Memorial University of Newfoundland

Faculty of Engineering and Applied Science

ENGR 5895

Final Design Document

Leanne Brockerville
200636215
Jermaine Francoeur
200638104

Date: Mar 28, 2011

1. Overview:

Project – Side scrolling space shooter game "Bovinursus"

We chose to this project because we felt that the amount of work in terms of length and difficulty for a space shooter game was appropriate. We figured a game that has easier mechanics will give us more time to pay attention to the details and make it really polished.

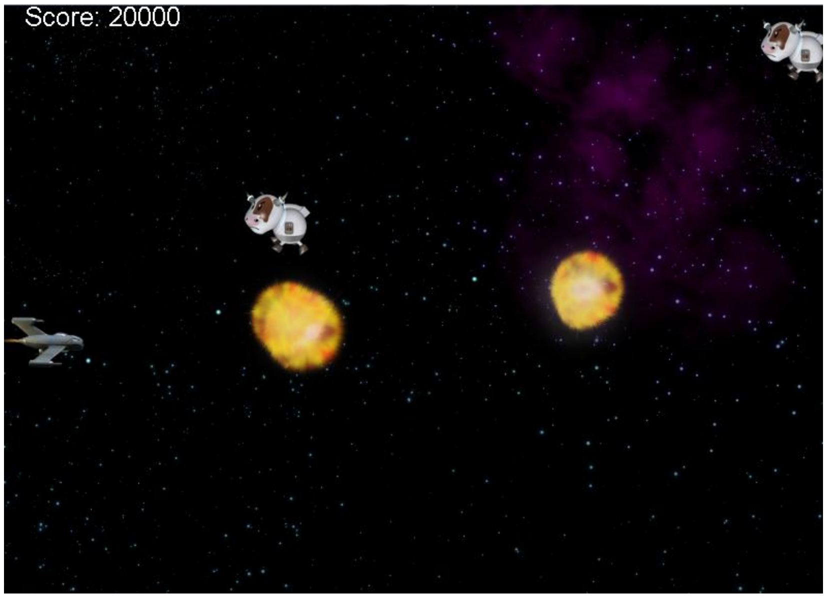


Figure 0 - Screenshot of Our Game

Game Mechanics:

In our game, the player will control a space ship travelling through outer space. At the beginning of the game, the player will be able to choose the difficulty level (easy/hard). During gameplay, the viewable screen will move slowly to the right with enemies appearing from the left. The player will be able to shoot down the enemies. The player will be able to pick up powerups that enhance their ship's abilities. At the end of each level there will be a boss.

Overview of Features for the Final Design:

Many new features are planned for the final iteration of this project. More details on these features can be found in section 2 – Use Cases.

- Multiple difficulty levels
- Multiple playable levels
- Animated Background
- Player ship has health (shield power)
- Player has multiple lives
- Powerups (extra life, health, weapon upgrade)
- Enemies can shoot projectiles at the player
- Enemies have different motion paths and images
- Landscape in the levels
- Boss at the end of each level

Table of Contents

2.1 Use Cases – First Iteration:

The following is a recap of the use cases that were implemented in the first iteration of our design. Some of the initial use cases have been modified for the final version. The modifications are shown in RED.

Actors:

Note the difference between the 'User' and the 'Player'

User: The human user of the application
Player: The user's presence in the game; the space ship
Projectile: A projectile fired by the player or an enemy
Enemy: Other entities that act as antagonists trying to destroy the player
Powerup: Entities that grant upgrades or extra powers to the player
Boss: An enemy that the player must fight at the end of a level.

Case #01: Start Game (First Iteration)
Actors: User
Triggered by: The application's initial launch by the user
Consequences:

- An artistic image following the game's theme is displayed with the title superimposed
- There is a several second delay
- The play area is shown and music starts
- The user can begin controlling the ship

Notes: If a menu screen with options is included in a later version it would be shown during this stage.

Case #01: Start Game (Final Iteration)
Actors: User
Triggered by: The application's initial launch by the user
Consequences:

- An artistic image following the game's theme is displayed with the title superimposed
- A selection menu is superimposed over the image. The selections are "hard" or "easy" difficulty.
- The user can use the direction keys (w,a,s,d) to move the cursor to the desired difficulty level and press the 'j' key to select that difficulty.
- After the difficulty is selected the play area of the first level is shown and music starts. The player can begin controlling the ship.
- The play area will have an animated background

Note: See use case #9 (final iteration) for more information on difficulty levels

Case #02: User Moves Ship
Actors: User, Player
Triggered by: The user presses one of the designated move keys on the keyboard.
Consequences: The ship's position changes
Notes:
- The player can move diagonally based on an appropriate combination of keys (ie, pressing up and right causes the player to move northeast)
- Pressing two keys in opposite directions simultaneously results in no motion in that axis (ie, pressing up and down causes no vertical displacement)
- The player's speed of motion is determined by a combination of the keys pressed and simulated momentum and drag forces.

Case #03: Player shoots a projectile
Actors: User, Player, Projectile
Triggered by: The user presses the designated shooting key on the keyboard
Consequences:
- A bullet is created
- A 'shooting' sound effect is played
- The amount and angle of the projectiles fired varies based on the player's gun power level (see use case #15 for info on powerups)

Notes: If the user hold the shooting key multiple projectiles will be shot, one at a time with a small delay between each.

Case #04: Enemy is created
Actors: Enemy
Triggered by: The player's progress through the level. The positions of the enemies are pre-determined in the structure of the level. As the player moves forward through the level, the enemies are created when the player reaches the point where they are defined to emerge.
Consequences:
- An enemy is created ahead of the player just outside the visible area

Notes: Enemies will vary in the way they move. For example, enemies can move sinusoidally, linearly (varying directions), or in a path that tracks the player. Enemies will vary in their appearance.

Case #05: Enemy gets hit by projectile
Actors: Enemy, Player Projectile
Triggered by: Any player projectile being within striking range of an enemy
Consequences:
- The player projectile is destroyed
- The enemy is destroyed
- A 'ship explosion' sound effect is played
- A 'ship explosion' effect is shown on screen where the enemy used to be
- The game score increases

Notes: If more than one player projectile would collide simultaneously with a single enemy only one projectile is destroyed, others continue on their path

Case #06: Enemy goes out of bounds
Actors: Enemy
Triggered by: An enemy being more than a specified distance outside the viewable area
Consequences: The enemy is destroyed
Notes: No extra points are earned

Case #07: Player collides with an enemy ship
Actors: Player, Enemy
Triggered by: The player is in striking range of any enemy ship
Consequences:

- The enemy is destroyed
- The player loses a predetermined amount of shield power
- A 'ship explosion' sound effect is played
- A 'ship explosion' effect is shown on screen for the enemy

Notes: No points are earned

Case #08: User pauses the game
Actors: User
Triggered by: The user presses the designated pausing key on the keyboard
Consequences:

- All gameplay ceases; player motion, enemy motion, projectile motion, music etc.
- The word 'Paused' is superimposed over the middle of the screen
- A 'Pause' sound effect is played

Notes: Gameplay resumes when the user presses the pause key again

Case #09: "Game over...The user wins" (First Iteration)
Actors: User, Player
Triggered by: The player successfully traverses the length of the level
Consequences:

- The user no longer has control of the ship
- The words 'You win!' are superimposed on the screen
- Gameplay resumes at the next level or the game ends

Notes: At this point the user will be taken back to the main screen
THIS IS REPLACED IN THE FINAL ITERATION BY CASE #10 AND #11

2.2 Use Cases – Final Iteration

The following are the use cases that will be implemented in the final iteration of the game:

Case #09: User chooses difficulty level
Actors: User, Enemy
Triggered by: The User selects a difficulty level at the title screen (See use case #1)
Consequences:
- All enemies in the game will move at a specified speed if the easy difficulty is selected. All enemies will move at a faster speed if the hard difficulty is selected.
- Enemies that shoot projectiles at the player will shoot more projectiles if the hard difficulty is selected and fewer projectiles if the easy difficulty is selected.

Case #10: Player reaches the end of a level
Actors: Player
Triggered by: The Player survives until the end of a level. If there is a boss at the end of the level, the boss is defeated by the player.
Consequences:
- The message "Level Complete" will be displayed on the screen, superimposed over the gameplay screen.
- After a short delay, the next level will begin.

