Gravity

Group 10

Daniel Walz

Jesper Ekhall

Lukas Kalinski

Fredrik Nordh

Alexander Nyberg

1. Introduction

This document contains the design of the gravity game. It's purpose is to guide when implementing the system. It describes the object oriented design complete with classes, dependencies, methods and associated requirements.

The intended audience are primarily developers of the project but also other stakeholders such as project managers, testers and commissioners. The reader of this document is assumed to have read the requirement document.

This document pertains to version 1.0 of the gravity game.

Related documents are:

- Requirement document
- Implementation plan

The implementation plan is a result of this document.

Glossary

For glossary refer to the requirement document.

Abstract

In section 2 of this document we provide a general overview of the system. There is also a description of the architecture.

In section 3 there are assumptions and dependencies of operating environment etc.

Section 4 contains a description of the user interface including functionality and appearance.

In section 5 there is a detailed description of the object oriented design.

Section 6 contains test cases for each functional requirement.

2. System Overview

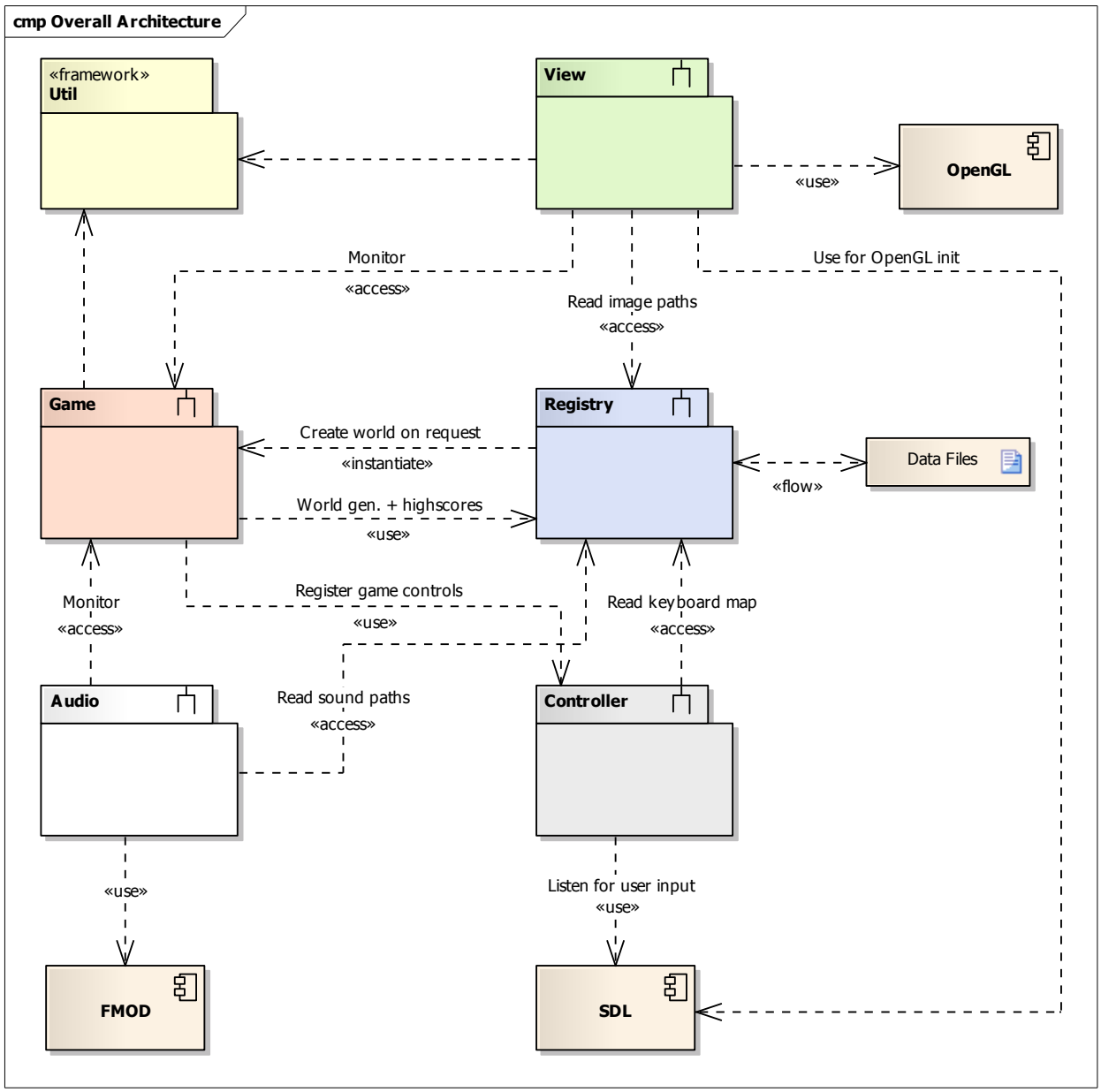
2.1 General description

Gravity is a game consisting of planets with their corresponding gravities and space ships with the ability to fire at least one weapon. Both space ships and some weapon projectiles will be affected by gravity. The key idea is to combine classic game shooting with gravity constraints, forcing the player to think about more factors than just where to shoot.

The design uses the basic Model View Controller (MVC) pattern. The idea of this pattern is to have the world and it's object in one place (known as Model), the View handles the rendering and the Controller handles changes to the Model (by means of simulated physics).

The design is prepared for the possible extension of networked multiplayer functionality in the future. This by means of a client server architecture.

2.2 Overall Architecture Description

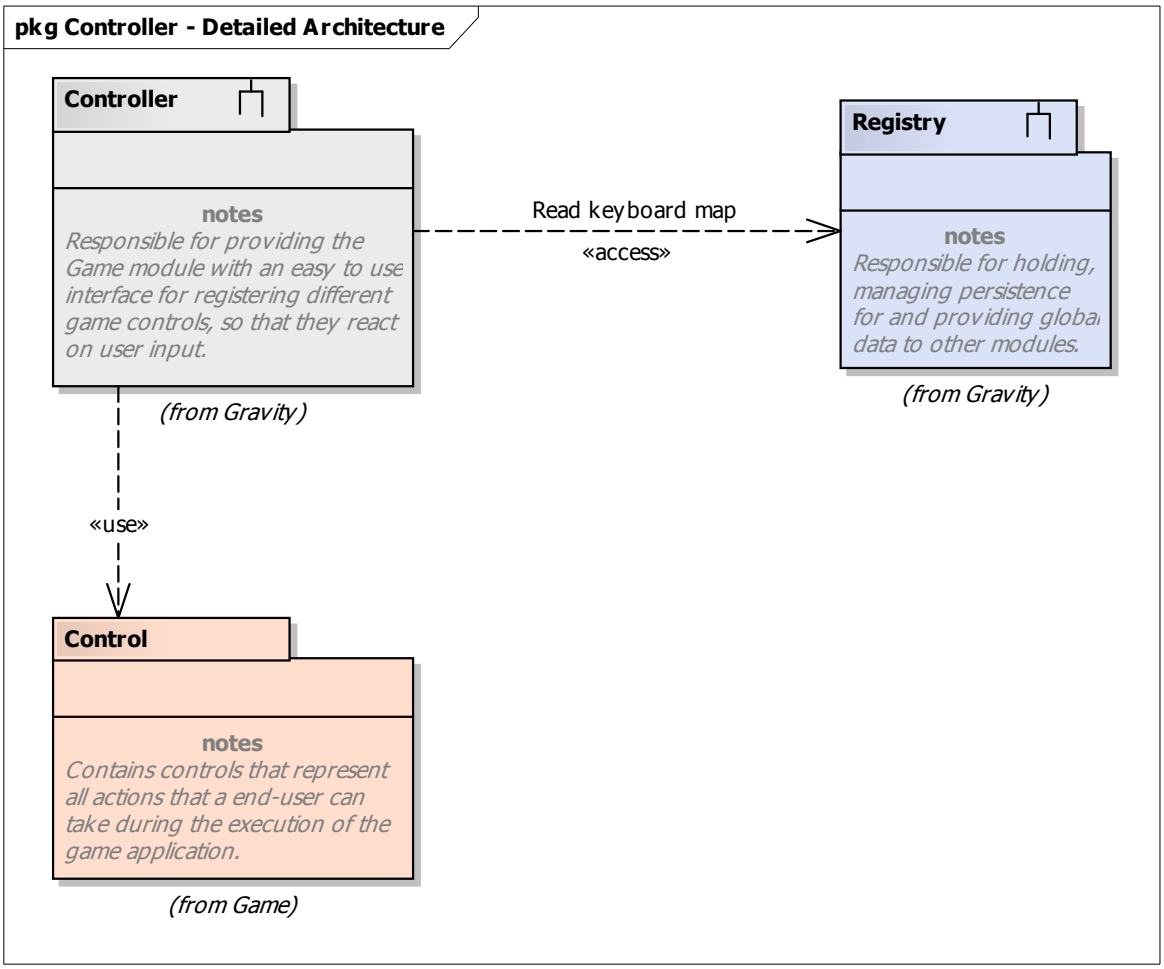


The system will consist of six modules. Which we in this (2.2) and the next (2.3) section will show you in Unified Modeling Language. The Game module is the game core, this is where all game-related operations, such as affecting the game world using an internal physics package as

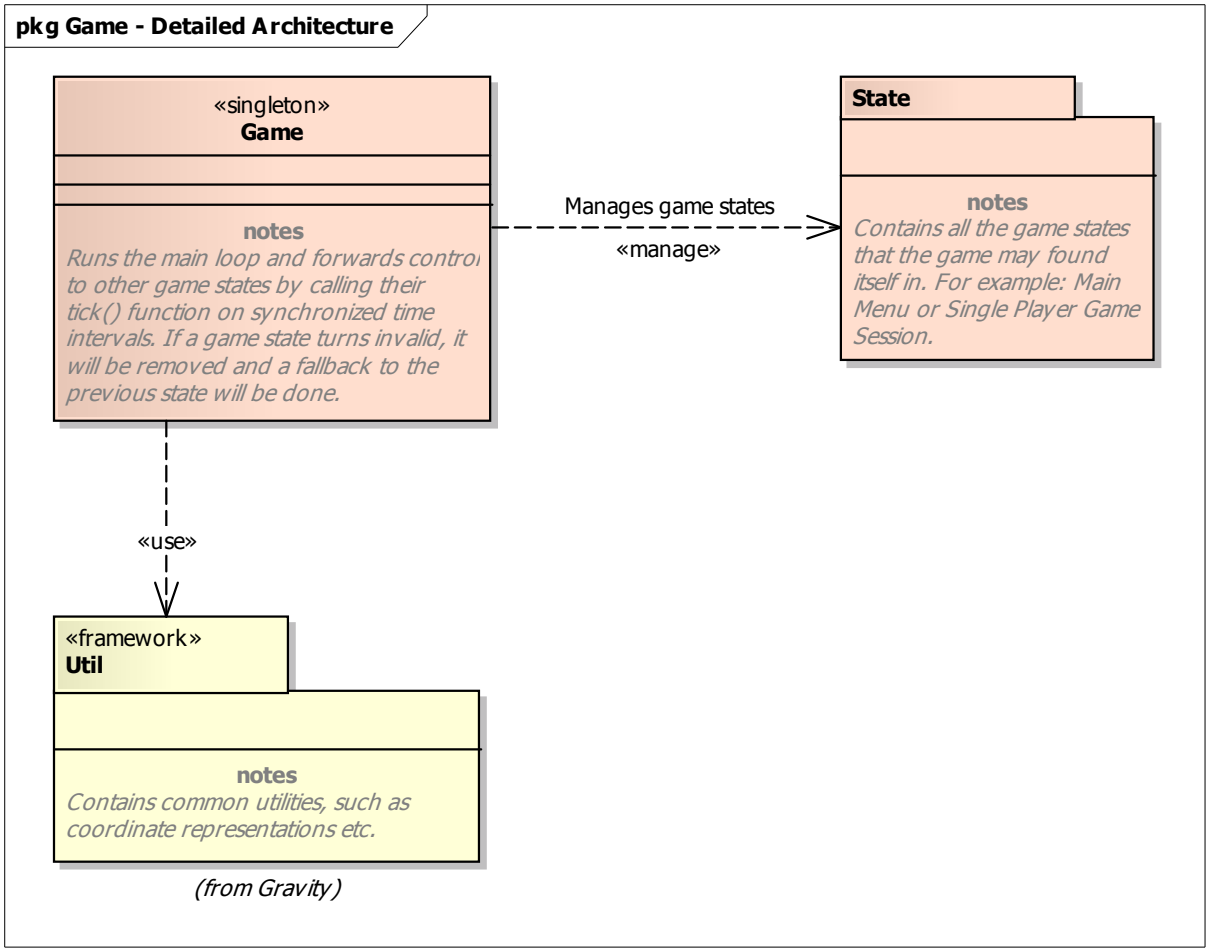
well as switching between the different game states, will be done. The View and Audio modules will be used by the game module to trigger visual and auditory feedback on what's happening in the game world. Further, these modules will use third party tools to realize their purposes, i.e., the View module will use the Simple DirectMedia Layer (SDL) library to initialize and control the visual experience, and the Audio module will use the FMOD API to play sound effects. The Controller module will be responsible for interpreting user input into game commands, by using matching functionality in the SDL library. The DataStore module will manage shared data, such as for example a player's position in the game world, as well as the game world itself. The DataPersistence module will be used by the DataStore module to write persistent data into files.

