main(args[] : String)
startSinglePlayer()
startTwoPlayer()
showLeaderboard()
showTutorial()
isPaused() : Boolean
displayWinner()
displayGameOver()
displayPlayerTwoTurn()
gameLogic()
startIOListener()
Graphics
-gs : GameState
updateGraphics()
AI
pathFind() : List<int[]>
alienPath()
targetPS(ps : PlayerShip) : Integer[]
GUI
- : Private MainPanel
- : Private SidePanel
Private SidePanel
drawSidePanel(g : Graphics)
updateSide()
Private MainPanel
generateMenu()
loadTutorial()
loadLeaderboard(f : File)
quit()
updateMain()
GameState
-ship : PlayerShip
-level : Integer
-highscore : Integer
-a : Asteroid
-bd : BonusDrop
-p : Projectile
addAstroid(a : Asteroid)
removeAsteroid(a : Asteroid)
getAsteroids() : List<Asteroid>
addProjectile(p : Projectile)
removeProjectile(p : Projectile)
getProjectiles() : List<Projectile>
addBonusDrop(bd : BonusDrop)
removeBonusDrop(bd : BonusDrop)
getBonusDrops() : List<BonusDrop>
addAlienShip(as : AlienShip)
getAlienShip() : AlienShip
addPlayerShip(ps : PlayerShip)
getPlayerShip() : PlayerShip
getLevel() : Integer
isLevelComplete() : Boolean
getHighScore()
addToHighScore(score : Integer)
MapObject
-velocity : Vector
-heading : Integer
-coordinates : Float[]
-img : ImageIcon
-gs : GameState
getVelocity() : Vector
setVelocity(velocity : Vector)
getHeading() : Integer
setHeading(heading : Integer)
getImage() : ImageIcon
getCoord() : Float[]
setCoord(coordinates : Float[])
update()
destroy()
Asteroid
-gs : GameState
-velocity : Vector
-heading : Integer
-coordinates : Float[]
-size : Integer
getSize() : Integer
BonusDrop
-type : Integer
-ttl : Integer
-gs : GameState
-sprite : ImageIcon
getType() : Integer
getTTL() : Integer
Projectile
-heading : Integer
-ttl : Integer
-coord : Float[]
-sprite : ImageIcon
getTTL() : Integer
Ship
-velocity : Vector
-heading : Integer
-coordinates : Float[]
-fireRate : Integer
-lives : Integer
getFireRate() : Integer
setFireRate(firerate : Integer)
getLives() : Integer
setLives(lives : Integer)
shoot()
PlayerShip
-velocity : Vector
-heading : Integer
-coordinates : Float[]
-fireRate : Integer
-lives : Integer
-sprite : ImageIcon
getAcceleration() : Vector
setAcceleration(acc : Vector) : Vector
getAngularVel() : Integer
setAngularVel(speed : Integer) : Integer
addBombs(quantity : Integer)
getBombs() : Integer
useBomb()
setShieldPoints(point : Integer)
getShieldPoints() : Integer
addShieldPoints(points : Integer)
updatePlayerShip()
isSheildActivated() : Boolean
activateShield()
isDead() : Boolean
AlienShip
-velocity : Vector
-heading : Integer
-coordinates : Float[]
-fireRate : Integer
-lives : Integer
1
1..*

Figure 1

3.0 Software Units

3.1 Classes

3.1.1 Logic (main class)

3.1.1.1 Purpose

The purpose of the logic class is to manage all other classes and methods in such a way as to produce a playable game. This includes (but is not limited to) calling for the creation of the main menu, displaying the leaderboard or starting the game in either single or two player mode as appropriate, depending on player actions.

3.1.1.2 Interface

void main(String args[])
Main method; creates the main menu and acts as appropriate depending on player input.

void startSinglePlayer()
This method will start the single player game.

void startTwoPlayer()
This method will start the game in 2 player mode.

void showLeaderboard()
This method will display the leaderboard to the user. The leaderboard file (*.txt) will be retrieved when method is called to display stored score information.

void showTutorial()
This method will show the tutorial information to the user.

boolean isPaused()
This method checks if the game is paused

void displayWinner()
The output displays text stating that the player has won (as appropriate for single or two player).

void displayGameOver()
The output display a “Game Over” message.

void displayPlayerTwoTurn()
The method will display information relevant to player two’s turn.

void gameLogic()
The method is a wrapper for numerous subclasses and private methods, implementing core game functionality - checking for, responding to and creating events as appropriate.

void startIOListener()
This method enables the recognition of keyboard and mouse input.