Notes: This case covers only the completion of a level that is not the final level in the game.

Case #11: Player reaches the end of the game
Actors: Player
Triggered by: The Player survives until the end of the final level in the game. If there is a boss at the end of the level, the boss is defeated by the player.
Consequences:
- The message "Level Complete" will be displayed on the screen, superimposed over the gameplay screen.
- After a short delay, an endgame screen will be displayed.
- The endgame screen will consist of credits and a background image and music playing.
- After a short delay, the game will reset (back to the title screen.)

Case #12: Player loses a life
Actors: Player, Enemy or Projectile
Triggered by: The player's shield power reaches zero. (The player's shield power is reduced whenever the player collides with an enemy or enemy projectile.)
Consequences:
- The player is destroyed
- A 'ship explosion' sound effect is played
- A 'ship explosion' effect is shown on screen for the player
- Enemy and projectile motion stops.
- The player will lose one life.
- After a short delay, gameplay will resume at the beginning of the current level.

Case #13: Player runs out of lives
Actors: Player
Triggered by: The player's shield power reaches zero while the player has no remaining lives.
Consequences:

- The player is destroyed
- A 'ship explosion' sound effect is played
- A 'ship explosion' effect is shown on screen for the player
- Enemy and projectile motion stops and a "Game Over" message will be displayed over the gameplay screen.
- After a short delay, the game will reset (back to the title screen.)

Case #14: Powerup is created
Actors: Powerup
Triggered by: The player's progress through the level. The positions of the powerups are pre-determined in the structure of the level. As the player moves forward through the level, the powerups are created when the player reaches the point where they are defined to emerge.
Consequences:

- A powerup is created ahead of the player just outside the visible area.
- The powerup moves slowly to the left through the playable area.

Notes: If the powerup reaches the right side of the screen, it will be destroyed.

Case #15: Player Collides with Powerup
Actors: Player
Triggered by: A powerup being within striking range of the player.
Consequences:

- The powerup will disappear.
- A "powerup" sound effect will be played.
- The ship will gain some kind of extra power, i.e. extra life, more shield power, better gun, etc.
- The appearance of the powerup will indicate the specific power to be gained.

Case #16: Enemy shoots a projectile
Actors: Enemy, Projectile, Player
Triggered by: The enemy is ready to shoot a projectile
Consequences:

- An enemy bullet is created
- The enemy bullet will follow a straight line path. The path will be in the direction of the player. (As the player moves, the projectile will not track the player but continue moving in a straight line.)

Notes: The enemy will be able to shoot projectiles at a predetermined rate. Whether the enemy is ready to shoot a projectile will depend on this rate and when the last projectile was shot.

2.3 Additional Use Cases

We have some ideas for some additional use cases. These may or may not be implemented, depending on time constraints.

<u>**Case #17:** Player enters enclosed area</u>
Actors: Player
Triggered by: The player's progress through the level. The landscape is pre-determined by the structure of the level. As the player moves forward through the level, landscape appears on the top and bottom of the viewable area when the player reaches the point where it is defined to appear.
Consequences:

- Landscape is created just outside the viewable area and scrolls slowly into view from the right.

<u>**Case #18:** Player collides with landscape</u>
Actors: Player
Triggered by: The player is within striking distance of the level's landscape
Consequences:

- The player is destroyed
- A 'ship explosion' sound effect is played
- A 'ship explosion' effect is shown on screen for the player
- Enemy and projectile motion stops.
- The player will lose one life.
- After a short delay, gameplay will resume at the beginning of the current level.

Notes: Note the difference between this case and when the player is hit by enemies. When the player is hit by enemies or enemy projectiles, it loses some shield power, however colliding with the landscape results in the immediate loss of a life.

<u>**Case #19:** Land based enemies shoot at the player</u>
Actors: Player, Enemy
Triggered by: The player is passing through an area with landscape
Consequences:

- A land-based enemy (turret) is created ahead of the player just outside the visible area
- The turret slowly scrolls to the left along with the landscape
- The turret shoots projectiles at the player
- If the turret reaches the right side of the screen, it will be destroyed and no points are earned by the player.

Notes: Similar to the flying enemies, the turret can be destroyed by the player's projectiles (use case 5 – enemy hit by projectile) and can shoot at the player (use case 16 – enemy shoots projectile.) Refer to these use cases for more details.

Case #20: Player fights boss
Actors: Player
Triggered by: The player completes the initial stage of a level and reaches the boss stage.
Consequences:

- An enemy with special abilities and defenses (boss) will appear and attempt to destroy the player
- Unlike other enemies, the boss will remain in the play area
- The boss will have its own shield power level
- The boss stage of the game will continue until either the player or the boss is destroyed.

3. Description of Design

The following pages contain class diagrams for our design. Figure1 shows the high-level structure and figures 2-4 show each package in more detail. We have chosen to use the model-view-controller design pattern for our overall structure. Thus our classes are divided into three separate packages:

Figure 1: Shows our design at a high level of abstraction. It is mainly the same as the high level class diagram from the initial design document. One difference to note is that we have removed the observer between the model and the controller. This was originally intended to allow the model to notify the controller when sounds needed to be played. We have since replaced the observer with a much simpler ArrayList of booleans.

Controller: The controller is responsible for handling user inputs, playing sounds and managing the calls to update the view and the model. The main class will be in the controller package, which will instantiate the controller, view, and model.

Model: The model will handle all of the internal game logic. It will keep track of the game state (ie. Title, Game, Lose, Win, etc.) and responds to the keyboard commands sent to it from the controller. It takes care of the creation of levels, enemies, powerups, etc. The main difference from the initial design class diagram is that the SpriteDataHandler and SpriteData classes are removed.

View: The view will be responsible for displaying the game to the screen. It will show different screen formats based on the current game state. The class diagram for the view has changed significantly from the initial design. This is the result of moving animation from the model package into the view package (See section 6 for more info on moving animation into the view.)

Communication between Classes:
The way the packages communicate with each other is shown in **Figure 1**. The Controller communicates to the model through the *IControllerModel* interface (Note we are using the façade design pattern here) and to the view through the *IControllerView* interface. The view can also communicate to the model through the *IViewModel* interface.

Figure 1

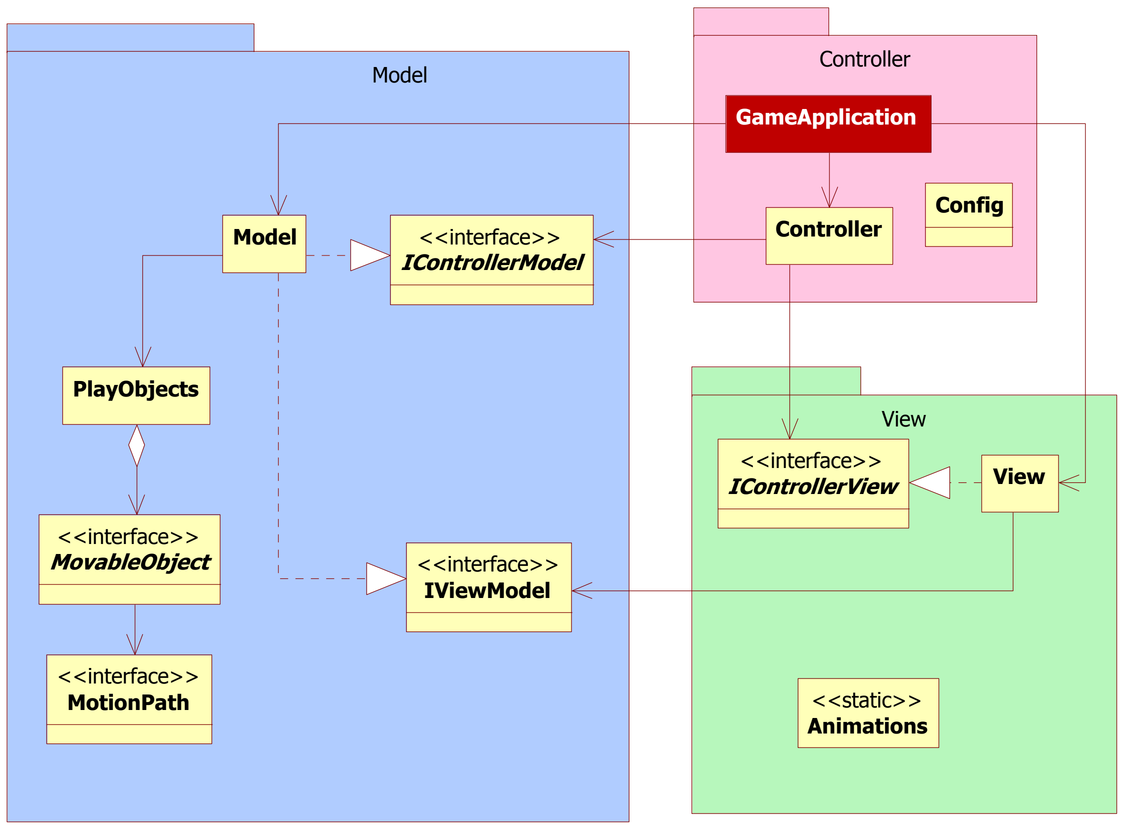

Model
Controller
GameApplication
Model
<<interface>>
IControllerModel
Controller
Config
PlayObjects
View
<<interface>>
IControllerView
View
<<interface>>
MovableObject
<<interface>>
IViewModel
<<interface>>
MotionPath
<<static>>
Animations