2.3 Detailed Architecture

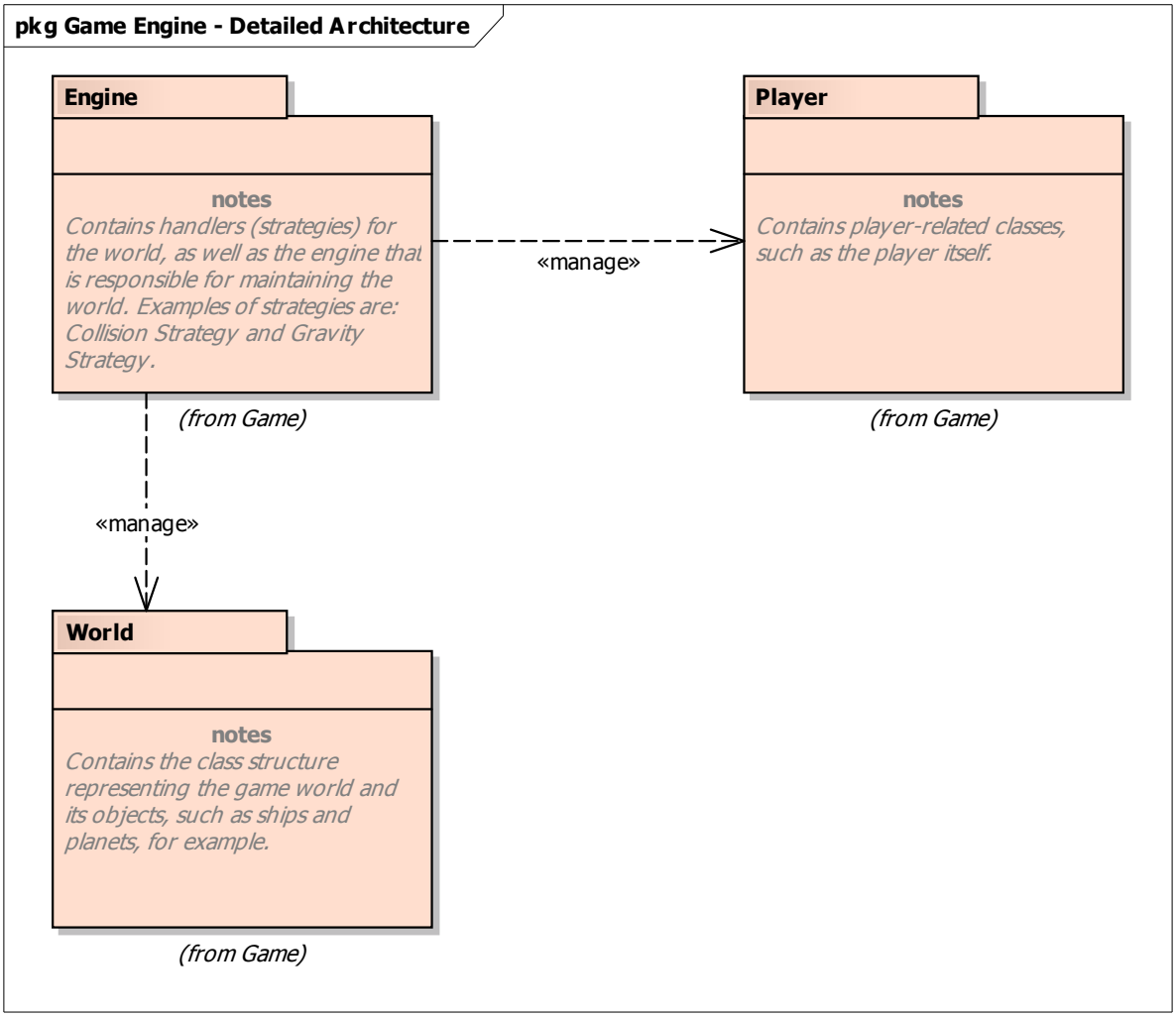
2.3.1 Controller



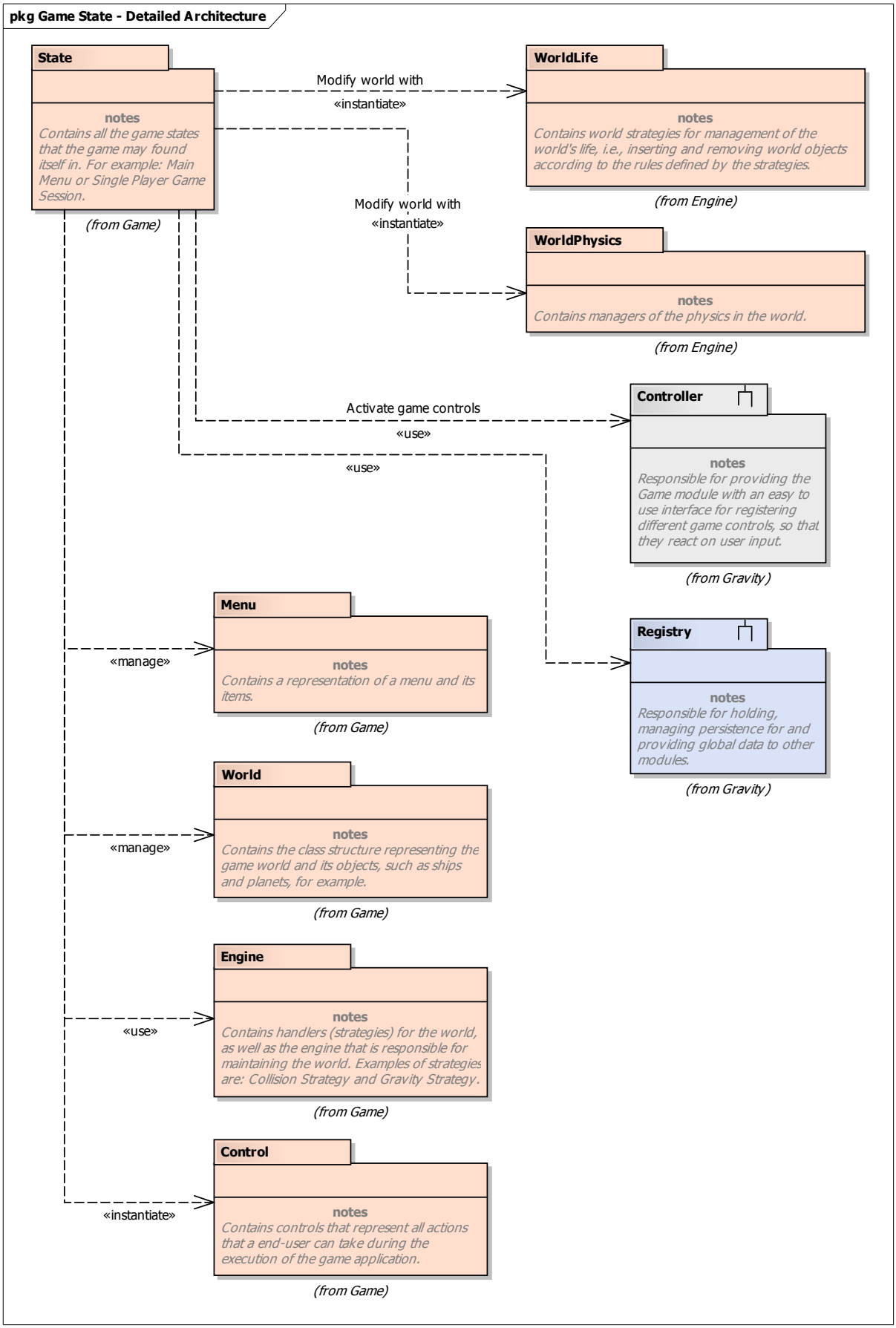
2.3.2 Game



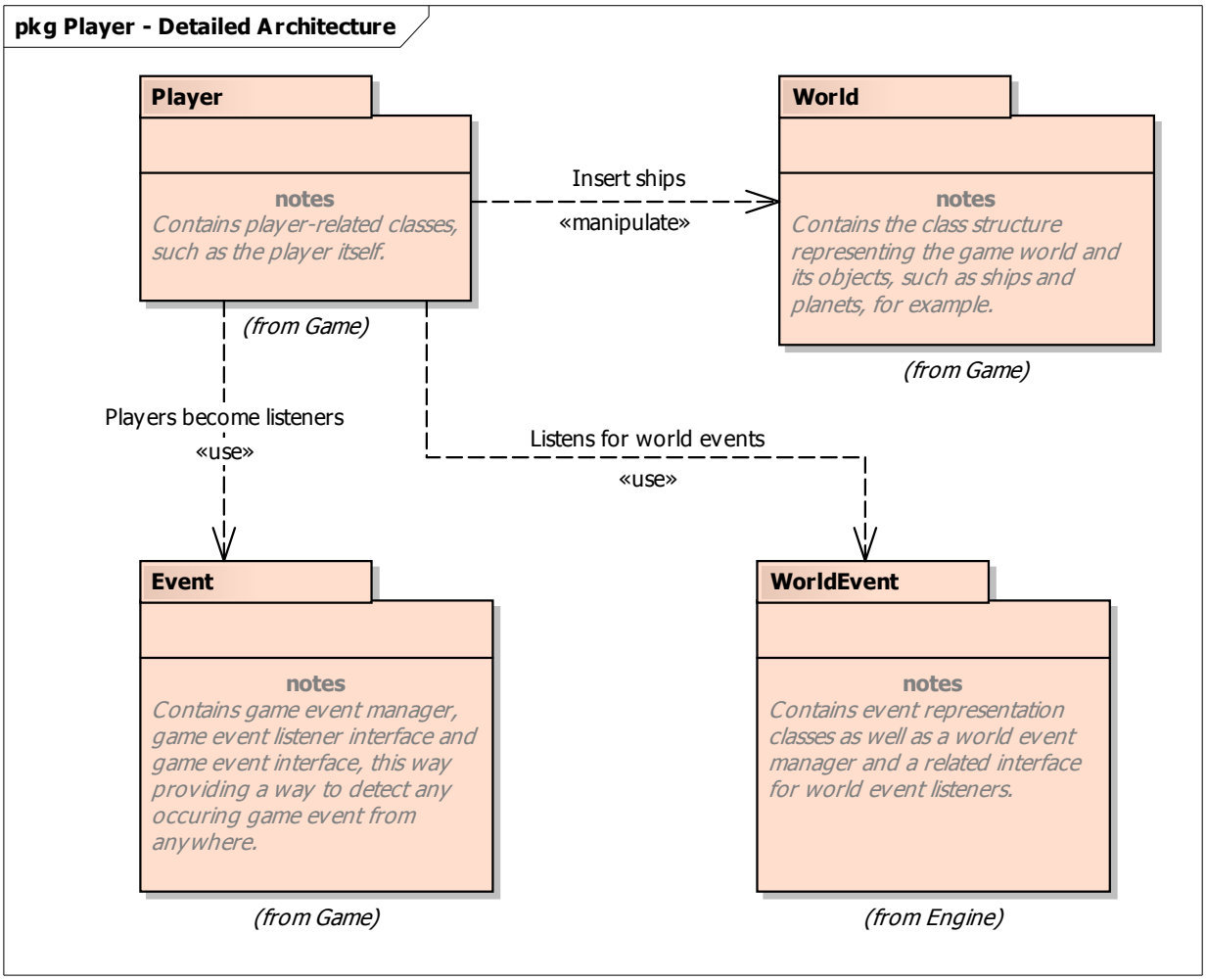
2.3.3 Game Engine



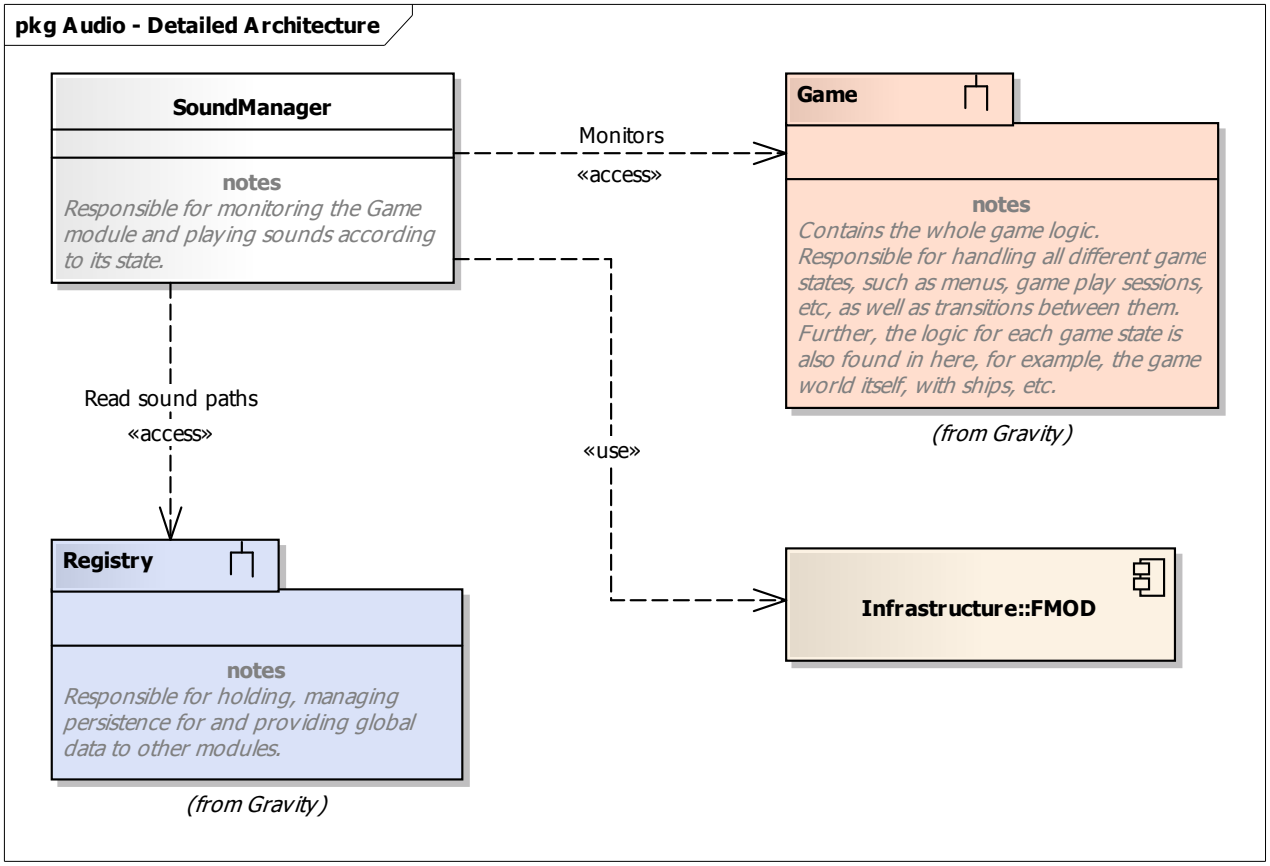
2.3.4 Game State


pkg Game State - Detailed Architecture
State
notes
Contains all the game states that the game may found itself in. For example: Main Menu or Single Player Game Session.
(from Game)
Modify world with
«instantiate»
WorldLife
notes
Contains world strategies for management of the world's life, i.e., inserting and removing world objects according to the rules defined by the strategies.
(from Engine)
Modify world with
«instantiate»
WorldPhysics
notes
Contains managers of the physics in the world.
(from Engine)
Activate game controls
«use»
«use»
Controller
notes
Responsible for providing the Game module with an easy to use interface for registering different game controls, so that they react on user input.
(from Gravity)
«manage»
Menu
notes
Contains a representation of a menu and its items.
(from Game)
Registry
notes
Responsible for holding, managing persistence for and providing global data to other modules.
(from Gravity)
«manage»
World
notes
Contains the class structure representing the game world and its objects, such as ships and planets, for example.
(from Game)
«use»
Engine
notes
Contains handlers (strategies) for the world, as well as the engine that is responsible for maintaining the world. Examples of strategies are: Collision Strategy and Gravity Strategy.
(from Game)
«instantiate»
Control
notes
Contains controls that represent all actions that a end-user can take during the execution of the game application.
(from Game)

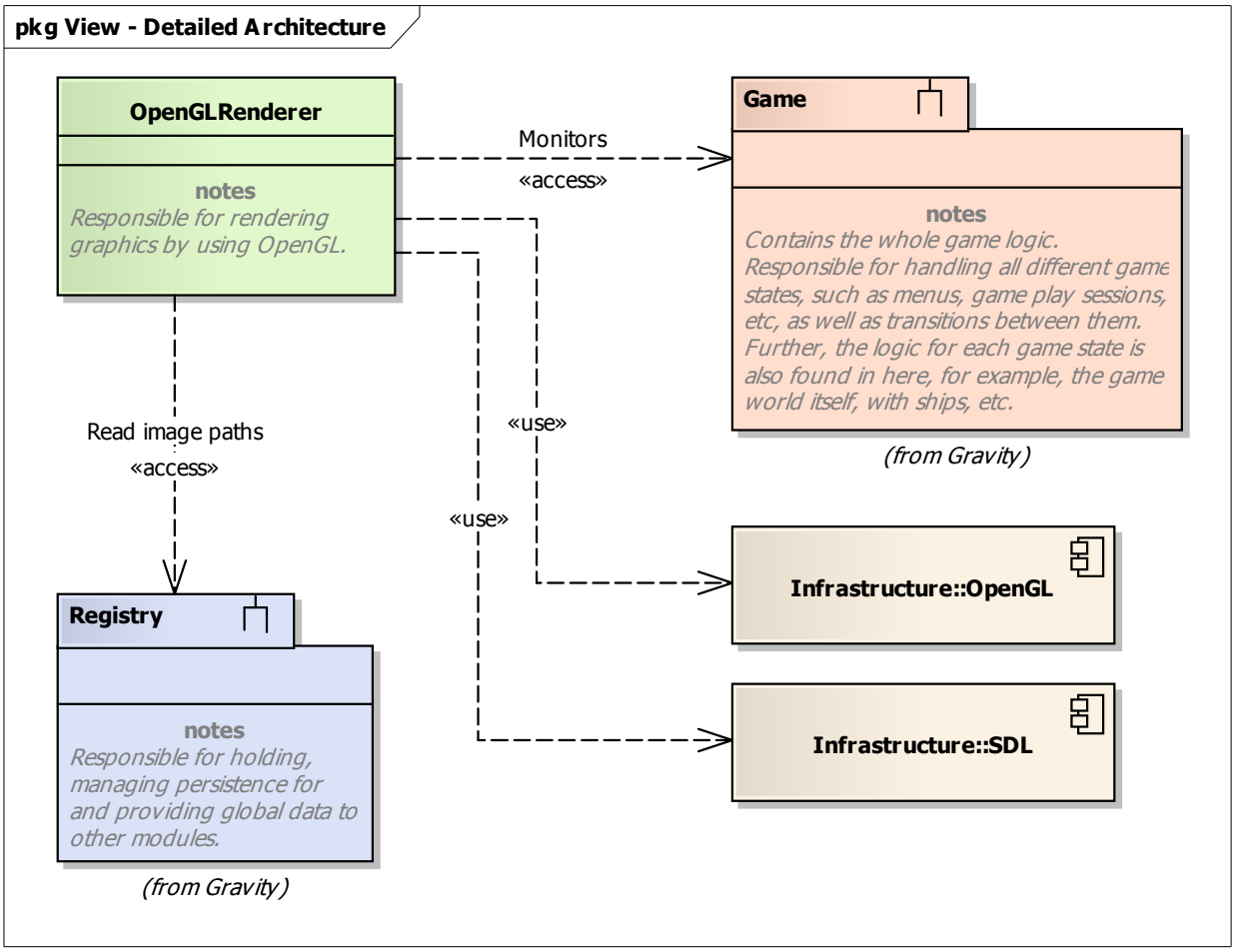
2.3.5 Player



2.3.6 Audio


pkg Audio - Detailed Architecture
SoundManager
notes
Responsible for monitoring the Game module and playing sounds according to its state.
Monitors
«access»
Game
notes
Contains the whole game logic. Responsible for handling all different game states, such as menus, game play sessions, etc, as well as transitions between them. Further, the logic for each game state is also found in here, for example, the game world itself, with ships, etc.
(from Gravity)
Read sound paths
«access»
«use»
Registry
notes