3.1.2. GUI

3.1.2.1 Purpose

The purpose of the GUI class is to provide the frame of the game. This frame will output the main menu screen, tutorial screen, and the gameplay.

3.1.2.2 Interface

GUI()
This is a constructor that initiates a JFrame. Uses MainPanel and SidePanel.

3.1.2.1 MainPanel

3.1.2.1.1 Purpose

The purpose of the MainPanel class is to provide the main gameplay area of the game.

3.1.2.1.2 Interface

MainPanel()
This constructor initiates the main panel of the main game.

void generateMenu()
This method will display the main menu user interface.

void loadTutorial()
This method will display the tutorial interface.

void loadLeaderboard(File f)
This method will display the leaderboard interface. It takes a file containing stored results.

void quit()
This method will close the MainPanel.

void updateMain()
This method will update the main panel.

3.1.2.2 SidePanel

3.1.2.2.1. Purpose

The purpose of the SidePanel class is to provide the player with important information such as the high score, lives, bombs remaining and shield points remaining.

3.1.2.2.2 Interface

sidePanel()
This constructor initiates a side frame with information as stated above.

drawSidePanel(Graphics g)
This method shall draw out the fonts for the text to be displayed.

void updateSide()
This method shall update the frame at a certain rate.

3.1.3 GameState

3.1.3.1 Purpose

The purpose of the GameState class is to keep the state of the game in memory. This is done by holding references to all the Asteroid, Projectile, Bonus Drop, and Ship objects currently active in the game.

3.1.3.2 Interface

GameState(PlayerShip ship, int level, int highscore)
This is the constructor for the GameState class. It takes an object of type PlayerShip and an integer value representing the level as arguments. It also creates empty lists for the following object types: Asteroid, BonusDrop, Projectile.

void addAsteroid(Asteroid a)
Will add the object a into the list of asteroids.

void removeAsteroid(Asteroid a)
Will remove the object a from the list of asteroids.

List<Asteroid> getAsteroids()
Returns a list of Asteroids.

void addProjectile(Projectile p)
Will add the object p into the list of projectiles.

void removeProjectile(Projectile p)
Will remove the object p from the list of projectiles.

List<Projectile> getProjectiles()
Returns a list of Projectiles.

void addBonusDrop(BonusDrop bd)
Will add the object bd to the list of bonus drops.

void removeBonusDrop(BonusDrop bd)
Will remove the object bd from the list of bonus drops.

List<BonusDrop> getBonusDrops()
Returns a list of Bonus Drops.

void addAlienShip(AlienShip as)
Will make the AlienShip variable reference object as.

AlienShip getAlienShip()
Returns the AlienShip object in game state.

void addPlayerShip(PlayerShip ps)
Will make the PlayerShip variable reference object ps.

PlayerShip getPlayerShip()
Returns the PlayerShip object in game state.

int getLevel()
Returns the level.

boolean isLevelComplete()
Return true if level is completed.

int getHighScore()
Returns the high score.

void addToHighScore(int score)
Adds to high score the value of score.

3.1.5 MapObject

3.1.5.1 Purpose

The purpose of the MapObject is to provide a general representation of game objects (such as asteroids and aliens). It includes functionality present in all objects (to some extent).

3.1.5.2 Interface

MapObject(Vector velocity, int heading, float[] coordinates, ImageIcon img, GameState gs)
The constructor, which takes all essential values: a velocity (magnitude and direction), heading and coordinates, along with an image representing the object in question and the GameState, used to add the object to the game when created or remove it when destroyed.

Vector getVelocity()
Returns a vector object representing the velocity of the object.

void setVelocity(Vector velocity)
Sets the velocity to the given vector.

int getHeading()
Returns the heading in which the object is facing (not to be confused with the direction related to velocity).

void setHeading(int heading)
Sets the heading to the specified value.

ImageIcon getImage()
Returns the image representing the object.

float[] getCoord()
Returns an array of length 2 containing the coordinates of the object.

void setCoord(float[] coordinates)
Sets the coordinates to those given in the *coordinates* array.

void update()
Updates the position of the object.

void destroy()
The destroy method is used to removed the MapObject from the game state. Necessary animations, sound effects or other effects of the destruction are also taken care of (destroy() overridden in inheriting classes when appropriate).