Figure 2

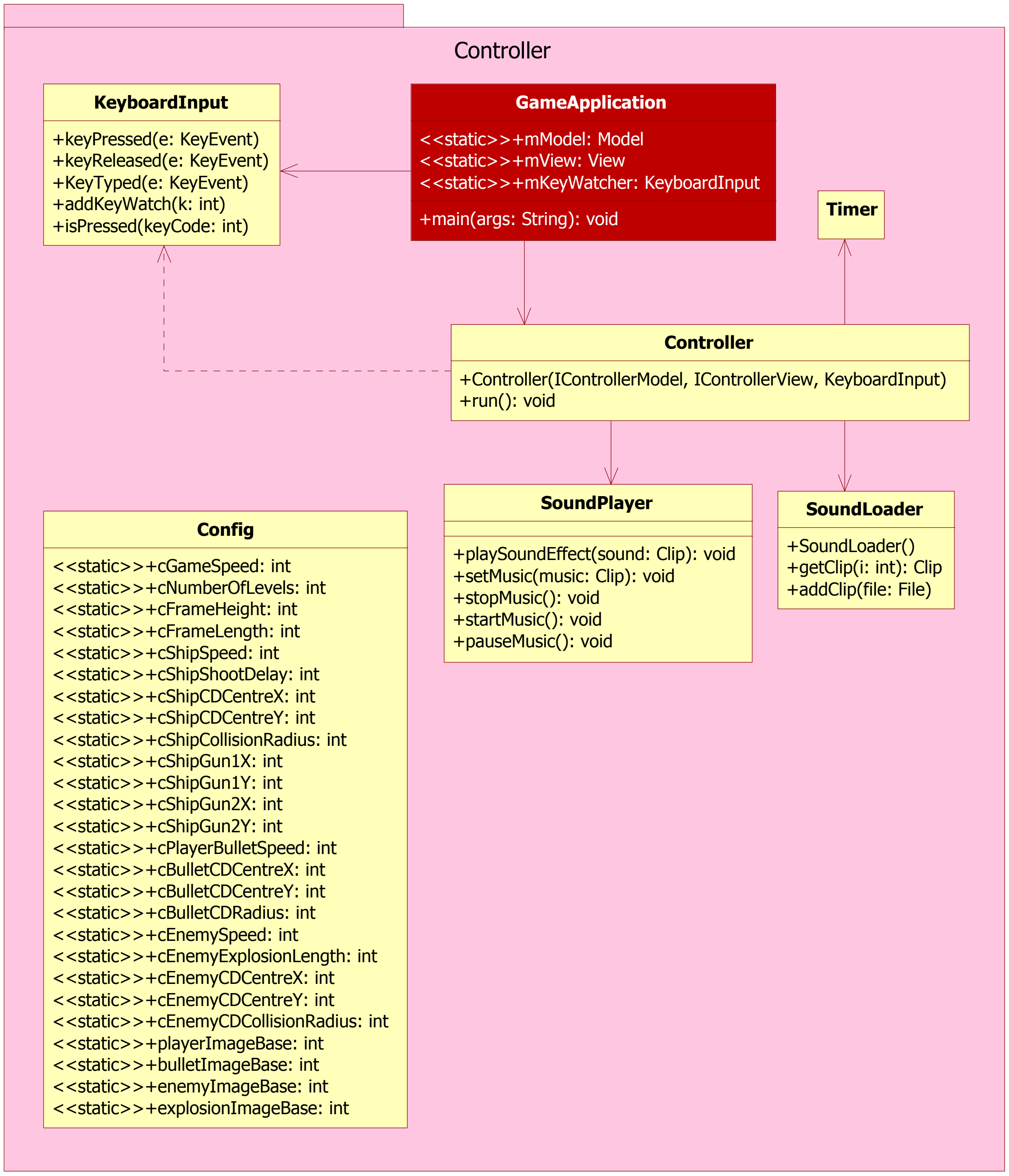

Controller
KeyboardInput
+keyPressed(e: KeyEvent)
+keyReleased(e: KeyEvent)
+KeyTyped(e: KeyEvent)
+addKeyWatch(k: int)
+isPressed(keyCode: int)
GameApplication
<<static>>+mModel: Model
<<static>>+mView: View
<<static>>+mKeyWatcher: KeyboardInput
+main(args: String): void
Timer
Controller
+Controller(IControllerModel, IControllerView, KeyboardInput)
+run(): void
SoundPlayer
+playSoundEffect(sound: Clip): void
+setMusic(music: Clip): void
+stopMusic(): void
+startMusic(): void
+pauseMusic(): void
SoundLoader
+SoundLoader()
+getClip(i: int): Clip
+addClip(file: File)
Config
<<static>>+cGameSpeed: int
<<static>>+cNumberOfLevels: int
<<static>>+cFrameHeight: int
<<static>>+cFrameLength: int
<<static>>+cShipSpeed: int
<<static>>+cShipShootDelay: int
<<static>>+cShipCDCentreX: int
<<static>>+cShipCDCentreY: int
<<static>>+cShipCollisionRadius: int
<<static>>+cShipGun1X: int
<<static>>+cShipGun1Y: int
<<static>>+cShipGun2X: int
<<static>>+cShipGun2Y: int
<<static>>+cPlayerBulletSpeed: int
<<static>>+cBulletCDCentreX: int
<<static>>+cBulletCDCentreY: int
<<static>>+cBulletCDRadius: int
<<static>>+cEnemySpeed: int
<<static>>+cEnemyExplosionLength: int
<<static>>+cEnemyCDCentreX: int
<<static>>+cEnemyCDCentreY: int
<<static>>+cEnemyCDCollisionRadius: int
<<static>>+playerImageBase: int
<<static>>+bulletImageBase: int
<<static>>+enemyImageBase: int
<<static>>+explosionImageBase: int