Responsible for holding, managing persistence for and providing global data to other modules.
(from Gravity)
Infrastructure::FMOD

2.3.7 View



3. Design considerations

3.1 Assumptions and dependencies

This system requires the software components Microsoft Windows XP and OpenGL. Hardware components needed are a Intel compatible PC with 1GB of RAM, 1GHz and a graphics card with hardware accelerated OpenGL.

The end users should be oriented in the windows operating systems.

3.2 General Constraints

A general constraint in real time graphical applications is performance, that's why we chose hardware accelerated OpenGL for graphics rendering.

Since the projects duration is severely limited we have tried to keep functionality simple and concise so as to be able to keep the schedule and have the project finished on time. Due to this constraint we have also tried to make the project easy to extend in the future.

Since all of the project members have little or no experience with larger software projects, this is also a factor in trying to keep the design small.

4. Graphical user interface

4.1 User interface overview

The main functionality for the user is divided in two parts. When starting the game the first part shown is the main menu. From the main menu the player can choose to see help, start single or multi player game, view high score, select player controls settings menu or to quit the game. When starting the game first a map choice menu appears. Also in multi player mode after selecting the map to play at, an option to choose how many lives each player should have will appear.

From the controls settings menu there are options to choose which function should be mapped to a key on the keyboard.

The in game view shows the players view of the game world, fuel, remaining lives, number of missiles etc. In multi player mode, the screen is split with one view for each player.

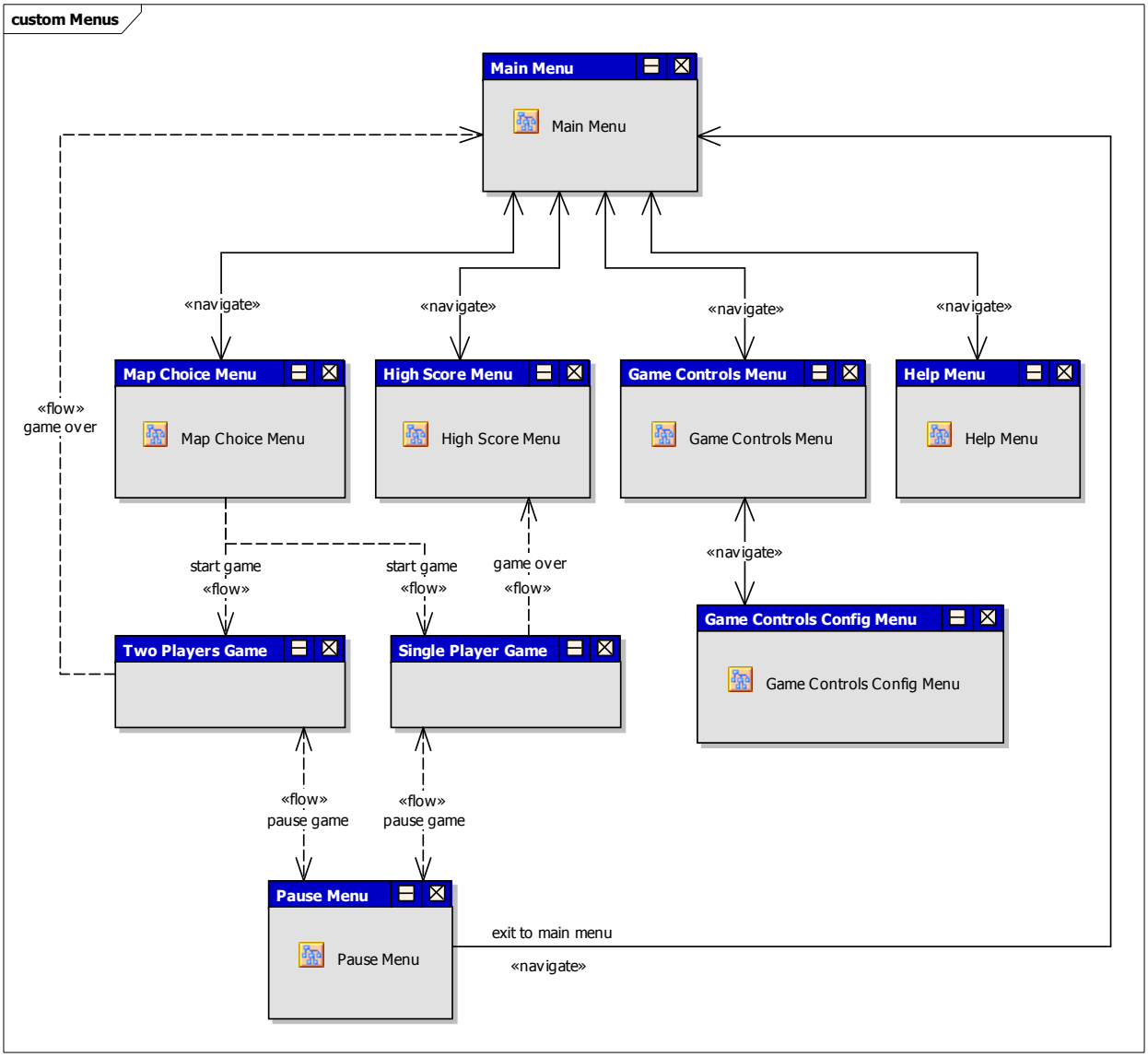
From the pause menu, there are only two choices: resume game and end game.

From the map choice menu you select which map to play from.

The high score screen shows a list of the players which have achieved the highest score in single player mode.

The help screen shows help for the game which explains the game play and the available options.