3.1.5.1 BonusDrop

3.1.5.1.1 Purpose

The purpose of the BonusDrop is to provide a general representation of the all the bonus drop objects. It inherits the MapObject class. In addition, it contains integer properties for the type and Time to Live (TTL).

3.1.5.1.2 Interface

BonusDrop(int type, int ttl)
The constructor of BonusDrop. The type indicates the type of bonus drop it is and the ttl indicates the time the bonus drop has until it disappears. The BonusDrop is added to the game when the object is created.

int getType()
Returns the integer value of the type property (represents whether its a life or a bomb).

int getTTL()
Returns the int value of the TTL property.

3.1.5.2 Projectile

3.1.5.2.1 Purpose

The purpose of the Projectile is to provide the representation for the projectile objects. It inherits properties and methods from the MapObject class.

3.1.5.2.2 Interface

Projectile(int ttl)
The constructor takes the TTL, and adds itself to the game.

int getTTL()
Returns the integer value of the TTL property.

3.1.5.3 Asteroid

3.1.5.3.1 Purpose

The purpose of the asteroid class is to represent asteroids in the game. Most parameters and methods are inherited from MapObject, with one exception.

3.1.5.3.2 Interfaces

Asteroid(int size)
Constructor takes the size of the asteroid. It adds the Asteroid object to the gamestate.

int getSize()
Returns the size of the asteroid.

3.1.5.4 Ship

3.1.5.4.1 Purpose

The ship object is used as a generic representation of a ship, either player or alien. It inherits from MapObject.

3.1.5.4.2 Interfaces

ship(int FireRate, int lives)
The constructor initializes the number of lives and FireRate. It also adds the object to the GameState.

int getFireRate()
Returns the firing rate.

void setFireRate(int firerate)
Sets the firing rate.

int getLives()
Returns the number of lives.

void setLives(int lives)
Sets the number of lives.

void shoot()
Creates a projectile.

3.1.5 PlayerShip

3.1.5.1 Purpose

The purpose of PlayerShip is to control the player ship that navigates throughout the field. It inherits from Ship.

3.1.5.2 Interface

PlayerShip()
The constructor initializes the object much like MapObject, with the addition of the firing rate and the number of lives. It adds itself to the GameState.

Vector getAcceleration()
Returns the vector Acceleration.

Vector setAcceleration(Vector acc)
Sets the vector Acceleration to acc.

int getAngularVel()
Returns the angular velocity.

int setAngularVel(int speed)
Sets the angular velocity to speed.

void addBombs(int quantity)
Adds quantity to bombs.

int getBombs()
Returns the amount of bombs.

void useBomb()
Detonates (uses) the bomb.

void setShieldPoints(int points)
Sets the amount of shield points.

int getShieldPoints()
Returns the amount of shield points.

void addShieldPoints(int points)
Adds points (can be negative) to the shield points.

boolean isShieldActivated()
This method checks if shield if activated

void activateShield()
This method activates the shield for the ship.

boolean isDead()
This method checks if the player or the ship is dead or not.

3.1.5 AlienShip

3.1.5.1 Purpose

The purpose of the AlienShip class is to represent the alienship AI in the game.

3.1.5.2 Interfaces

AlienShip()
The constructor initializes the ship, and adds it to the GameState.

3.1.6 Physics

3.1.6.1 Purpose

The purpose of the physics class is to provide the game physics, dealing with calculations involving collision and trajectories.

3.1.6.2 Interfaces

void update(MapObject mo)
Updates the fields of the given MapObject.

void update(Ship s)
Updates the fields of the given Ship.

List<Trajectory> getTrajectory(AlienShip as)
Gets available trajectories for alienship.

void checkCollision(MapObject PlayerShip, List <MapObject> AlienShip, List<MapObject> Asteroids, List<MapObject> Projectiles, List<MapObject> BonusDrops)

This method takes as input all objects on the map, and checks for collisions between them, If there are collisions, the objects are dealt with as appropriate - for example, destruction of an asteroid or the loss of life of the player ship.

3.1.7 Sound

3.1.7.1 Purpose

This purpose of the Sound class is to output the sound needed for the game.