Figure 3

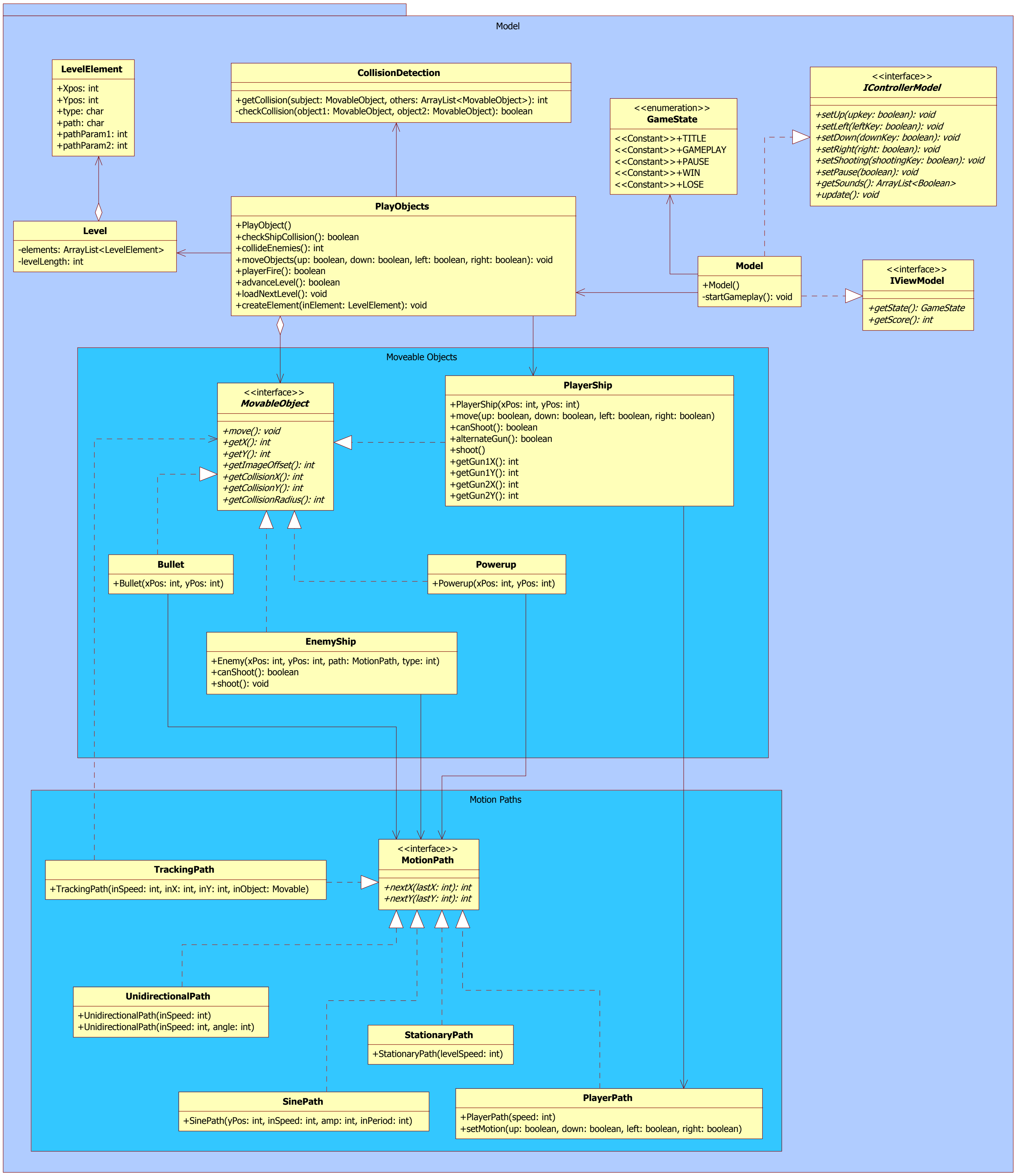
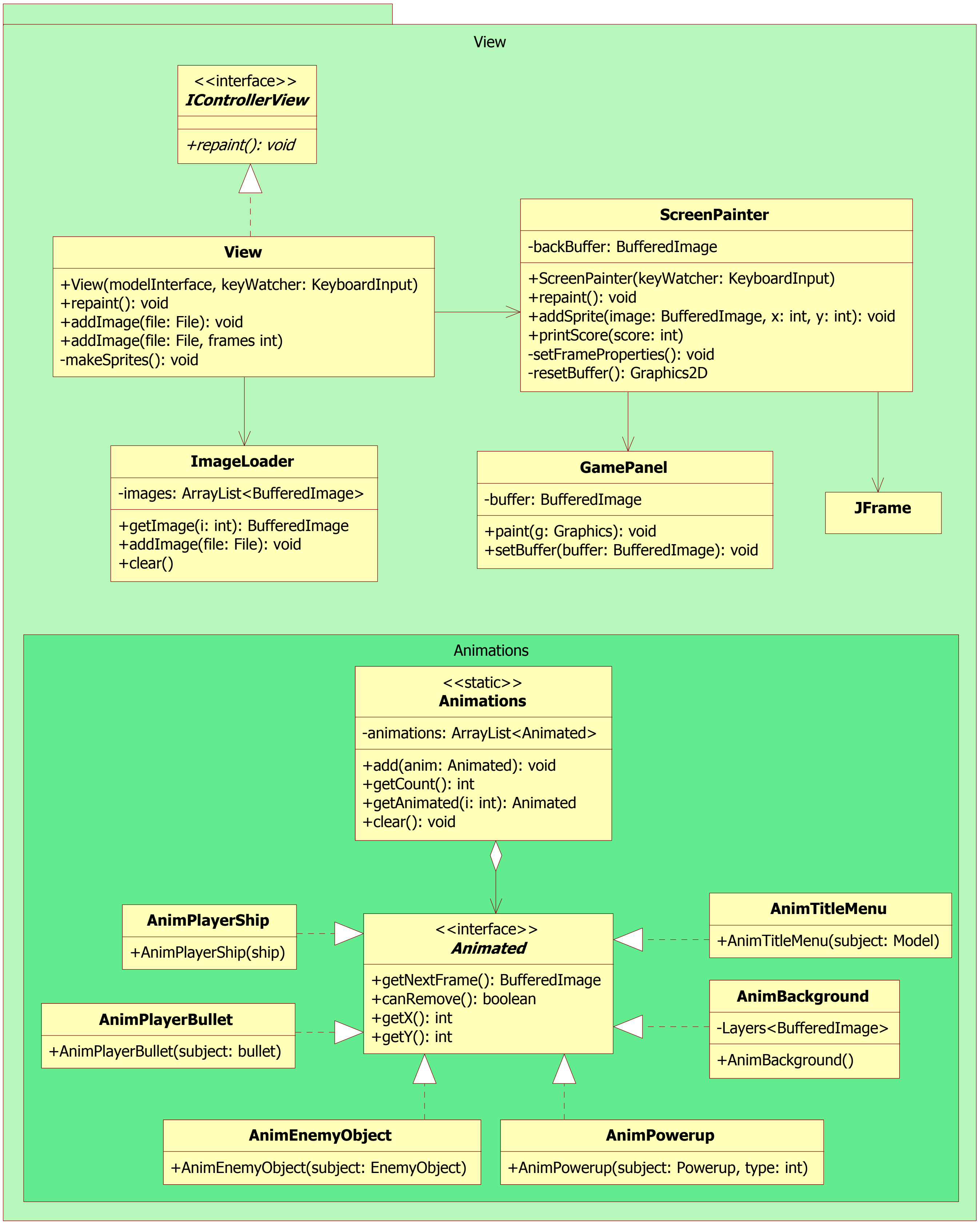

Model
LevelElement
+Xpos: int
+Ypos: int
+type: char
+path: char
+pathParam1: int
+pathParam2: int
Level
-elements: ArrayList<LevelElement>
-levelLength: int
CollisionDetection
+getCollision(subject: MovableObject, others: ArrayList<MovableObject>): int
-checkCollision(object1: MovableObject, object2: MovableObject): boolean
PlayObjects
+PlayObject()
+checkShipCollision(): boolean
+collideEnemies(): int
+moveObjects(up: boolean, down: boolean, left: boolean, right: boolean): void
+playerFire(): boolean
+advanceLevel(): boolean
+loadNextLevel(): void
+createElement(inElement: LevelElement): void
<<enumeration>>
GameState
<<Constant>>+TITLE
<<Constant>>+GAMEPLAY
<<Constant>>+PAUSE
<<Constant>>+WIN
<<Constant>>+LOSE
<<interface>>
IControllerModel
+setUp(upkey: boolean): void
+setLeft(leftKey: boolean): void
+setDown(downKey: boolean): void
+setRight(right: boolean): void
+setShooting(shootingKey: boolean): void
+setPause(boolean): void
+getSounds(): ArrayList<Boolean>
+update(): void
Model
+Model()
-startGameplay(): void
<<interface>>
IViewModel
+getState(): GameState
+getScore(): int
Moveable Objects
<<interface>>
MovableObject
+move(): void
+getX(): int
+getY(): int
+getImageOffset(): int
+getCollisionX(): int
+getCollisionY(): int
+getCollisionRadius(): int
PlayerShip
+PlayerShip(xPos: int, yPos: int)
+move(up: boolean, down: boolean, left: boolean, right: boolean)
+canShoot(): boolean
+alternateGun(): boolean
+shoot()
+getGun1X(): int
+getGun1Y(): int
+getGun2X(): int
+getGun2Y(): int
Bullet
+Bullet(xPos: int, yPos: int)
Powerup
+Powerup(xPos: int, yPos: int)
EnemyShip
+Enemy(xPos: int, yPos: int, path: MotionPath, type: int)
+canShoot(): boolean
+shoot(): void
Motion Paths
<<interface>>
MotionPath
+nextX(lastX: int): int
+nextY(lastY: int): int
TrackingPath
+TrackingPath(inSpeed: int, inX: int, inY: int, inObject: Movable)
UnidirectionalPath
+UnidirectionalPath(inSpeed: int)
+UnidirectionalPath(inSpeed: int, angle: int)
StationaryPath
+StationaryPath(levelSpeed: int)
SinePath
+SinePath(yPos: int, inSpeed: int, amp: int, inPeriod: int)
PlayerPath
+PlayerPath(speed: int)
+setMotion(up: boolean, down: boolean, left: boolean, right: boolean)