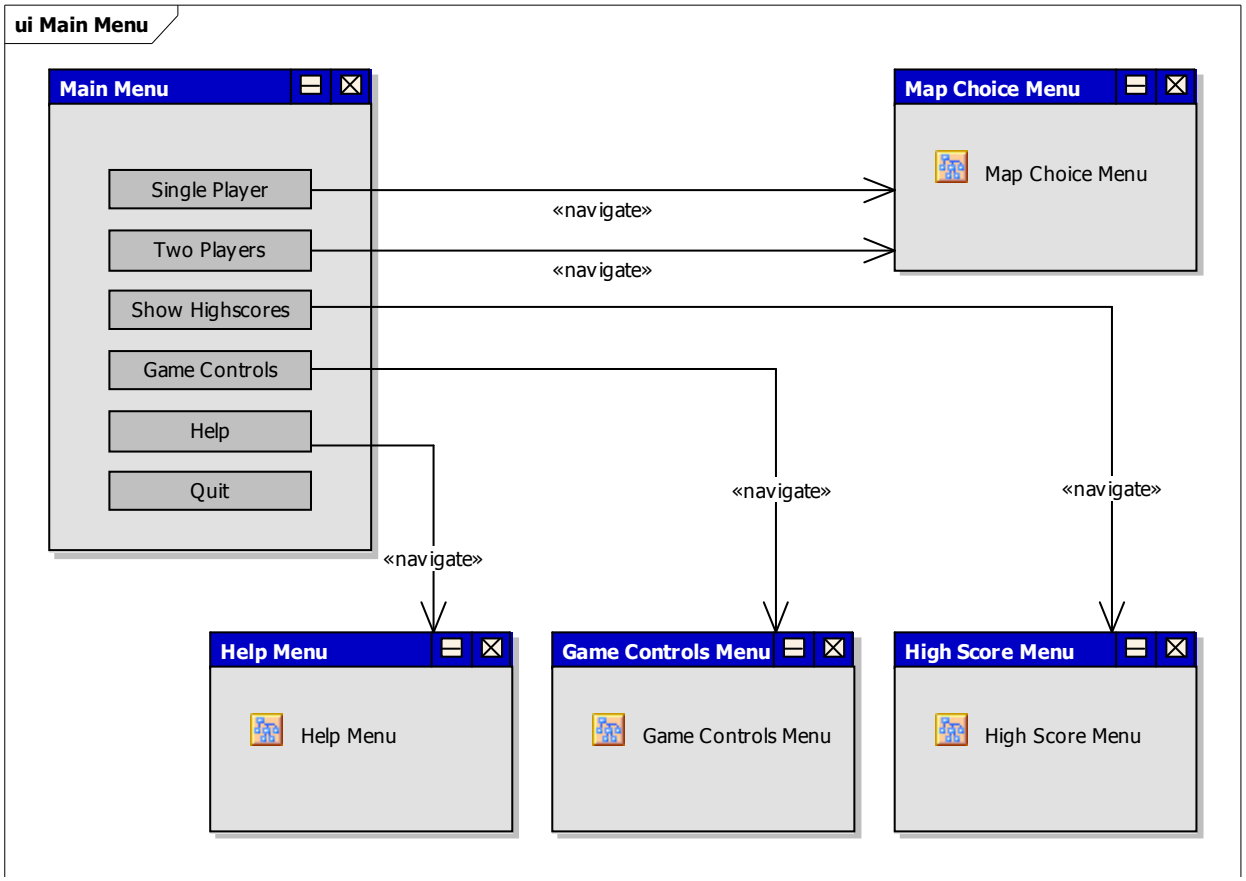
4.2 The GUI elements and functional requirements associated with them