3.1.7.2 Interfaces

void playBackgroundSound()
This method will play the background music.

void playGunSound()
This method will play the sound of the gun shooting.

void playBombSound()
This method will play the sound of the bomb exploding.

void playTheme()
This method will play the main theme

void playShieldSound()
This method will play the shield sound.

void playThruster()
This method will play the sound of the thruster firing.

void playExplosion()
This method will play the sound of an explosion.

void playVictory()
This method will play the victory sound.

void playDefeated()
This method will play the Game Over sound.

3.1.8 Graphics

3.1.8.1 Purpose

The purpose of the graphics class is to update and draw a single frame at a time of the game. More specifically, it updates and draws each component of the game such as the player's ship, the asteroids or the player information grid.

3.1.8.1 Interfaces

Graphics(GameState gs)
The constructor takes the GameState as input, using it to determine what to draw.

void updateGraphics()
Updates the current frame components by calling various draw methods and checking the GameState.

3.1.9 AI

3.1.9.1 Purpose

The AI class serves a controller for the Alien ship. In other words, it will control how the Alien ship moves and reacts.

3.1.9.2 Interfaces

List<int[]> pathfind()
Finds and returns the a set of coordinates representing the path it should follow.

void alienPath()
The alien will traverse a certain path, before disappearing.

int[] targetPS(PlayerShip ps)
Returns a set of coordinates of where the player ship might be next and shoot the projectile at that point.

3.2 Categories

AI	Physics	Sound	Graphics	GUI	Game Objects	Logic
AI	Physics	Sound	Graphics	GUI MainPanel SidePanel	MapObject BonusDrop Projectile Asteroid Ship PlayerShip AlienShip	Logic (main class) GameState

4.0 Analysis

The following traceability matrix demonstrates the coherency of the SAD and SRS.

4.1 Traceability Matrix

Requirement	AI	Physics	Sound	Graphics	GUI	Game Objects	Logic
R3.1.1.3.1	-	-	-	-	generateMenu()	-	startSinglePlayer()
R3.1.1.3.2	-	-	-	-	loadLeaderBoard()	-	showLeaderboard()
R3.1.1.3.3	-	-	-	-	startTwoPlayer()	-	start2Player()
R3.1.1.3.4	-	-	-	-	loadTutorial()	-	showTutorial()
R3.1.1.3.5	-	-	-	-	quit()	-	-
R3.1.2.3.1	-	-	-	-	drawSidePanel()	-	-
R3.1.2.3.2	-	-	-	updateGraphics()	-	-	gameLogic()
R3.1.2.3.3	-	-	-	-	-	-	gameLogic()
R3.1.2.3.4	-	checkCollision()	-	-	-	destroy()	-
R3.1.2.3.5	-	-	-	-	displayGameOver()	-	-
R3.1.2.3.6	-	-	-	-	displayWinn	-	-

					er()		
R3.1.2.3.7	-	-	-	-	updateSide( )	-	gameLogi c()
R3.1.2.3.8	-	-	-	-	-	-	gameLogi c()
R3.1.3.3.1	-	-	-	-	loadLeader Board()	-	showLead erboard()
R3.1.3.3.2	-	-	-	-	-	-	showLead erboard()
R3.1.3.3.3	-	-	-	-	loadLeader Board()	-	-
R3.1.3.3.4	-	-	-	-	loadLeader Board()	-	-
R3.1.3.3.5	-	-	-	-	generateMa inMenu()	-	-
R3.1.4.3.1	-	-	-	-	drawSidePa nel()	-	-
R3.1.4.3.2	-	-	-	updateGrap hics()	-	-	startTwoPl ayer()
R3.1.4.3.3	-	-	-	-	drawSidePa nel()	-	startTwoPl ayer()
R3.1.4.3.4	-	-	-	-	UpdateSide ()	-	gameLogi c()
R3.1.4.3.5	-	-	-	-	displayPlay erTwoTurn( )	-	-
R3.1.4.3.6	-	-	-	-	displayPlay erTwoTurn( )	-	-
R3.1.4.3.7	-	-	-	-	displayPlay erTwoTurn( )	-	-
R3.1.4.3.8	-	-	-	-	displayWinn er()	-	-
R3.1.4.3.9	-	-	-	-	UpdateSide ()	-	gameLogi c()
R3.1.5.3.1	-	-	-	-	loadTutorial ()	-	showTutor ial()