Figure 4


View
<<interface>>
IControllerView
+repaint(): void
View
+View(modelInterface, keyWatcher: KeyboardInput)
+repaint(): void
+addImage(file: File): void
+addImage(file: File, frames int)
-makeSprites(): void
ScreenPainter
-backBuffer: BufferedImage
+ScreenPainter(keyWatcher: KeyboardInput)
+repaint(): void
+addSprite(image: BufferedImage, x: int, y: int): void
+printScore(score: int)
-setFrameProperties(): void
-resetBuffer(): Graphics2D
ImageLoader
-images: ArrayList<BufferedImage>
+getImage(i: int): BufferedImage
+addImage(file: File): void
+clear()
GamePanel
-buffer: BufferedImage
+paint(g: Graphics): void
+setBuffer(buffer: BufferedImage): void
JFrame
Animations
<<static>>
Animations
-animations: ArrayList<Animated>
+add(anim: Animated): void
+getCount(): int
+getAnimated(i: int): Animated
+clear(): void
AnimPlayerShip
+AnimPlayerShip(ship)
<<interface>>
Animated
+getNextFrame(): BufferedImage
+canRemove(): boolean
+getX(): int
+getY(): int
AnimTitleMenu
+AnimTitleMenu(subject: Model)
AnimPlayerBullet
+AnimPlayerBullet(subject: bullet)
AnimBackground
-Layers<BufferedImage>
+AnimBackground()
AnimEnemyObject
+AnimEnemyObject(subject: EnemyObject)
AnimPowerup
+AnimPowerup(subject: Powerup, type: int)

4. Sequence Descriptions

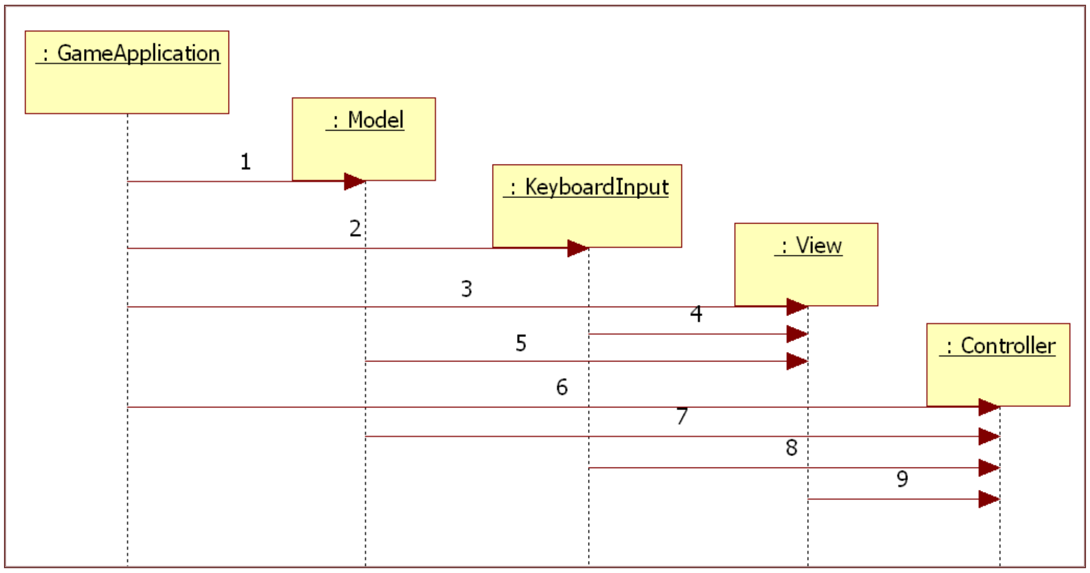
4.1 Building the Model-View-Controller Pattern

The following code snippet shows our game's main function, which is located in the GameApplication Class. It first builds the model-view-controller structure, then calls the controller's run method.

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

// create the model, view and controller classes
mModel = new model.Model();

mKeyWatcher = new KeyboardInput();

mView = new view.View( (IViewModel)mModel, mKeyWatcher);

mController = new Controller( (IControllerModel)mModel,
      (view.IControllerView)mView, mKeyWatcher);