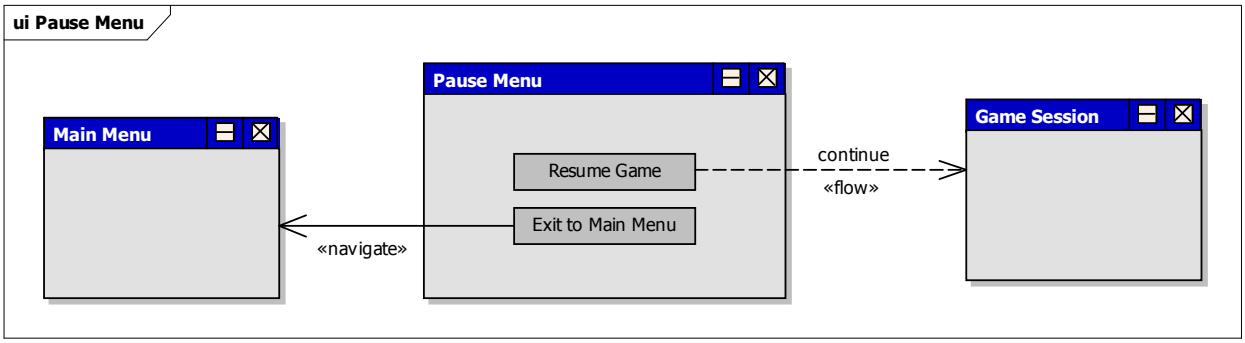
4.2.1 Main Menu

Functional requirements

- Single Player or Two Player Choice
- Exiting The Game



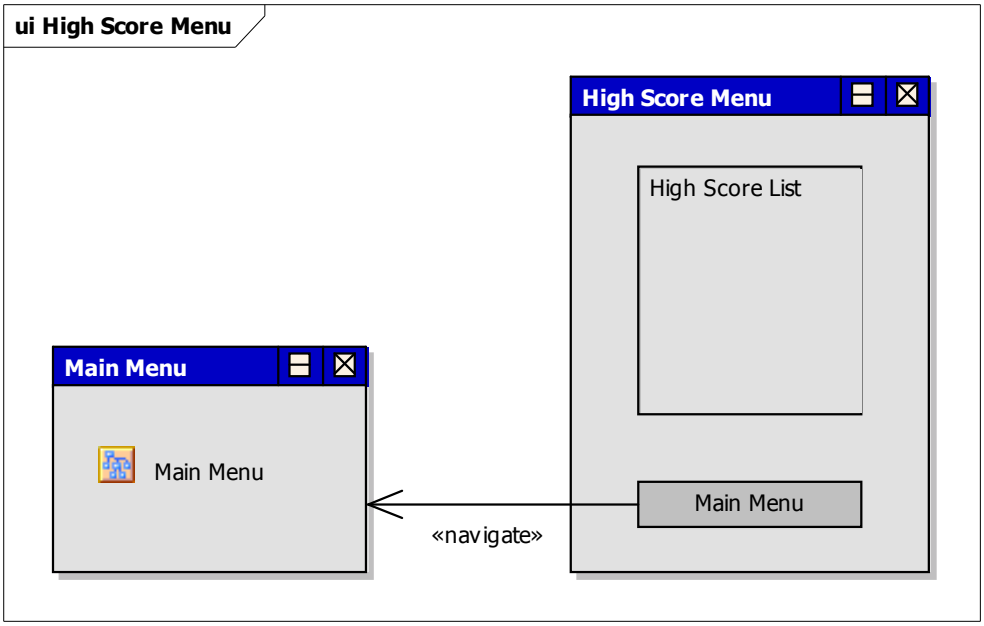
4.2.2 Pause Menu



4.2.3 High Score Menu

Functional requirements

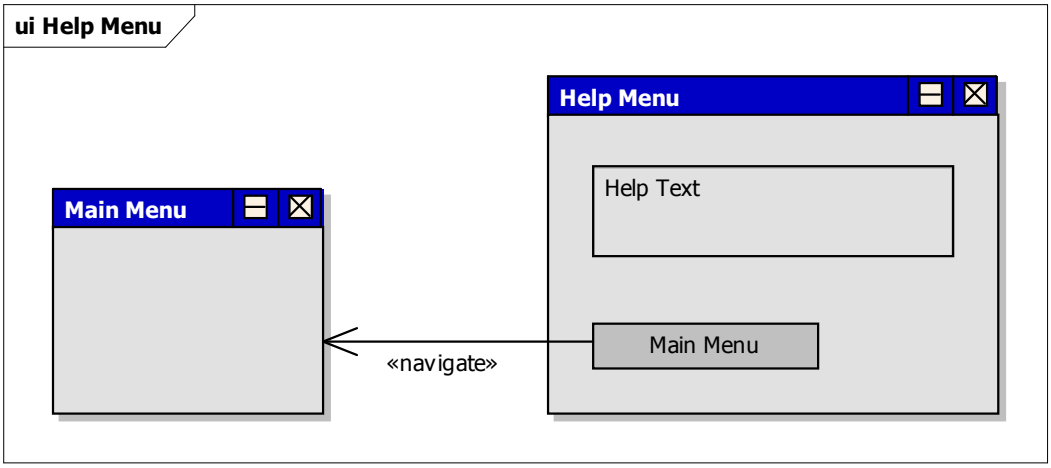
- Single Player High Score List



4.2.4 Help Menu

Functional requirements

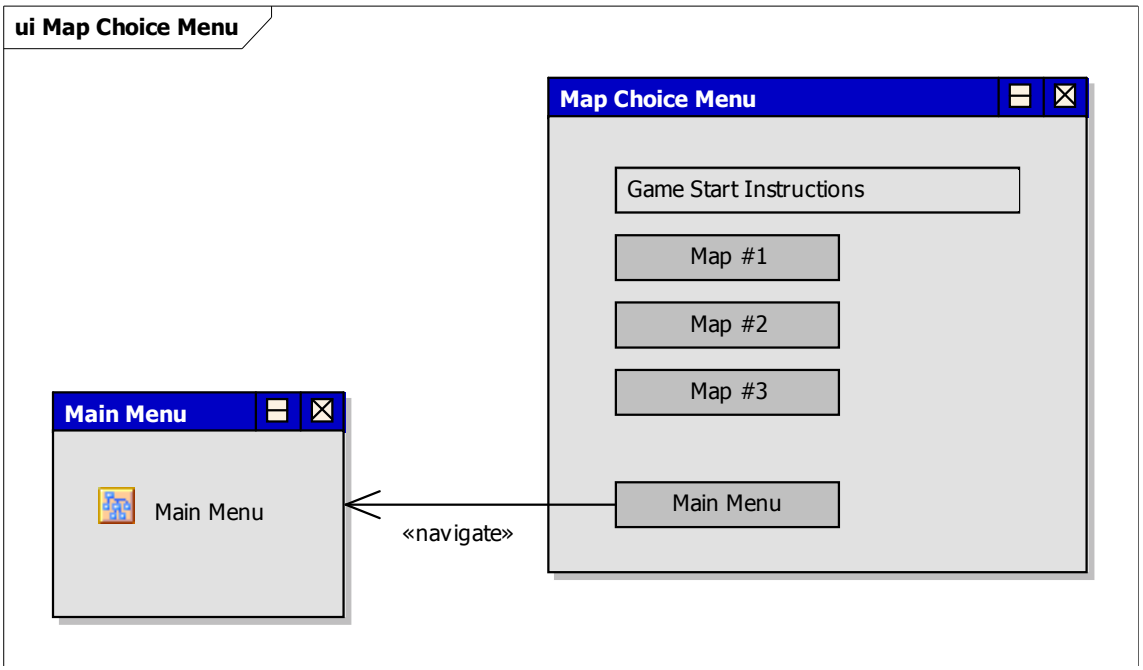
- Quick Start Help



4.2.5 Map Choice Menu

Functional requirements

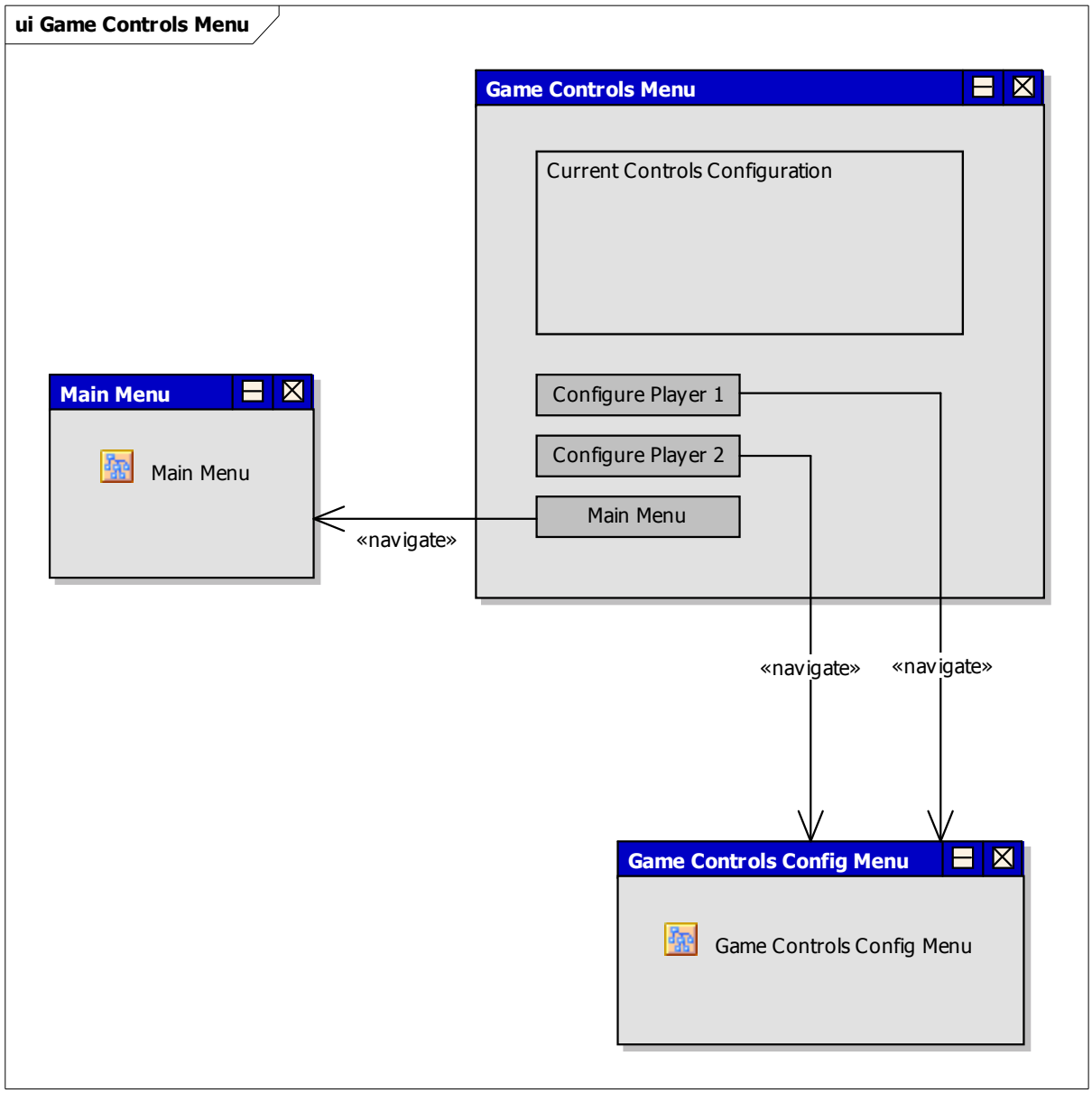
- Map Choice

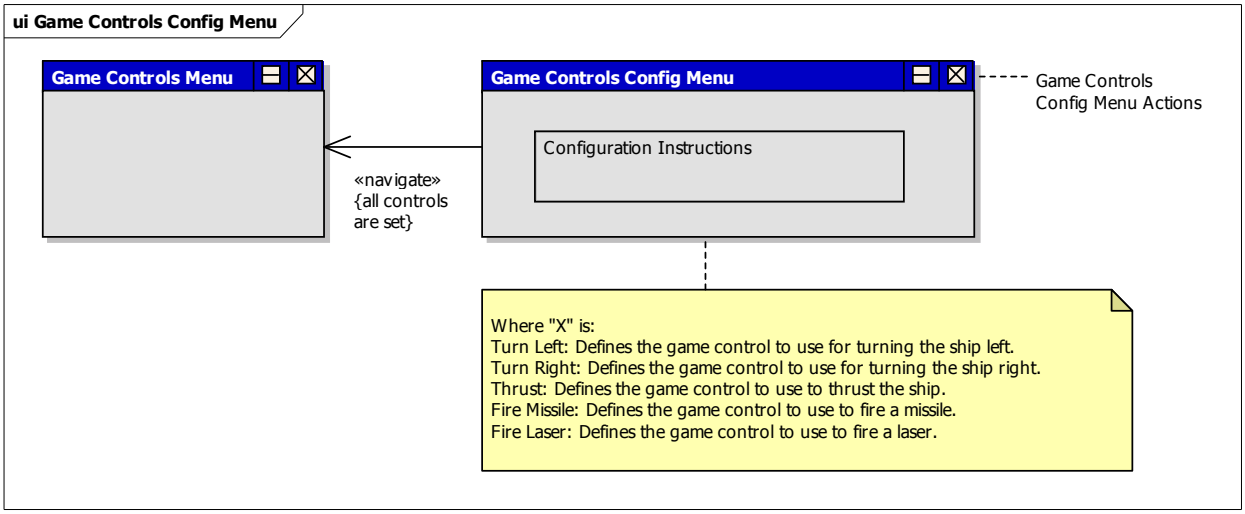


4.2.6 Controller Setup

Functional requirements

- Controls Configuration

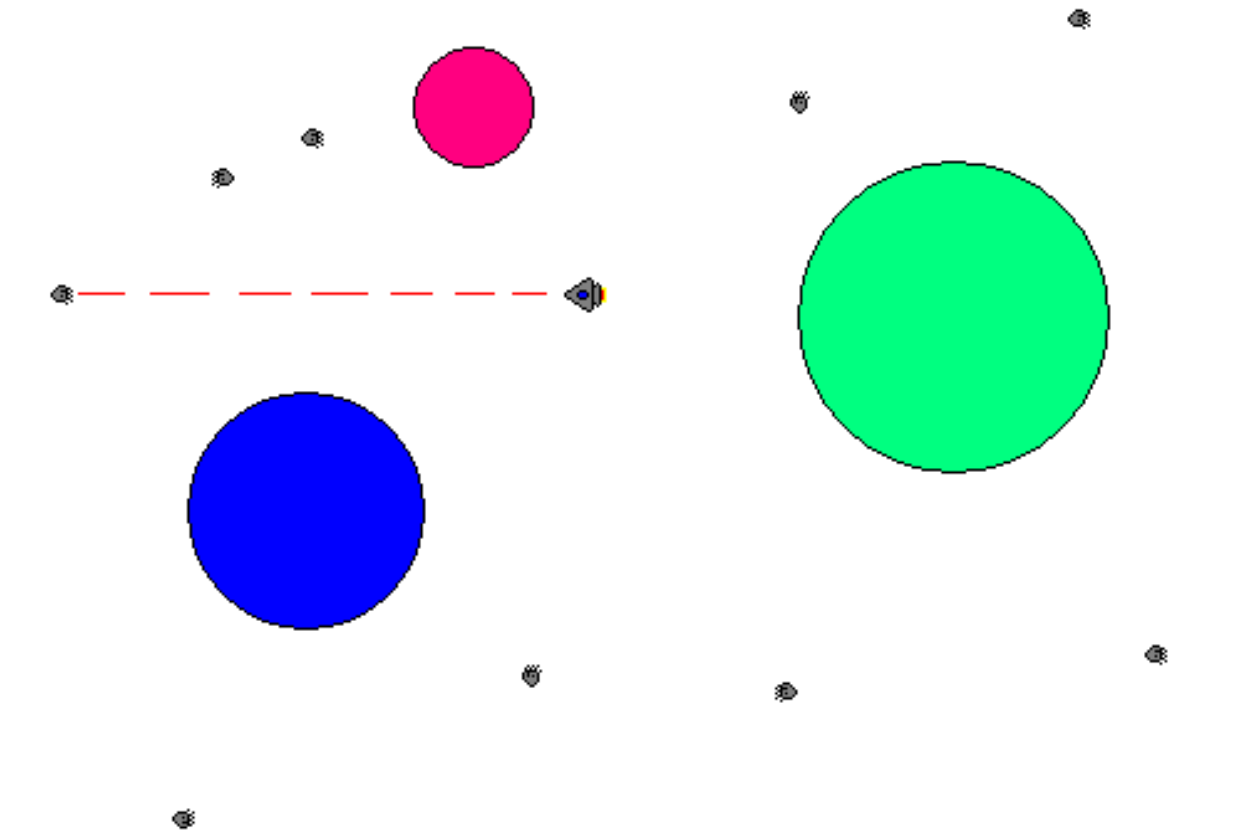

ui Game Controls Menu
Game Controls Menu
Current Controls Configuration
Configure Player 1
Configure Player 2
Main Menu
Main Menu
Main Menu
«navigate»
«navigate»
«navigate»
Game Controls Config Menu
Game Controls Config Menu



4.2.7 In-Game Single Player Screen

Functional requirements

- Fuel Restriction
- Planets
- Asteroids
- Items
- World Boundary Wrapping
- Players World View
- Game Play Info
- Single/Multi Player Scoring
- Ship Lives

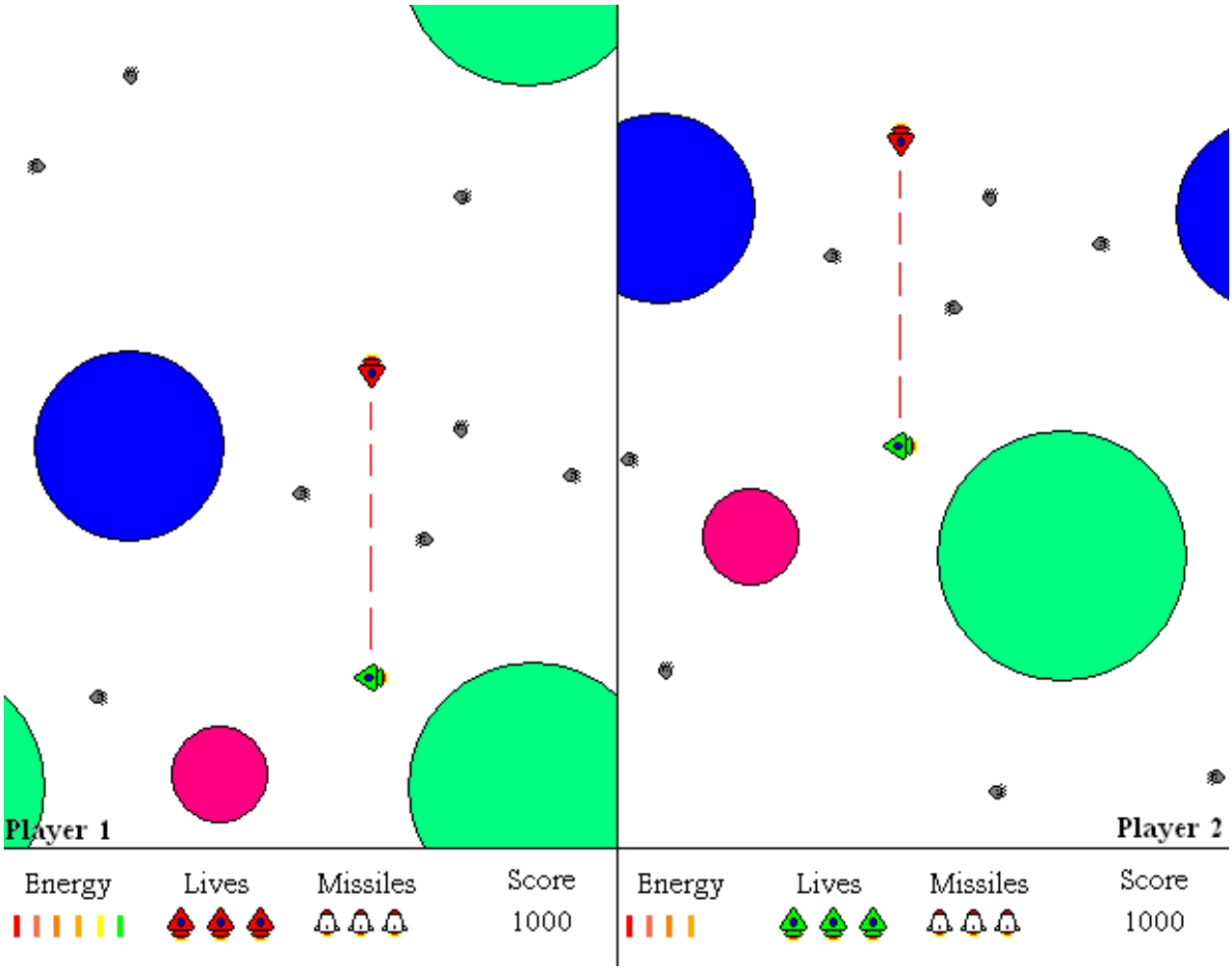


4.2.8 In-Game Multi Player Screen

Functional requirements

- Planets
- Asteroids
- Items
- World boundary wrapping
- Players world view
- Game play info
- Single/multi player scoring

- Ship lives



4.3 Names of controls, methods/procedures and triggers for each screen

4.3.1 Main Menu

***Triggered by*:**

a. Starting the game application;
b. A game session's ending (after viewing the “High Score” screen);
c. Return from any of the sub-menus;

Controls

- *Single Player*: Triggers a game state transition to the “map choice” menu.
- *Multi Player*: Triggers a game state transition to the “map choice” menu.
- *High Score*: Triggers a game state transition to the “high score” menu.
- *Game Controls*: Triggers a game state transition to the “game controls” menu.
- *Quit Game*: Triggers a game application exit/end.

4.3.2 Pause Menu

***Triggered by*:** Invoking the "Pause" control by pressing the appropriate key during a game session.

Controls

- *Resume Game*: The game session continues.
- *End Game*: The game session ends.

4.3.3 High Score Menu

***Triggered by*:**

a. Invoking the "High Score" control in the main menu;
b. A game session's ending;

Controls

- *Continue*:

a. *If this menu is triggered from the main menu*: Triggers a game state transition to the "main" menu;
b. *If this menu is triggered as a result of a finished game session*: Triggers a game state transition to the "main" menu;

4.3.4 Help Menu

***Triggered by*:**

a. Invoking the "Help" control in the "Main" menu;
b. Invoking the "Help" control by pressing the appropriate key during a game session;

Controls

- *Return*: Returns to the game state from which it was triggered.

4.3.5 Map Choice

***Triggered by*:**

a. Invoking the "Single Player" control in the "Main" menu;
b. Invoking the "Multi Player" control in the "Main" menu;

Fields

- *Map*: Defines the map that the game session is to be played on.

Controls

- *Start*: Initiates a new game session and performs a game state transition to that session.

4.3.6 Game Controls Menu

***Triggered by*:** Invoking the "Game Controls" control in the "Main" menu.

Fields

- *Turn Left*: Defines the game control to use for turning the ship left.
- *Turn Right*: Defines the game control to use for turning the ship right.
- *Thrust*: Defines the game control to use to thrust the ship.
- *Fire Missile*: Defines the game control to use to fire a missile.
- *Fire Laser*: Defines the game control to use to fire a laser.

Controls

- *Save*: Saves the set game controls and performs a game state transition to the "Main" menu.
- *Cancel*: Cancels any changes made to the game controls and performs a game state transition to the "Main" menu.

4.3.7 Single Player Game Screen

***Triggered by*:** Invoking the "Start Game" control in the "Map Choice" menu.

Controls

- *Pause*: Triggers a game state transition to the "Pause" menu.

4.3.8 Multi Player Game Screen

***Triggered by*:** Selecting start game from the multi player game rule choice screen.

Controls

- *Pause*: Triggers a game state transition to the "Pause" menu.

5. Design Details

5.1 Class Responsibility Collaborator (CRC) Cards

5.1.1 Audio

Responsible for playing sounds for the game. Does so by monitoring the Game module.

Audio::SoundManager

Type: *public* **Class**
Implements: *GameEventListener, Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Audio

Details: Created on 08-03-10 13:59:41. Modified on 08-03-10 14:00:30. Author: Lukas Kalinski

Responsible for monitoring the Game module and playing sounds according to its state.