R3.1.5.3.2	-	-	-	-	generateMainMenu()	-	-
R3.1.6.3.1	-	update()	-	-	-	-	startIOListener()
R3.1.6.3.2	-	-	-	-	-	useBomb()	startIOListener()
R3.1.6.3.3	-	update()	-	-	-	-	startIOListener()
R3.1.6.3.4	-	update()	-	-	-	-	startIOListener()
R3.1.6.3.5	-	update()	-	-	-	-	startIOListener()
R3.1.6.3.6	-	-	-	-	-	shoot()	startIOListener()
R3.1.6.3.7	-	-	-	-	-	activateShield()	startIOListener()
R3.1.7.3.1	-	update()	-	-	-	-	startIOListener()
R3.1.7.3.2	-	-	-	updateGraphics()	-	-	-
R3.1.7.3.3	-	-	-	-	-	-	gameLogic()
R3.1.8.3.1	-	update()	-	-	-	setLives()	gameLogic()
R3.1.8.3.2	-	CheckForCollision()	-	updateGraphics()	-	-	-
R3.1.8.3.3	-	-	-	-	-	-	-
R3.1.8.3.4	-	-	-	-	-	-	-
R3.1.8.3.5	-	-		-	-	-	-
R3.1.8.3.6	-	update()	-	-	-	-	-
R3.1.9.3.1	-	-	-	-	-	shoot()	-
R3.1.9.3.2	-	-	-	-	-	setFireRate()	-
R3.1.9.3.3	-	-		-	-	setFireRate()	-

R3.1.9.3.4	-	-	-	-	-	-	startIOListener()
R3.1.9.3.5	-	-	-	-	-	useBomb()	-
R3.1.9.3.6	-	-	-	-	-	addBomb()	-
R3.1.9.3.7	-	-	-	-	-	useBomb()	-
R3.1.9.3.8	-	-	-	-	-	useBomb()	-
R3.1.9.3.9	-	update()	-	-	-	-	-
R3.1.9.3.10	-	-	-	-	-	-	-
R3.1.9.3.11	-	-	-	-	-	Projectiles()	-
R3.1.9.3.12	-	-	-	-	-	shoot()	-
R3.1.9.3.13	-	checkCollision()	-	-	-	-	-
R3.1.9.3.14	-	update()	-	-	-	setVelocity()	-
R3.1.10.3.1	-	-	-	-	-	setShieldPoints()	-
R3.1.10.3.2	-	checkCollision()	-	-	-	-	gameLogic()
R3.1.10.3.3	-	-	-	updateGraphics()	-	activateShield()	-
R3.1.10.3.4	-	-	-	updatesGraphics()	-	activateShield()	-
R3.1.10.3.5	-	-	-	-	-	setShieldPoints()	-
R3.1.10.3.6	-	-	-	-	-	-	gameLogic()
R3.1.11.3.1	-	checkCollsion()	-	-	updateSide()	-	gameLogic()
R3.1.11.3.2	-	checkCollision()	-	-	updateSide()	-	gameLogic()
R3.1.11.3.3	-	checkCollision()	-	-	updateSide()	-	gameLogic()
R3.1.11.3.4	-	checkCollision()	-	-	updateSide()	-	gameLogic()
R3.1.11.3.5	-	checkCollisi	-	-	-	-	gameLogi

		on()					c()
R3.1.12.3.1	-	-	-	updateGraphics()	-	-	-
R3.1.12.3.2	-	checkCollision()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.3	-	checkCollision()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.4	-	checkCollision()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.5	-	checkCollision()	-	-	-	-	gameLogic()
R3.1.12.3.6	-	checkCollision()	-	updateGraphics()	-	-	gameLogic()
R3.1.12.3.7	-	-	-	updateGraphics	-	-	gameLogic()
R3.1.13.3.1	-	checkCollision()	-	-	-	-	gameLogic()
R3.1.13.3.2	-	-	-	-	-	-	gameLogic()
R3.1.13.3.3	targetPS()	-	-	-	-	-	-
R3.1.13.3.4	targetPS()	-	-	-	-	-	-
R3.1.13.3.5	-	-	-	updateGraphics()	-	-	-
R3.1.13.3.6	alienPath()	update()	-	-	-	-	-
R3.1.13.3.7	alienPath()	-	-	-	-	-	-
R3.1.13.3.8	-	-	-	updateGraphics()	-	-	gameLogic()
R3.1.14.3.1	-	checkCollision()	-	updateGraphics()	-	destroy()	-
R3.1.14.3.2	-	checkCollision()	-	updateGraphics()	-	-	-
R3.1.14.3.3	-	checkCollision()	-	-	-	-	gameLogic()
R3.1.14.3.4	-	checkCollision()	-	-	updateSidePanel()	-	-
R3.1.14.3.5	-	-	-	-	-	-	gameLogi