mController.run();
}
```

The main method first instantiates the Model and KeyboardInput. Then it instantiates the View, passing it the KeyBoardInput and an Interface to the Model. Lastly it instantiates the Controller, passing it an interface to the Model, an Interface to the View and the KeboardInput.

Figure 5 – MVC Sequence Diagram



4.2 The Run Loop:

The method that runs the game is the 'run' method in the Controller Class. The run method does the following things in this order:

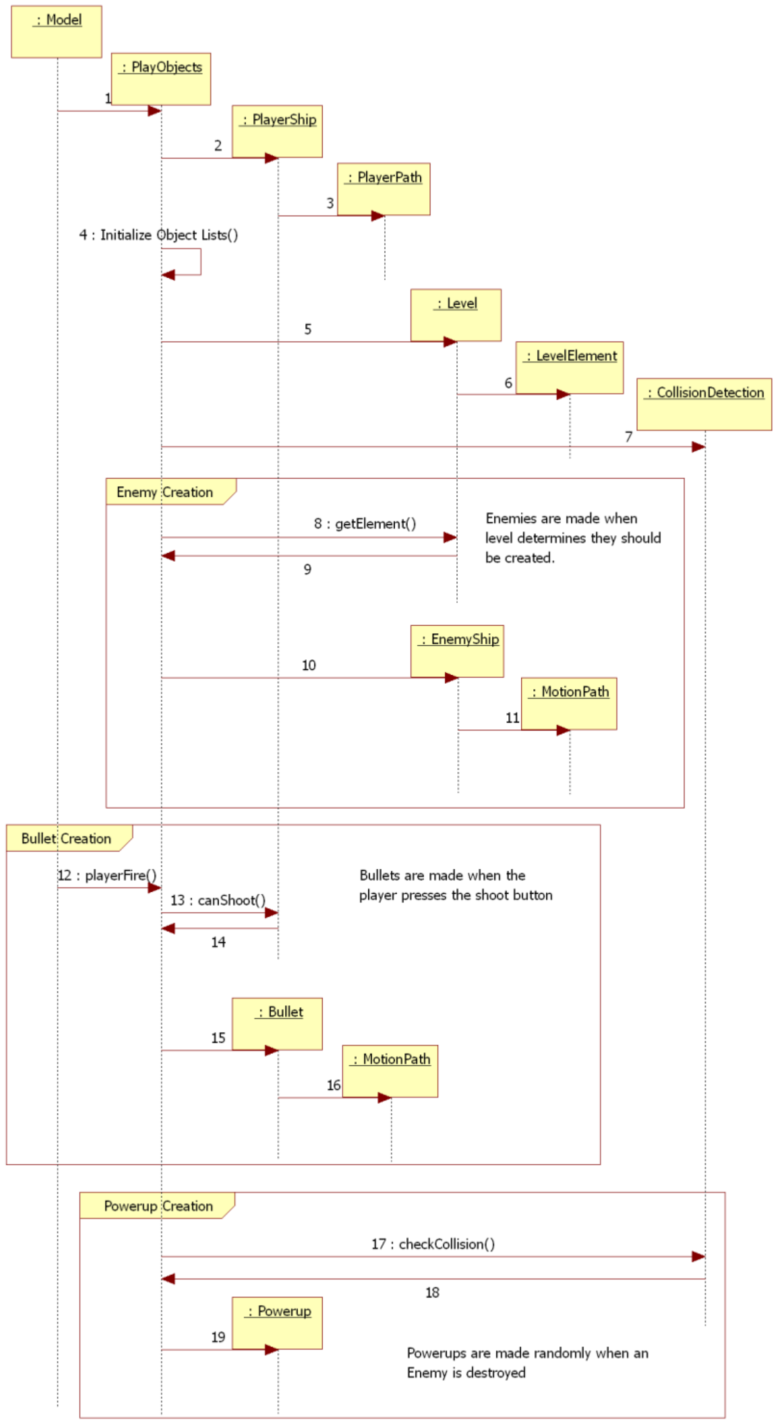
- Sets the keys pressed in the model, via the IControllerModel interface.
- Updates the Model by calling the model's update method via the model interface.
- Gets a list of sounds that need to be played from the Model via the model interface.
- Plays the sounds
- Redraws the Panel by calling the View's repaint method via the IControllerView interface.

The following code snippet shows our Controller's run method:

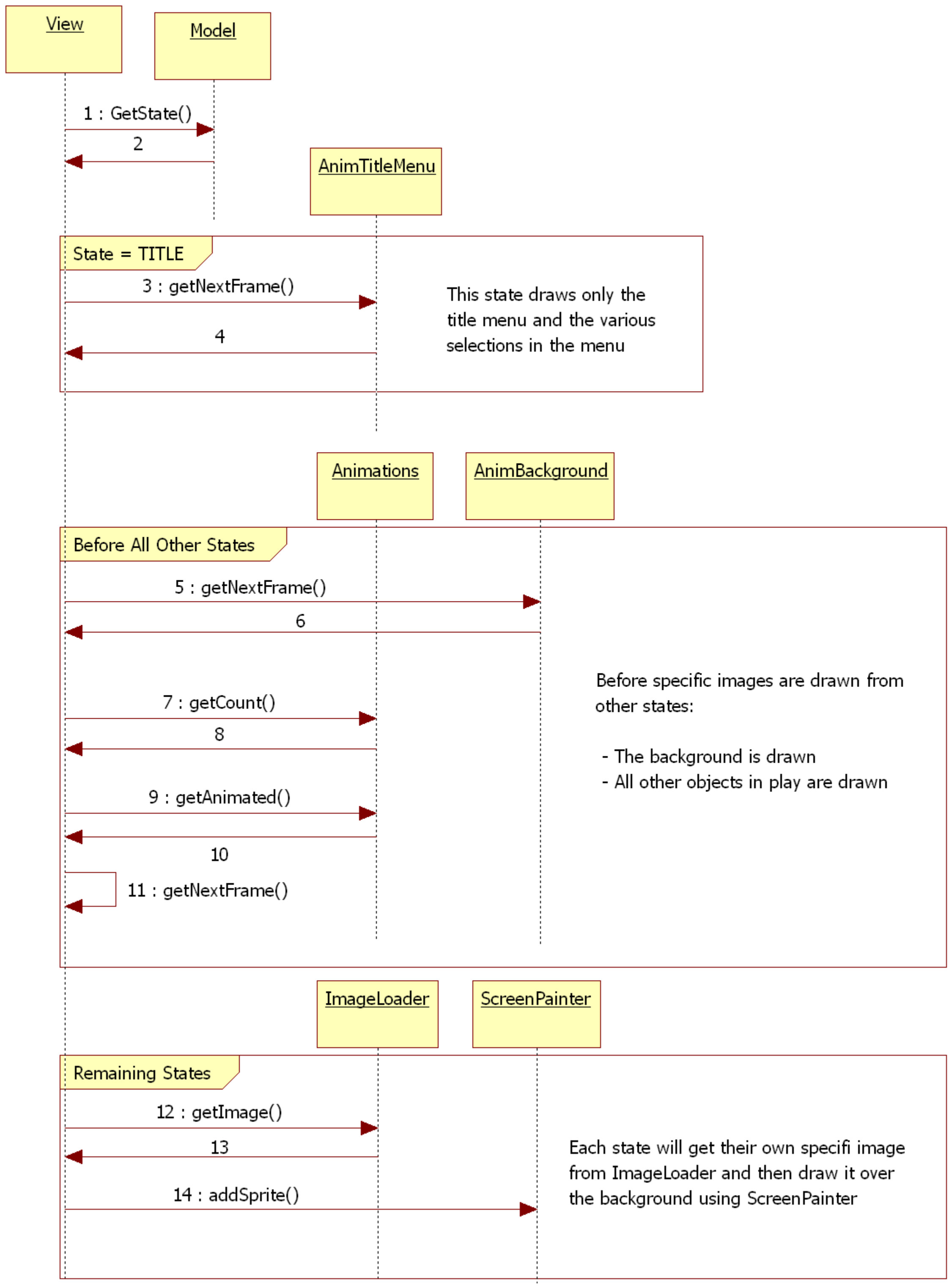
```
public void run() {

      // timer
      int timerValue = 1000/Config.cGameSpeed;
      Timer timer = new Timer(timerValue, new ActionListener() {
        public void actionPerformed(ActionEvent e) {

           // Set the Keys Pressed
          if(mKeyWatcher.isPressed(kup)){mModel.setUp(true);}
          if(mKeyWatcher.isPressed(kleft)){mModel.setLeft(true);};
          if(mKeyWatcher.isPressed(kdown)){mModel.setDown(true);};
          if(mKeyWatcher.isPressed(kright)){mModel.setRight(true);};
          if(mKeyWatcher.isPressed(kshoot)){mModel.setShooting(true);};
          if(mKeyWatcher.isPressed(kpause)){mModel.setPause(true);};

          // Update the model
          mModel.update();

          // Get Sounds
          ArrayList<Boolean> soundCommands = mModel.getSounds();

          // Play Sounds
          for(int i = 0;i<soundCommands.size()-1;i++){
            if(soundCommands.get(i))
              mSoundPlayer.playSoundEffect(mSoundLoader.getClip(i));}
          if(soundCommands.get(3))
            mSoundPlayer.pauseMusic();
           if(soundCommands.get(5)||soundCommands.get(4)||
                   soundCommands.get(0)){
            mSoundPlayer.stopMusic();}
          if(soundCommands.get(6))
            mSoundPlayer.startMusic();

          // Redraw the View
          mView.repaint();