Connections

- Access link to class *Game<Game>*
- Dependency link to component *FMOD<Infrastructure>*
- Realization link to interface *GameEventListener<Event>*
- Realization link to interface *Tickable<Util>*

5.1.2 Controller

Responsible for providing the Game module with an easy to use interface for registering different game controls, so that they react on user input.

Controller::GameControlManager

Type: *package* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Controller

Details: Created on 08-03-09 18:08:29. Modified on 08-03-09 18:29:27. Author: Lukas Kalinski

Responsible for managing a control's activation/deactivation according to changes registered by the input listener(s).

Connections

- Aggregation link to class *InputManager*
- Aggregation link from class *GameControl* <*Control*>
- Manipulate link from class *InputListener*
- Instantiate link from class *InputManager*

Controller::InputListener

Type: *package abstract* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Controller

Details: Created on 08-03-08 17:03:52. Modified on 08-03-09 18:29:33. Author: Lukas Kalinski

Abstract input listener.

Connections

- Aggregation link to class *InputManager*
- Manipulate link to class *GameControlManager*
- Generalization link from class *KeyboardListener*

Controller::InputManager

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Controller

Details: Created on 08-01-03 23:04:27. Modified on 08-03-08 23:24:38. Author: Lukas K

Responsible for detecting what game controls are being activated or deactivated, and calling the corresponding function on the contained game control objects.

Connections

- Aggregation link from class *InputListener*
- Aggregation link from class *GameControlManager*
- Use link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *SinglePlayerPlayState* <*State*>. Creates one for ordinary controls (i.e., playing the game) and one when game is over and the player is requested to "press <key> to continue".
- Instantiate link from class *TwoPlayersPlayState* <*State*>. Creates one for ordinary

controls (i.e., playing the game) and one when game is over and the player is requested to "press <key> to continue".

- Use link from class *TwoPlayersPlayState <State>*
- Instantiate link to class *GameControlManager*
- Use link from class *MenuState <State>*
- Instantiate link from class *MenuState <State>*

Controller::KeyboardListener

Type: *public* **Class**
Extends: *InputListener.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Controller

Details: Created on 08-03-08 21:09:59. Modified on 08-03-09 18:00:40. Author: Lukas Kalinski

Responsible for monitoring a defined set of keyboard keys and call InputListener::switchOn/switchOff functions when a key's status changes (pressed/unpressed).

Connections

- Use link to artifact *SDL*
- Access link to class *ConfigRegistry<Registry>*
- Use link from class *ConfigRegistry <Registry>*
- Generalization link to class *InputListener*

5.1.3 Game

Contains the whole game logic. Responsible for handling all different game states, such as menus, game play sessions, etc, as well as transitions between them. Further, the logic for each game state is also found in here, for example, the game world itself, with ships, etc.

Game::Game

Type: *public «singleton»* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Game

Details: Created on 08-01-04 07:44:45. Modified on 08-03-03 16:28:16. Author: Lukas K

Runs the main loop and forwards control to other game states by calling their tick() function on synchronized time intervals. If a game state turns invalid, it will be removed and a fallback to the previous state will be done.

Connections

- Aggregation link from class *GameState <State>*
- Aggregation link from class *GameState <State>*
- Access link from class *OpenGLRenderer <View>*
- Access link from class *SoundManager <Audio>*
- Use link from class *SinglePlayerPlayState <State>*
- Use link from class *LeaveStateAction <Menu>*
- Call link from class *PauseGameControl <Control>*
- Call link from class *EnterStateAction <Menu>*
- Call link to class *GameState<State>*
- Instantiate link to class *MainMenuState<State>*

5.1.4 Control

Control::GameControl

Type: *public abstract* **Class** {root}

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-29 14:20:20. Modified on 08-03-09 14:58:27. Author: Lukas Kalinski

Abstract game control class, whose leaf nodes represent a specific game control, used to manipulate the game state. Examples of possible controls are: "pause game", "steer ship left", etc.

Connections

- Aggregation link to class *GameControlManager<Controller>*
- Dependency link to requirement *All child controls must call their parent's activate/deactivate functions if they receive the corresponding call themselves.*
- Generalization link from class *ConfigKeyboardMapControl*
- Generalization link from class *FinishStateControl*
- Generalization link from class *MenuControl*
- Generalization link from class *GamePlayControl*

Control::MenuControl

Type: *public abstract* **Class**
Extends: *GameControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-01 19:53:53. Modified on 08-03-07 16:41:22. Author: Lukas Kalinski

Controls used in a menu.

Connections

- Aggregation link from class *Menu <Menu>*
- Generalization link from class *PressMenuButtonControl*
- Generalization link from class *NextMenuButtonControl*
- Generalization link from class *PrevMenuButtonControl*
- Generalization link to class *GameControl*

Control::PrevMenuButtonControl

Type: *public* **Class** {leaf}
Extends: *MenuControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-01 20:03:14. Modified on 08-03-08 14:38:53. Author: Lukas Kalinski

Responsible for setting a menu's button backward iteration on when activated, and off when deactivated.

Connections

- Manage link to class *Menu<Menu>*
- Instantiate link from class *MenuState <State>*
- Generalization link to class *MenuControl*

Control::NextMenuButtonControl

Type: *public* **Class** {leaf}
Extends: *MenuControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-01 20:04:07. Modified on 08-03-08 14:38:45. Author: Lukas Kalinski

Responsible for setting a menu's button forward iteration on when activated, and off when deactivated.

Connections

- Manage link to class *Menu<Menu>*
- Instantiate link from class *MenuState <State>*
- Generalization link to class *MenuControl*

Control::PressMenuButtonControl

Type: *public* **Class** {leaf}
Extends: *MenuControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-01 20:04:47. Modified on 08-03-08 14:38:29. Author: Lukas Kalinski

Responsible for pressing the menu's currently selected button.

Connections

- Manage link to class *Menu<Menu>*
- Instantiate link from class *MenuState <State>*
- Generalization link to class *MenuControl*

Control::GamePlayControl

Type: *public abstract* **Class**
Extends: *GameControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-29 16:59:08. Modified on 08-03-08 12:27:47. Author: Lukas Kalinski

Abstract class for controls that manipulate the game play state (e.g., ship steering).

Connections

- Generalization link to class *GameControl*
- Generalization link from class *PlayerControl*
- Generalization link from class *PauseGameControl*

Control::PauseGameControl

Type: *public* **Class** {leaf}
Extends: *GamePlayControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 14:35:51. Modified on 08-03-03 16:28:18. Author: Lukas Kalinski

Pause a game session.

Connections

- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>
- Call link to class *Game*<*Game*>
- Instantiate link to class *PauseMenuState*<*State*>
- Generalization link to class *GamePlayControl*

Control::PlayerControl

Type: *public abstract* **Class**
Extends: *GamePlayControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 15:14:36. Modified on 08-03-08 15:19:28. Author: Lukas Kalinski

Abstract class for controls that affect a player.

Connections

- Aggregation link from class *Player* <*Player*>
- Generalization link from class *ShipMissileFireControl*
- Generalization link from class *ShipLaserFireControl*
- Generalization link from class *ShipThrottleControl*
- Generalization link from class *ShipRightControl*
- Generalization link from class *ShipLeftControl*
- Generalization link to class *GamePlayControl*

Control::ShipThrottleControl

Type: *public* **Class** {leaf}
Extends: *PlayerControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 15:16:13. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Throttle a ship.

Connections

- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>. Create one for each player.
- Manipulate link to class *Ship*<*World*>
- Access link to class *Player*<*Player*>
- Generalization link to class *PlayerControl*

Control::ShipLeftControl

Type: *public* **Class** {leaf}
Extends: *PlayerControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 15:15:58. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Turn a ship left.

Connections

- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>. Create one for each player.
- Manipulate link to class *Ship*<*World*>
- Access link to class *Player*<*Player*>
- Generalization link to class *PlayerControl*

Control::ShipRightControl

Type: *public* **Class** {leaf}
Extends: *PlayerControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 15:16:05. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Turn a ship right.

Connections

- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>. Create one for each player.
- Manipulate link to class *Ship*<*World*>
- Access link to class *Player*<*Player*>
- Generalization link to class *PlayerControl*

Control::ShipLaserFireControl

Type: *public* **Class** {leaf}
Extends: *PlayerControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-11 15:16:21. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Fire a ship's laser gun.

Connections

- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>. Create one for each player.
- Manipulate link to class *Ship*<*World*>
- Access link to class *Player*<*Player*>
- Generalization link to class *PlayerControl*

Control::ShipMissileFireControl

Type: *public* **Class** {leaf}
Extends: *PlayerControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-02-27 00:34:56. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Fire a ship's missile launcher.

Connections

- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*. Create one for each player.
- Manipulate link to class *Ship<World>*
- Access link to class *Player<Player>*
- Generalization link to class *PlayerControl*

Control::FinishStateControl

Type: *public* **Class** {leaf}
Extends: *GameControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-08 14:27:12. Modified on 08-03-08 14:35:44. Author: Lukas Kalinski

Responsible for finishing the contained state when activated.

Connections

- Aggregation link from class *GameState <State>*
- Call link to class *GameState<State>*
- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Generalization link to class *GameControl*

Control::ConfigKeyboardMapControl

Type: *public* **Class** {leaf}
Extends: *GameControl.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Control

Details: Created on 08-03-08 15:17:03. Modified on 08-03-08 15:21:41. Author: Lukas Kalinski

Responsible for reconfiguring the mapping between a keyboard key and a game control.

Connections

- Generalization link to class *GameControl*

5.1.5 Engine

Contains manipulators and managers of the game world. All game events (such as a collision in the world) will be triggered from here.

Engine::Engine

Type: *public* **Class**
Implements: *Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Engine

Details: Created on 08-02-10 23:01:25. Modified on 08-03-05 17:30:59. Author: Lukas Kalinski

Responsible for maintaining the players and the game world they play within.

Connections

- Aggregation link from class *World* <*World*>
- Association link from class *PlayState* <*State*>
- Association link to class *Player*<*Player*>
- Call link from class *SinglePlayerPlayState* <*State*>
- Manage link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Call link from class *TwoPlayersPlayState* <*State*>
- Manage link from class *TwoPlayersPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>
- Manage link to class *World*<*World*>
- Realization link to interface *Tickable*<*Util*>

5.1.6 WorldEvent

Contains event representation classes as well as a world event manager and a related interface for world event listeners.

WorldEvent::CollisionEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 00:15:31. Modified on 08-03-10 13:08:13. Author: Lukas Kalinski

Represents an event of a collision between two objects in the game world and provides these objects. This event should be cascaded ONCE for each collision (i.e., we do not differentiate the objects participating in the collision here). Further, a collision event is expected to be cascaded regardless of whether one or both of the colliders were destroyed.

Connections

- Aggregation link from class *WorldObject* <*World*>
- Aggregation link from class *WorldObject* <*World*>
- Send link from class *CollisionStrategy* <*WorldPhysics*>
- Generalization link to class *WorldEvent*

WorldEvent::DamageEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 16:01:33. Modified on 08-03-10 13:08:02. Author: Lukas Kalinski

Represents a damage event and provides the destroyable object that was damaged (which isn't the same as being destroyed).

Connections

- Aggregation link from class *DestroyableObject* <*World*>

- Send link from class *CollisionStrategy* <*WorldPhysics*>
- Generalization link to class *WorldEvent*

WorldEvent::DestructionEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 16:01:41. Modified on 08-03-10 13:08:06. Author: Lukas Kalinski

Represents a destruction event and provides the destroyable object that was destroyed as well as the world object that caused its destruction (by, for example, colliding with it). A destruction event must not be treated as a RemovalOrderEvent, and vice versa, as a destruction doesn't necessarily mean that the object will be removed before next tick.

Connections

- Aggregation link from class *DestroyableObject* <*World*>
- Aggregation link from class *WorldObject* <*World*>
- Send link from class *CollisionStrategy* <*WorldPhysics*>
- Generalization link to class *WorldEvent*

WorldEvent::InsertionEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 14:56:11. Modified on 08-03-10 13:07:54. Author: Lukas Kalinski

Represents the event of a world object being inserted into the game world (i.e., not queued for insertion!).

Connections

- Aggregation link from class *WorldObject* <*World*>
- Generalization link to class *WorldEvent*

WorldEvent::ItemPickupEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-06 09:57:03. Modified on 08-03-10 13:08:17. Author: Lukas Kalinski

Represents the event of a ship picking up an item, and provides both the ship and the picked up item.

Connections

- Aggregation link from class *Ship <World>*
- Aggregation link from class *Item <World>*
- Send link from class *CollisionStrategy <WorldPhysics>*
- Generalization link to class *WorldEvent*

WorldEvent::ProjectileFireEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 12:32:10. Modified on 08-03-10 13:08:10. Author: Lukas Kalinski

Represents the event of firing a projectile and provides the projectile that was fired.

Connections

- Aggregation link from class *Projectile <World>*
- Instantiate link from class *Ship <World>*
- Generalization link to class *WorldEvent*

WorldEvent::RemovalOrderEvent

Type: *public «Message»* **Message**
Extends: *WorldEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-05 11:53:50. Modified on 08-03-10 13:07:58. Author: Lukas Kalinski

Represents the event of the world being ordered to remove a world object and provides that object. The removal order will be realized at the end of the world's current tick call. This event should be listened for by all classes that are keeping pointers to world objects, so that they know when to get rid of them.