							c()
R3.1.14.3.6	-	checkCollision()	-	updateGraphics	-	-	gameLogic()
R3.1.15.3.1	-	-	-	-	-	-	gameLogic()
R3.1.15.3.2	-	-	-	-	-	-	gameLogic()
R3.1.16.3.1	-	-	-	updateGraphics()	-	-	-
R3.1.16.3.2	-	-	-	-	-	-	-
R3.1.16.3.3	-	-	-	updateGraphics()	-	isDead()	-
R3.1.16.3.4	-	-	-	updateGraphics()	-	destroy()	-
R3.1.16.3.5	-	-	-	updateGraphics()	-	getAcceleration()	-
R3.1.16.3.6	-	-	-	updateGraphics()	-	-	gameLogic()
R3.1.17.3.1	-	-	playExplosion()	-	-	destroy()	-
R3.1.17.3.2	-	-	playExplosion()	-	-	isDead()	-
R3.1.17.3.3	-	-	playExplosion()	-	-	destroy()	-
R3.1.17.3.4	-	-	playGunSound()	-	-	shoot()	-
R3.1.17.3.5	-	checkCollision()	playShieldSound()	-	-	-	-
R3.1.17.3.6	-	-	playThruster()	-	-	setAcceleration()	-
R3.1.17.3.7	-	-	playTheme()	-	-	-	-
R3.1.17.3.8	-	-	playTheme()	-	-	isDead()	-
R3.1.17.3.9	-	-	playVictory()	-	-	-	-

4.2 Quality Requirements

[R3.2.1.2] The biggest usage of memory will be in loading all the graphical and audio assets of the game into RAM. We are assuming the average size of a sprite is 1 MB each, 1 MB each for the various sound effects, and 4 MB for the background theme song. In total there will be 10 (3x asteroids, 1x player ship, 1x alien ship, 1x projectile, 3x bonus drops, 1x explosion) sprites, 5 sound effects (1x explosion, 1x shooting, 1x acceleration, 1x victory, 1x game over), and 1 background sound. Therefore the total memory usage will be 24 MB (10 MB graphics, 9 MB sound, 5 MB code).

[R3.2.1.3] Assuming the same average sizes as above, and assuming a total size of 2 MB for code, the game will use a maximum of 21 MB of hard drive storage.

[R3.2.1.4] The game will be programmed in Java and therefore is compatible with most modern operating systems that have a Java Virtual Machine.

The follow requirements are currently not applicable. They will be satisfied during the testing phase:

R3.2.1.1
R3.2.1.4
R3.2.1.5
R3.2.2.1

The software architecture was not designed with maintainability in mind. It is specifically specified in the System Requirements Specification that maintainability is not a requirement. For reuse and flexibility please see our design rationale that uses example cases to show that our architecture was designed taking reuse and flexibility into account.

4.3 Design Rationale

A custom architecture was chosen for this project, as common design layouts were not suited for a project of AMBroSIA's scale. For a small project with only a few group members and limited time to meet, a modular approach was needed. As such, the following model was chosen:

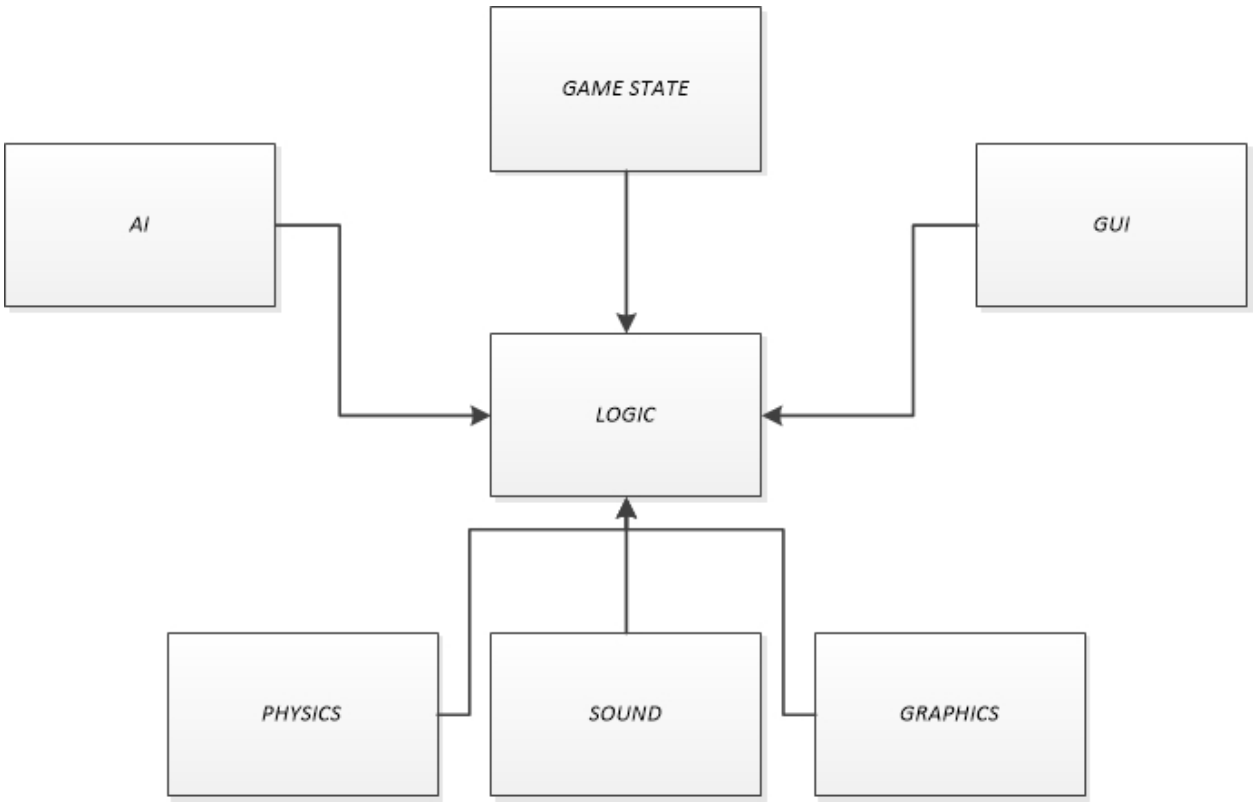

Figure 2

This model allows individual classes to be developed separately. with placeholders for not yet completed classes or methods. For example, objects present on the map - such as the player ship or the asteroids - can use classes such as physics, sound or graphics, without knowing the implementation details involving, for example, trajectory calculations. This also allows code performing similar functionality to be contained within the appropriate class, rather than being split up among many different classes. This reduces interdependencies, improving flexibility.

A good example involves the calculation of the next position of an object for the next frame. Physics implements the code for calculating the next position, allowing any object to call update() and allowing the physics class to handle all the code as appropriate - for example, asteroids move at constant velocity, but the player ship has acceleration and momentum. This requires a different calculation, but having both implemented in the physics class reduces the need of code duplication.

With regards to other classes, a MapObject was used (with inheritance) as it allows functionality common to all objects to be grouped together. It also ensures that code that needs to use properties common to all objects (but nothing more) can access anything from the player's ship to an asteroid without using different classes. This simplifies code and improves its readability, as well as promoting code reuse.

In terms of visuals, the Graphics class was created to draw the sprites and animations of each individual game object, while the GUI class was developed to handle the drawing of the actual interface (ie. the game panels). While they are related, they do perform drastically different functions, making it best to keep them separate. This keeps the code as flexible as possible.

When it comes to storing game data, a GameState class was created whose sole purpose is to keep track of all objects and data in the game. This allows a central access point from which any piece of code can view and change game information, simplifying function calls and making it easy to keep track of what objects exist. It also ensures changes in data storage do not have an adverse affect on other classes - an example of code flexibility.

Finally, the AI class is used to control the AlienShip. It provides paths for the AlienShip and coordinates for which the AlienShip should direct its weapon fire to (the player ship). Given that the algorithms involved would likely be quite different from the AlienShip objects and other in-game objects, the decision was made to keep it separate to allow for code flexibility.