        }});
       timer.start();
}}
```

4.3 Building the Play Area

During the Model's update process, if the game state changes from title to gameplay, the Model will call a private method called startGameplay. This method creates an instance of PlayObjects. PlayObjects in turn instantiates all of the objects required to run the Gameplay stage of the game. The following Sequence Diagram Illustrates this process. See **Figure 6** below:



4.4 Drawing Process in the View Package:


View
Model
1 : GetState()
2
AnimTitleMenu
State = TITLE
3 : getNextFrame()
4
This state draws only the title menu and the various selections in the menu
Animations
AnimBackground
Before All Other States
5 : getNextFrame()
6
7 : getCount()
8
9 : getAnimated()
10
11 : getNextFrame()
Before specific images are drawn from other states:
- The background is drawn
- All other objects in play are drawn
ImageLoader
ScreenPainter
Remaining States
12 : getImage()
13
14 : addSprite()
Each state will get their own specifi image from ImageLoader and then draw it over the background using ScreenPainter

5. Design Choices

Design Principles:

We feel that our design adheres all of the design principles discussed in class. The following are some examples highlighting our adherence to specific design principles:

Open-Closed Principle:

Our project was designed such that adding new features required creating new code and not modifying code. This is shown by the fact that our class diagrams have undergone relatively few changes after the addition of many extra features. For example, since the first iteration design, we have added tracking paths (give enemies the ability to chase the player) and powerups. Adding this functionality only required the new class "Powerup" to implement MovableObject and the new class "TrackingPath" to implement MotionPath. See figure 8 below.

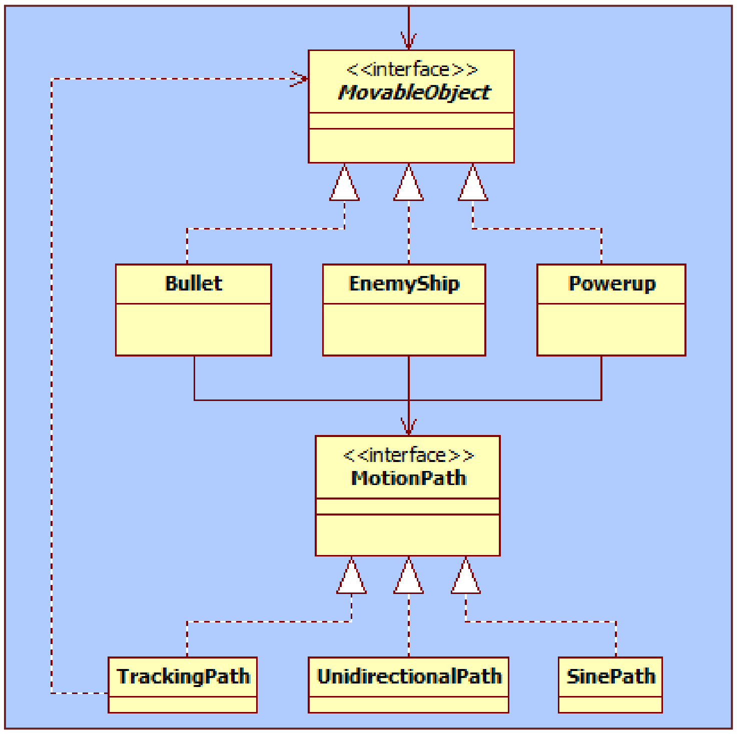


Figure 8 – MovableObject-MotionPath Structure

Liskov Substitution Principle:

The LSP states that "Subtypes must be substitutable for their base types." A good example of this in our code is again the MovableObject structure shown in Figure 8. Any object that implement MovableObject, whether it be an enemy, projectile, powerup or the player (not shown in fig.8) can be used in the collision detection class shown in Figure 9 (Next Page.) This has allowed us to greatly simplify the collision detection function.

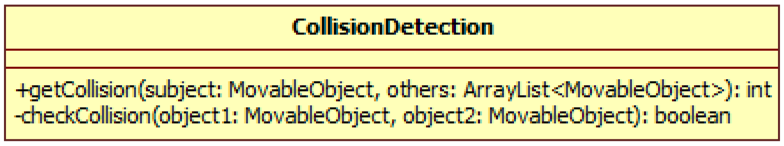

CollisionDetection
+getCollision(subject: MovableObject, others: ArrayList<MovableObject>): int
-checkCollision(object1: MovableObject, object2: MovableObject): boolean

Figure 9 – Collision Detection Class

The collision detection class takes one MovableObject and an ArrayList of MovableObjects and checks the collisions of the one object against each object in the ArrayList. So whether the objects are enemies, projectiles, etc. it does not make any difference to the collision detection because the objects are substitutable for MovableObjects.

Design Patterns:

We have employed the following design patterns in our project:

Model-View-Controller:
We chose to use the MVC pattern because it helped to organize our project into classes. It separates the game's logic from how the game is displayed. See Figure 1 on page 10 for the high-level class diagram showing the model-view-controller structure.

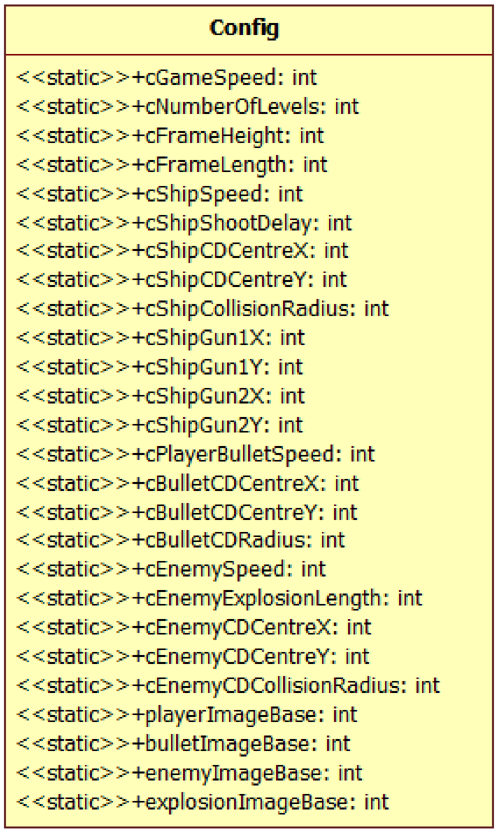
Façade:
We are using the Façade pattern to facilitate the communication between classes. This can be seen on the high-level class diagram (Figure 1, page 10.) The Controller can get and set information in the model through the *IControllerModel* interface and to the view through the *IControllerView* interface. The view can also communicate to the model through the *IViewModel* interface. This simplifies the interaction of the different packages by allowing one interface to handle the communication.

Strategy:
In the model package, the interface MotionPath is an example of the strategy pattern. See Figure 8 on the previous page. The position of any movableObject can be updated by various implementations of this interface. Thus, identical enemies can move in completely different ways if their motion paths are different.

6. Notable Changes from First Design

Configuration Class:

We have added one class to the Controller Package that we feel has greatly improved our design. The configuration class holds the values of game parameters that need to be adjusted while tuning the game. This has reduced the amount of parameters that need to be passed from one function to another.



Config