Connections

- Aggregation link from class *WorldObject* <*World*>
- Generalization link to class *WorldEvent*

WorldEvent::WorldEvent

Type: *public «Message»* **Message**
Extends: *GameEvent.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldEvent

Details: Created on 08-03-10 13:07:24. Modified on 08-03-10 13:07:36. Author: Lukas Kalinski

Connections

- Generalization link to class *GameEvent*<*Event*>
- Generalization link from class *ItemPickupEvent*
- Generalization link from class *DestructionEvent*
- Generalization link from class *DamageEvent*
- Generalization link from class *InsertionEvent*
- Generalization link from class *ProjectileFireEvent*
- Generalization link from class *RemovalOrderEvent*
- Generalization link from class *CollisionEvent*

5.1.7 WorldLife

Contains world strategies for management of the world's life, i.e., inserting and removing world objects according to the rules defined by the strategies.

WorldLife::AsteroidStrategy

Type: *public* **Class**
Implements: *WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldLife

Details: Created on 08-02-28 15:30:12. Modified on 08-03-05 14:55:21. Author: Lukas Kalinski

Responsible for providing the world with asteroids, based on time-based constraints.

Connections

- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Access link to class *World<World>*. Reads the world's self-defined timestamp in order to calculate when an insertion should be made.
- Manage link to class *World<World>*
- Realization link to interface *WorldStrategy<World>*

WorldLife::ExpirationStrategy

Type: *public* **Class**
Implements: *GameEventListener, WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldLife

Details: Created on 08-03-05 11:29:59. Modified on 08-03-05 11:31:37. Author: Lukas Kalinski

Responsible for deciding what world objects should expire and when they should do so, resulting in being removed from the world. Examples are: projectiles, which shouldn't be in the world too long.

Connections

- Aggregation link from class *GameEventManager <Event>*
- Aggregation link from class *MissileItem <World>*
- Aggregation link from class *FuelItem <World>*
- Aggregation link from class *MissileProjectile <World>*
- Aggregation link from class *LaserProjectile <World>*
- Access link to class *World<World>*. Reads the world's self-defined timestamp in order to calculate expirations.
- Use link to class *World<World>*. When an object expires, its removal is queued in the world.
- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Realization link to interface *WorldStrategy<World>*

- Realization link to interface *GameEventListener<Event>*

WorldLife::ItemStrategy

Type: *public* **Class**
Implements: *WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldLife

Details: Created on 08-02-28 15:30:27. Modified on 08-03-05 14:07:46. Author: Lukas Kalinski

Responsible for providing the world with items, based on time-based constraints.

Connections

- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Manage link to class *World<World>*. Items may safely be inserted into the world, as they don't make a spawn point unavailable, as well as they can appear on a spawn point that is unavailable.
- Access link to class *World<World>*. Reads the world's self-defined timestamp in order to calculate when an insertion should be made.
- Realization link to interface *WorldStrategy<World>*

5.1.8 WorldPhysics

Contains managers of the world physics.

WorldPhysics::BoundaryStrategy

Type: *public abstract* **Class**
Implements: *WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldPhysics

Details: Created on 08-02-16 15:42:00. Modified on 08-03-05 17:21:38. Author: Lukas Kalinski

Responsible for making sure that each and every world object is within the world boundaries defined in this object, and if it's not, then it is repositioned according to the reposition() implementation in the concrete class.

Connections

- Aggregation link to class *World<World>*
- Access link to class *WorldObject<World>*
- Manage link to class *WorldObject<World>*. Repositions world object if found beyond world boundaries.
- Generalization link from class *RectangularBoundaryStrategy*
- Realization link to interface *WorldStrategy<World>*

WorldPhysics::CollisionStrategy

Type: *public* **Class**
Implements: *WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldPhysics

Details: Created on 08-02-11 00:08:58. Modified on 08-03-05 20:48:55. Author: Lukas Kalinski

Responsible for detecting and handling collisions between objects in the game world, provided a game world instance.

Connections

- Manage link to class *FuelItem<World>*. When picked up.
- Use link to class *World<World>*. On collisions leading to destruction, the concerned object(s) are queued for removal in the world.
- Send link to class *CollisionEvent<WorldEvent>*
- Send link to class *DamageEvent<WorldEvent>*
- Send link to class *DestructionEvent<WorldEvent>*
- Manage link to class *DestroyableObject<World>*
- Manage link to class *LaserProjectile<World>*
- Manage link to class *Asteroid<World>*
- Manage link to class *Ship<World>*
- Instantiate link from class *SinglePlayerPlayState <State>*
- Dependency link to class *MovableObject<World>*
- Dependency link to class *WorldObject<World>*
- Manage link to class *SpawnPoint<World>*. Toggles on availability on all spawn points before processing them.
- Dependency link to class *Item<World>*
- Dependency link to class *Planet<World>*
- Send link to class *ItemPickupEvent<WorldEvent>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Dependency link to class *Projectile<World>*
- Realization link to interface *WorldStrategy<World>*

WorldPhysics::GravityStrategy

Type: *public* **Class**
Implements: *WorldStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldPhysics

Details: Created on 08-02-11 00:12:08. Modified on 08-03-03 16:28:16. Author: Lukas Kalinski

Responsible for calculating and applying gravity affections for each gravity-affectable game world object.

Connections

- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Dependency link to class *WorldObject<World>*
- Manage link to class *MovableObject<World>*
- Access link to class *Planet<World>*
- Realization link to interface *WorldStrategy<World>*

WorldPhysics::RectangularBoundaryStrategy

Type: *public* **Class**
Extends: *BoundaryStrategy.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: WorldPhysics

Details: Created on 08-02-28 17:09:30. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

Represents the world boundaries, i.e., the area that world objects are allowed to appear on. Here, the world boundaries have a rectangular shape.

Connections

- Aggregation link from class *Coord2d <Util>*
- Aggregation link from class *Coord2d <Util>*
- Instantiate link from class *SinglePlayerPlayState <State>*
- Instantiate link from class *TwoPlayersPlayState <State>*
- Generalization link to class *BoundaryStrategy*

5.1.9 Event

Event::GameEvent

Type: *public «Message»* **Message**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Event

Details: Created on 08-03-05 11:59:43. Modified on 08-03-09 23:02:23. Author: Lukas Kalinski

Abstract world event class, bringing all events together under a common type.

Connections

- Use link from class *GameEventManager*
- Generalization link from class *WorldEvent* <*WorldEvent*>

Event::GameEventManager

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Event

Details: Created on 08-03-05 00:14:23. Modified on 08-03-09 23:02:30. Author: Lukas Kalinski

Responsible for cascading every event that is received to all registered observers (for example, the view and audio module).

Connections

- Aggregation link from interface *GameEventListener*
- Aggregation link to class *ExpirationStrategy*<*WorldLife*>
- Aggregation link to class *GameState*<*State*>
- Aggregation link to class *Ship*<*World*>
- Send link to interface *GameEventListener*
- Use link to class *GameEvent*
- Instantiate link from class *GameState* <*State*>

5.1.10 Menu

Menu::EnterStateAction

Type: *public* **Class**
Implements: *MenuAction.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Menu

Details: Created on 08-02-08 12:22:37. Modified on 08-03-03 16:28:15. Author: sfish

Responsible for entering a new state from the current one.

Connections

- Instantiate link from class *ControlsMenuState* <*State*>
- Instantiate link from class *MapChoiceMenuState* <*State*>. Instantiates one for each available map.
- Instantiate link from class *MainMenuState* <*State*>
- Instantiate link to class *GameState*<*State*>
- Call link to class *Game*<*Game*>
- Generalization link from class *EnterPlayingStateAction* <*State*>
- Realization link to interface *MenuAction*

Menu::LeaveStateAction

Type: *public* **Class**
Implements: *MenuAction.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Menu

Details: Created on 08-03-02 18:54:46. Modified on 08-03-03 16:44:27. Author: Lukas Kalinski

Responsible for leaving the current state to the previous one, or if none is available, to quit the game.

Connections

- Instantiate link from class *HelpMenuState* <*State*>
- Instantiate link from class *ControlsMenuState* <*State*>
- Instantiate link from class *HighScoreMenuState* <*State*>. The "back" button action.
- Instantiate link from class *MapChoiceMenuState* <*State*>. For the "back" button.

- Instantiate link from class *PauseMenuState <State>*
- Instantiate link from class *MainMenuState <State>*
- Use link to class *Game<Game>*
- Call link to class *GameState<State>*
- Generalization link from class *LeavePauseMenuStateAction <State>*
- Realization link to interface *MenuAction*

Menu::Menu

Type: *public* **Class**
Implements: *Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Menu

Details: Created on 08-02-08 12:19:32. Modified on 08-03-09 14:55:50. Author: sfish

Responsible for holding a set of buttons and providing functionality to "press" them as well as navigate in the button list.

Connections

- Aggregation link to class *MenuState<State>*
- Aggregation link to class *MenuControl<Control>*
- Aggregation link from class *MenuButton*
- Aggregation link from class *MenuButton*
- Instantiate link from class *HighScoreMenuState <State>*
- Instantiate link from class *MainMenuState <State>*
- Maintain link from class *MenuState <State>*
- Instantiate link from class *MapChoiceMenuState <State>*
- Manage link from class *PressMenuButtonControl <Control>*
- Instantiate link from class *ControlsMenuState <State>*
- Instantiate link from class *ControlsConfigMenuState <State>*
- Manage link from class *ControlsConfigMenuState <State>*
- Instantiate link from class *HelpMenuState <State>*
- Manage link from class *PrevMenuButtonControl <Control>*
- Manage link from class *NextMenuButtonControl <Control>*
- Instantiate link from class *PauseMenuState <State>*
- Realization link to interface *Tickable<Util>*

Menu::MenuButton

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Menu

Details: Created on 08-03-02 18:55:42. Modified on 08-03-09 01:09:47. Author: Lukas Kalinski

Responsible for holding one or more actions to take when pressed.

Connections

- Aggregation link to class *Menu*
- Aggregation link to class *Menu*
- Aggregation link from interface *MenuAction*
- Instantiate link from class *HelpMenuState* <*State*>. The "back" button.
- Instantiate link from class *ControlsMenuState* <*State*>. One for player 1 config, one for player 2 config and one for "back to main menu".
- Instantiate link from class *HighScoreMenuState* <*State*>. The "back" button.
- Instantiate link from class *MapChoiceMenuState* <*State*>. Instantiates one for each available map plus one for the "back" action.
- Instantiate link from class *PauseMenuState* <*State*>
- Instantiate link from class *MainMenuState* <*State*>

5.1.11 Player

Player::Player

Type: *public* **Class**
Implements: *GameEventListener.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Player

Details: Created on 08-02-11 15:48:29. Modified on 08-03-05 22:38:24. Author: Lukas Kalinski

Responsible for maintaining generic statistics about a player's activity in the game world, as well as for managing the player's ships.

Connections

- Aggregation link from class *Ship* <*World*>
- Aggregation link to class *PlayerControl*<*Control*>
- Association link to class *World*<*World*>. The world is fetched from engine during initialization.
- Association link to class *Ship*<*World*>
- Association link from class *Engine* <*Engine*>
- Instantiate link from class *SinglePlayerPlayState* <*State*>
- Instantiate link from class *TwoPlayersPlayState* <*State*>
- Access link from class *ShipMissileFireControl* <*Control*>
- Access link from class *ShipLaserFireControl* <*Control*>
- Access link from class *ShipThrottleControl* <*Control*>
- Access link from class *ShipRightControl* <*Control*>
- Access link from class *ShipLeftControl* <*Control*>
- Realization link to interface *GameEventListener*<*Event*>

5.1.12 State

State::GameState

Type: *public abstract* **Class** {root}
Implements: *Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-11 15:53:38. Modified on 08-03-10 12:55:31. Author: Lukas Kalinski

Represents an abstract state that the game may find itself in.

Connections

- Aggregation link from class *GameEventManager <Event>*
- Aggregation link to class *FinishStateControl<Control>*
- Aggregation link to class *Game<Game>*
- Aggregation link to class *Game<Game>*
- Call link from class *FinishStateControl <Control>*
- Call link from class *LeaveStateAction <Menu>*
- Instantiate link from class *EnterStateAction <Menu>*
- Instantiate link to class *GameEventManager<Event>*
- Call link from class *Game <Game>*
- Generalization link from class *PlayState*
- Generalization link from class *MenuState*
- Realization link to interface *Tickable<Util>*

State::MenuState

Type: *public abstract* **Class**
Extends: *GameState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-01-04 04:12:54. Modified on 08-03-03 16:28:18. Author: Lukas K

The state to be in when navigating through menus.

Connections

- Aggregation link from class *Menu <Menu>*
- Maintain link to class *Menu<Menu>*
- Instantiate link to class *PrevMenuButtonControl<Control>*
- Instantiate link to class *NextMenuButtonControl<Control>*
- Instantiate link to class *PressMenuButtonControl<Control>*
- Use link to package *Menu<Game>*
- Use link to class *InputManager<Controller>*
- Instantiate link to class *InputManager<Controller>*
- Generalization link from class *HelpMenuState*
- Generalization link from class *MainMenuState*
- Generalization link from class *PauseMenuState*
- Generalization link from class *MapChoiceMenuState*

- Generalization link from class *HighScoreMenuState*
- Generalization link from class *ControlsMenuState*
- Generalization link to class *GameState*
- Generalization link from class *ControlsConfigMenuState*

State::MainMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:05:11. Modified on 08-03-03 16:28:17. Author: Lukas Kalinski

Connections

- Instantiate link to class *HelpMenuState*
- Instantiate link to class *HighScoreMenuState*
- Instantiate link to class *ControlsMenuState*
- Instantiate link to class *MapChoiceMenuState*
- Instantiate link to class *EnterStateAction<Menu>*
- Instantiate link to class *Menu<Menu>*
- Instantiate link to class *MenuButton<Menu>*
- Instantiate link to class *LeaveStateAction<Menu>*
- Instantiate link from class *Game <Game>*
- Generalization link to class *MenuState*

State::MapChoiceMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:07:32. Modified on 08-03-03 21:08:54. Author: Lukas Kalinski

The map choice menu, containing one button for each available map.