```
<<static>>+cGameSpeed: int
<<static>>+cNumberOfLevels: int
<<static>>+cFrameHeight: int
<<static>>+cFrameLength: int
<<static>>+cShipSpeed: int
<<static>>+cShipShootDelay: int
<<static>>+cShipCDCentreX: int
<<static>>+cShipCDCentreY: int
<<static>>+cShipCollisionRadius: int
<<static>>+cShipGun1X: int
<<static>>+cShipGun1Y: int
<<static>>+cShipGun2X: int
<<static>>+cShipGun2Y: int
<<static>>+cPlayerBulletSpeed: int
<<static>>+cBulletCDCentreX: int
<<static>>+cBulletCDCentreY: int
<<static>>+cBulletCDRadius: int
<<static>>+cEnemySpeed: int
<<static>>+cEnemyExplosionLength: int
<<static>>+cEnemyCDCentreX: int
<<static>>+cEnemyCDCentreY: int
<<static>>+cEnemyCDCollisionRadius: int
<<static>>+playerImageBase: int
<<static>>+bulletImageBase: int
<<static>>+enemyImageBase: int
<<static>>+explosionImageBase: int
```

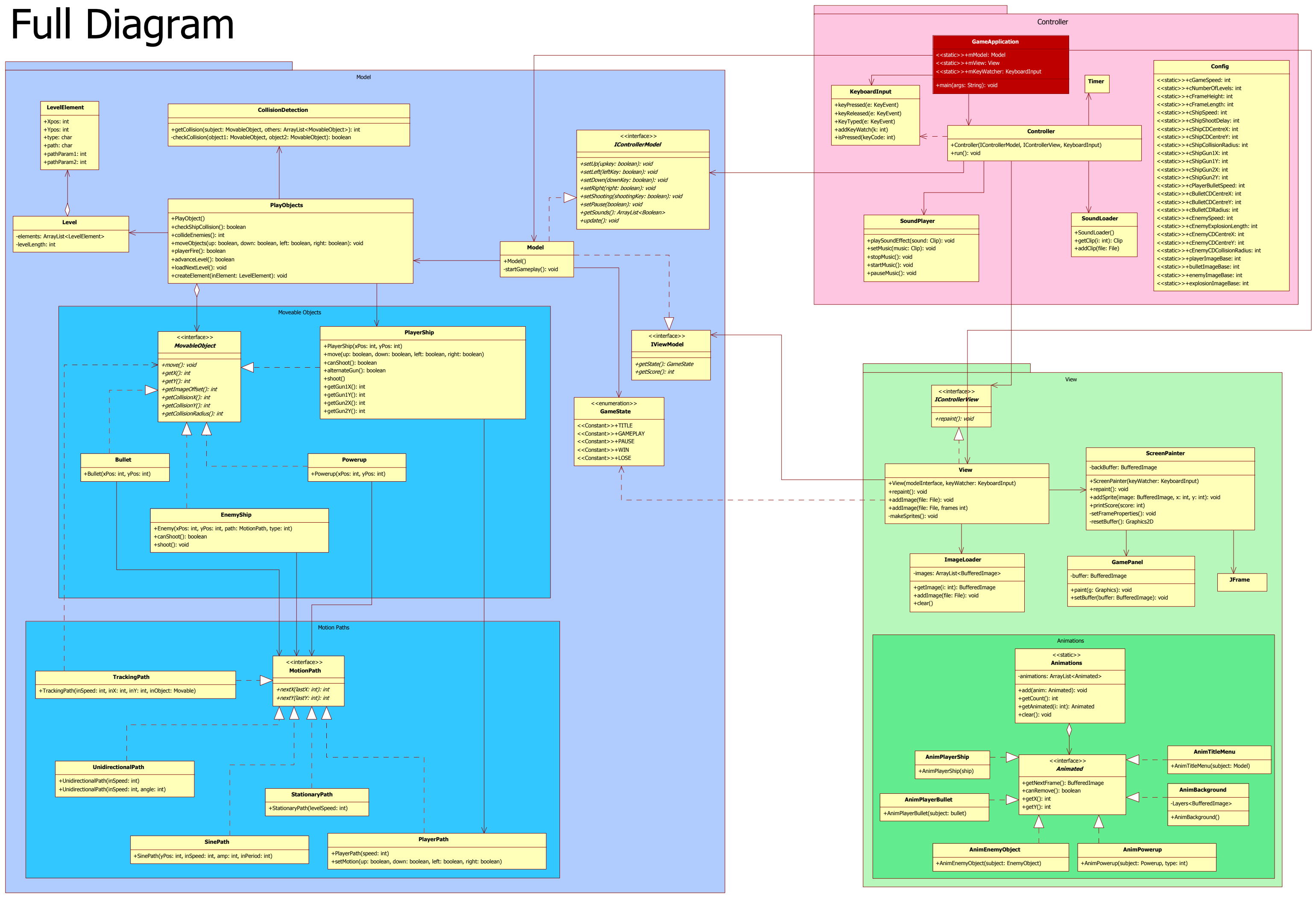
Figure 10 – Configuration Class

Animation:

One of the larger changes to program was moving animations to the view. In the first iteration the animations were handled by several classes and in the end drawn by view. The issue became that the view was handed a new set of objects to draw each frame which made having different types of animations in a row difficult to achieve, such as having an explosion animation play after an enemy is destroyed.

Animations are now handled by the view in a class called 'Animations' which contains a list of objects implementing the 'Animated' interface. 'Animated' objects can be made and added to the 'Animations' class by any class in the program that required an animation. Because of how the implementation of 'Animated' only requires a BufferedImage to be returned; how the 'animated' object creates the BufferedImage can be as sophisticated as is necessary. In essence, an animation could have several sub-animations inside itself.

Full Diagram


Model
LevelElement
+Xpos: int
+Ypos: int
+type: char
+path: char
+pathParam1: int
+pathParam2: int
CollisionDetection
+getCollision(subject: MovableObject, others: ArrayList<MovableObject>): int
-checkCollision(object1: MovableObject, object2: MovableObject): boolean
Level
-elements: ArrayList<LevelElement>
-levelLength: int
PlayObjects
+PlayObject()
+checkShipCollision(): boolean
+collideEnemies(): int
+moveObjects(up: boolean, down: boolean, left: boolean, right: boolean): void
+playerFire(): boolean
+advanceLevel(): boolean
+loadNextLevel(): void
+createElement(inElement: LevelElement): void
<<interface>>
IControllerModel
+setUp(upkey: boolean): void
+setLeft(leftKey: boolean): void
+setDown(downKey: boolean): void
+setRight(right: boolean): void
+setShooting(shootingKey: boolean): void
+setPause(boolean): void
+getSounds(): ArrayList<Boolean>
+update(): void
Model
+Model()
-startGameplay(): void
<<interface>>
IViewModel
+getState(): GameState
+getScore(): int
<<enumeration>>
GameState
<<Constant>>+TITLE
<<Constant>>+GAMEPLAY
<<Constant>>+PAUSE
<<Constant>>+WIN
<<Constant>>+LOSE
Moveable Objects
<<interface>>
MovableObject
+move(): void
+getX(): int
+getY(): int
+getImageOffset(): int
+getCollisionX(): int
+getCollisionY(): int
+getCollisionRadius(): int
PlayerShip
+PlayerShip(xPos: int, yPos: int)
+move(up: boolean, down: boolean, left: boolean, right: boolean)
+canShoot(): boolean
+alternateGun(): boolean
+shoot()
+getGun1X(): int
+getGun1Y(): int
+getGun2X(): int
+getGun2Y(): int
Bullet
+Bullet(xPos: int, yPos: int)
Powerup
+Powerup(xPos: int, yPos: int)
EnemyShip
+Enemy(xPos: int, yPos: int, path: MotionPath, type: int)
+canShoot(): boolean
+shoot(): void
Motion Paths
<<interface>>
MotionPath
+nextX(lastX: int): int
+nextY(lastY: int): int
TrackingPath
+TrackingPath(inSpeed: int, inX: int, inY: int, inObject: Movable)
UnidirectionalPath
+UnidirectionalPath(inSpeed: int)
+UnidirectionalPath(inSpeed: int, angle: int)
StationaryPath
+StationaryPath(levelSpeed: int)
SinePath
+SinePath(yPos: int, inSpeed: int, amp: int, inPeriod: int)
PlayerPath
+PlayerPath(speed: int)
+setMotion(up: boolean, down: boolean, left: boolean, right: boolean)
Controller
GameApplication
<<static>>+mModel: Model
<<static>>+mView: View
<<static>>+mKeyWatcher: KeyboardInput
+main(args: String): void
KeyboardInput
+keyPressed(e: KeyEvent)
+keyReleased(e: KeyEvent)
+KeyTyped(e: KeyEvent)
+addKeyWatch(k: int)
+isPressed(keyCode: int)
Timer
Controller
+Controller(IControllerModel, IControllerView, KeyboardInput)
+run(): void
SoundPlayer
+playSoundEffect(sound: Clip): void
+setMusic(music: Clip): void
+stopMusic(): void
+startMusic(): void
+pauseMusic(): void
SoundLoader
+SoundLoader()
+getClip(i: int): Clip
+addClip(file: File)
Config
<<static>>+cGameSpeed: int
<<static>>+cNumberOfLevels: int
<<static>>+cFrameHeight: int
<<static>>+cFrameLength: int
<<static>>+cShipSpeed: int
<<static>>+cShipShootDelay: int
<<static>>+cShipCDCentreX: int
<<static>>+cShipCDCentreY: int
<<static>>+cShipCollisionRadius: int
<<static>>+cShipGun1X: int
<<static>>+cShipGun1Y: int
<<static>>+cShipGun2X: int
<<static>>+cShipGun2Y: int
<<static>>+cPlayerBulletSpeed: int
<<static>>+cBulletCDCentreX: int
<<static>>+cBulletCDCentreY: int
<<static>>+cBulletCDRadius: int
<<static>>+cEnemySpeed: int
<<static>>+cEnemyExplosionLength: int
<<static>>+cEnemyCDCentreX: int
<<static>>+cEnemyCDCentreY: int
<<static>>+cEnemyCDCollisionRadius: int
<<static>>+playerImageBase: int
<<static>>+bulletImageBase: int
<<static>>+enemyImageBase: int
<<static>>+explosionImageBase: int
View
<<interface>>
IControllerView
+repaint(): void
View
+View(modelInterface, keyWatcher: KeyboardInput)
+repaint(): void
+addImage(file: File): void
+addImage(file: File, frames int)
-makeSprites(): void
ScreenPainter
-backBuffer: BufferedImage
+ScreenPainter(keyWatcher: KeyboardInput)
+repaint(): void
+addSprite(image: BufferedImage, x: int, y: int): void
+printScore(score: int)
-setFrameProperties(): void
-resetBuffer(): Graphics2D
ImageLoader
-images: ArrayList<BufferedImage>
+getImage(i: int): BufferedImage
+addImage(file: File): void
+clear()
GamePanel
-buffer: BufferedImage
+paint(g: Graphics): void
+setBuffer(buffer: BufferedImage): void
JFrame
Animations
<<static>>
Animations
-animations: ArrayList<Animated>
+add(anim: Animated): void
+getCount(): int
+getAnimated(i: int): Animated
+clear(): void
AnimPlayerShip
+AnimPlayerShip(ship)
<<interface>>
Animated
+getNextFrame(): BufferedImage
+canRemove(): boolean
+getX(): int
+getY(): int
AnimTitleMenu
+AnimTitleMenu(subject: Model)
AnimPlayerBullet
+AnimPlayerBullet(subject: bullet)
AnimBackground
-Layers<BufferedImage>
+AnimBackground()
AnimEnemyObject
+AnimEnemyObject(subject: EnemyObject)
AnimPowerup
+AnimPowerup(subject: Powerup, type: int)