Connections

- Aggregation link from class *World <World>*
- Use link to class *PlayState*. Bind the play state that was received on construction to all the map button actions.

- Instantiate link to class *EnterPlayingStateAction*. One enter playing state instance for each available map button.
- Friend link to class *EnterPlayingStateAction*
- Instantiate link to class *LeaveStateAction<Menu>*. For the "back" button.
- Instantiate link to class *MenuButton<Menu>*. Instantiates one for each available map plus one for the "back" action.
- Instantiate link to class *EnterStateAction<Menu>*. Instantiates one for each available map.
- Instantiate link to class *Menu<Menu>*
- Use link to class *WorldMapRegistry<Registry>*
- Access link to class *WorldMapRegistry<Registry>*
- Instantiate link from class *MainMenuState*
- Access link from class *SinglePlayerPlayState*
- Access link from class *TwoPlayersPlayState*
- Generalization link to class *MenuState*
- Inner class link from class *EnterPlayingStateAction*

State::MapChoiceMenuState::EnterPlayingStateAction

Type: *public* **Class**
Extends: *EnterStateAction.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-03-03 17:23:50. Modified on 08-03-03 20:17:04. Author: Lukas Kalinski

Connections

- Instantiate link from class *MapChoiceMenuState*. One enter playing state instance for each available map button.
- Friend link from class *MapChoiceMenuState*
- Generalization link to class *EnterStateAction<Menu>*
- Inner class link to class *MapChoiceMenuState*

State::ControlsMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:08:04. Modified on 08-03-03 20:56:15. Author: Lukas Kalinski

Connections

- Instantiate link to class *ControlsConfigMenuState*. One for each player (i.e., player 1 and 2).
- Instantiate link to class *MenuButton<Menu>*. One for player 1 config, one for player 2 config and one for "back to main menu".
- Instantiate link to class *LeaveStateAction<Menu>*
- Instantiate link to class *Menu<Menu>*
- Instantiate link to class *EnterStateAction<Menu>*
- Instantiate link from class *MainMenuState*
- Generalization link to class *MenuState*

State::ControlsConfigMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:08:16. Modified on 08-03-08 15:50:02. Author: Lukas Kalinski

The state of the controls configuration loop, i.e., where each control for a single player is reconfigured on a step-by-step basis.

Connections

- Trace link to requirement *Player 1 and Player 2 must have distinct controls*
- Manage link to class *Menu<Menu>*
- Instantiate link to class *Menu<Menu>*
- Instantiate link from class *ControlsMenuState*. One for each player (i.e., player 1 and 2).
- Generalization link to class *MenuState*

State::HighScoreMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:07:43. Modified on 08-03-03 16:28:16. Author: Lukas Kalinski

Connections

- Instantiate link to class *LeaveStateAction<Menu>*. The "back" button action.
- Instantiate link to class *MenuButton<Menu>*. The "back" button.
- Instantiate link to class *Menu<Menu>*
- Instantiate link from class *MainMenuState*
- Instantiate link from class *SinglePlayerPlayState*
- Dependency link from class *SinglePlayerPlayState*
- Generalization link to class *MenuState*

State::HelpMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-03-01 17:58:48. Modified on 08-03-03 16:28:16. Author: Lukas Kalinski

Connections

- Instantiate link to class *LeaveStateAction<Menu>*
- Instantiate link to class *MenuButton<Menu>*. The "back" button.
- Instantiate link to class *Menu<Menu>*
- Instantiate link from class *MainMenuState*
- Generalization link to class *MenuState*

State::PauseMenuState

Type: *public* **Class** {leaf}
Extends: *MenuState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-29 20:05:25. Modified on 08-03-03 16:28:18. Author: Lukas Kalinski

Connections

- Instantiate link to class *LeavePauseMenuStateAction*
- Instantiate link to class *LeaveStateAction<Menu>*

- Instantiate link to class *MenuButton<Menu>*
- Friend link to class *LeavePauseMenuStateAction*
- Instantiate link to class *Menu<Menu>*
- Access link from class *SinglePlayerPlayState*. Reads whether the pause menu state orders a game quit or not.
- Access link from class *TwoPlayersPlayState*
- Instantiate link from class *PauseGameControl <Control>*
- Generalization link to class *MenuState*
- Inner class link from class *LeavePauseMenuStateAction*

State::PauseMenuState::LeavePauseMenuStateAction

Type: *public* **Class**
Extends: *LeaveStateAction.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-03-03 16:17:06. Modified on 08-03-03 17:24:42. Author: Lukas Kalinski

Responsible for handling a press on the pause menu's "Quit Game" button.

Connections

- Instantiate link from class *PauseMenuState*
- Friend link from class *PauseMenuState*
- Generalization link to class *LeaveStateAction<Menu>*
- Inner class link to class *PauseMenuState*

State::PlayState

Type: *public abstract* **Class**
Extends: *GameState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-01-04 04:14:14. Modified on 08-03-05 23:08:40. Author: Lukas K

The game state to be in when a game session is active. Responsible for setting up and managing the game engine.

Connections

- Association link to class *Engine<Engine>*

- Use link from class *MapChoiceMenuState*. Bind the play state that was received on construction to all the map button actions.
- Use link to package *Engine<Game>*
- Generalization link from class *SinglePlayerPlayState*
- Generalization link from class *TwoPlayersPlayState*
- Generalization link to class *GameState*

State::SinglePlayerPlayState

Type: *public* **Class** {leaf}
Extends: *PlayState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-27 21:09:39. Modified on 08-03-06 14:01:28. Author: Lukas Kalinski

Responsible for starting a single player game and keeping it going until game over rules are met, resulting in entering the high score menu.

Connections

- Instantiate link to class *ShipMissileFireControl<Control>*
- Instantiate link to class *Engine<Engine>*
- Manage link to class *Engine<Engine>*
- Use link to class *HighScoreRegistry<Registry>*
- Access link to class *PauseMenuState*. Reads whether the pause menu state orders a game quit or not.
- Dependency link to class *HighScoreMenuState*
- Use link to class *Game<Game>*
- Instantiate link to class *Player<Player>*
- Call link to class *Engine<Engine>*
- Instantiate link to class *ShipThrottleControl<Control>*
- Instantiate link to class *HighScoreMenuState*
- Access link to class *MapChoiceMenuState*
- Instantiate link to class *ShipRightControl<Control>*
- Instantiate link to class *Ship<World>*
- Instantiate link to class *ShipLaserFireControl<Control>*
- Instantiate link to class *PauseGameControl<Control>*
- Instantiate link to class *FinishStateControl<Control>*
- Instantiate link to class *InputManager<Controller>*. Creates one for ordinary controls (i.e., playing the game) and one when game is over and the player is requested to "press <key> to continue".
- Use link to class *InputManager<Controller>*

- Instantiate link to class *AsteroidStrategy<WorldLife>*
- Instantiate link to class *ExpirationStrategy<WorldLife>*
- Instantiate link to class *ItemStrategy<WorldLife>*
- Instantiate link to class *RectangularBoundaryStrategy<WorldPhysics>*
- Instantiate link to class *CollisionStrategy<WorldPhysics>*
- Instantiate link to class *GravityStrategy<WorldPhysics>*
- Manage link to class *World<World>*
- Instantiate link to class *ShipLeftControl<Control>*
- Generalization link to class *PlayState*

State::TwoPlayersPlayState

Type: *public* **Class** {leaf}
Extends: *PlayState.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: State

Details: Created on 08-02-27 21:07:33. Modified on 08-03-06 14:00:59. Author: Lukas Kalinski

Responsible for starting a two player game and keeping it going until game over rules are met.

Connections

- Instantiate link to class *ShipLaserFireControl<Control>*. Create one for each player.
- Use link to class *InputManager<Controller>*
- Access link to class *MapChoiceMenuState*
- Access link to class *PauseMenuState*
- Instantiate link to class *Engine<Engine>*
- Manage link to class *Engine<Engine>*
- Call link to class *Engine<Engine>*
- Instantiate link to class *ShipThrottleControl<Control>*. Create one for each player.
- Instantiate link to class *ShipLeftControl<Control>*. Create one for each player.
- Instantiate link to class *Player<Player>*
- Instantiate link to class *ShipMissileFireControl<Control>*. Create one for each player.
- Manage link to class *World<World>*
- Instantiate link to class *PauseGameControl<Control>*
- Instantiate link to class *FinishStateControl<Control>*
- Instantiate link to class *InputManager<Controller>*. Creates one for ordinary controls (i.e., playing the game) and one when game is over and the player is requested to "press <key> to continue".
- Instantiate link to class *Ship<World>*. Use the same for both players.
- Instantiate link to class *AsteroidStrategy<WorldLife>*
- Instantiate link to class *ExpirationStrategy<WorldLife>*
- Instantiate link to class *ItemStrategy<WorldLife>*

- Instantiate link to class *RectangularBoundaryStrategy<WorldPhysics>*
- Instantiate link to class *CollisionStrategy<WorldPhysics>*
- Instantiate link to class *GravityStrategy<WorldPhysics>*
- Instantiate link to class *ShipRightControl<Control>*. Create one for each player.
- Generalization link to class *PlayState*

5.1.13 World

World::Asteroid

Type: *public* **Class**
Extends: *DestroyableObject, MovableObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:17:41. Modified on 08-03-03 16:28:15. Author: sfish

Representation of an asteroid flying around randomly in the game world.

Connections

- Aggregation link to class *World*
- Manage link from class *CollisionStrategy <WorldPhysics>*
- Generalization link to class *DestroyableObject*
- Generalization link to class *MovableObject*

World::DestroyableObject

Type: *public abstract* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-03-04 20:31:56. Modified on 08-03-04 20:33:43. Author: Lukas Kalinski

World objects implementing this interface are considered destroyable, i.e., it is possible to destroy them by causing enough damage to them. Note that destruction has nothing to do with removal from the world.

Connections

- Aggregation link to class *DestructionEvent<WorldEvent>*

- Aggregation link to class *DamageEvent<WorldEvent>*
- Manage link from class *CollisionStrategy <WorldPhysics>*
- Generalization link from class *MissileProjectile*
- Generalization link from class *Ship*
- Generalization link from class *Asteroid*

World::FuelItem

Type: *public* **Class**
Extends: *Item.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:21:03. Modified on 08-03-03 16:28:16. Author: sfish

Representation of an item that contains a fuel powerup.

Connections

- Aggregation link to class *ExpirationStrategy<WorldLife>*
- Manage link from class *CollisionStrategy <WorldPhysics>*. When picked up.
- Generalization link to class *Item*

World::Item

Type: *public abstract* **Class**
Extends: *StaticObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:17:51. Modified on 08-03-04 21:17:01. Author: sfish

Contains common behavior and properties of items occuring in the game world.

Connections

- Aggregation link to class *ItemPickupEvent<WorldEvent>*
- Aggregation link to class *World*
- Dependency link from class *CollisionStrategy <WorldPhysics>*
- Generalization link from class *MissileItem*
- Generalization link from class *FuelItem*
- Generalization link to class *StaticObject*

World::LaserProjectile

Type: *public* **Class**
Extends: *Projectile.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:18:45. Modified on 08-03-04 14:16:21. Author: sfish

Representation of a laser projectile, which is not affected by gravities.

Connections

- Aggregation link to class *ExpirationStrategy<WorldLife>*
- Manage link from class *CollisionStrategy <WorldPhysics>*
- Instantiate link from class *Ship*. Uses internal world instance to do so.
- Generalization link to class *Projectile*

World::MissileItem

Type: *public* **Class**
Extends: *Item.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:21:31. Modified on 08-03-03 16:28:18. Author: sfish

Representation of an item that contains ship missiles.

Connections

- Aggregation link to class *ExpirationStrategy<WorldLife>*
- Generalization link to class *Item*

World::MissileProjectile

Type: *public* **Class**
Extends: *DestroyableObject, Projectile.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:18:50. Modified on 08-03-03 16:28:18. Author: sfish

Representation of a gravity-affectable missile projectile.

Connections

- Aggregation link to class *ExpirationStrategy<WorldLife>*
- Instantiate link from class *Ship*. Uses internal world instance to do so.
- Generalization link to class *Projectile*
- Generalization link to class *DestroyableObject*
- Realization link to requirement *Affected movement affects orientation too*

World::MovableObject

Type: *public abstract* **Class**
Extends: *WorldObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:17:29. Modified on 08-03-03 16:28:18. Author: sfish

Contains common behavior and properties of movable game world objects.

Connections

- Aggregation link from class *Vector2d <Util>*
- Aggregation link to class *World*
- Manage link from class *GravityStrategy <WorldPhysics>*
- Dependency link from class *CollisionStrategy <WorldPhysics>*
- Generalization link from class *Projectile*
- Generalization link from class *Ship*
- Generalization link from class *Asteroid*
- Generalization link to class *WorldObject*

World::Planet

Type: *public* **Class**
Extends: *StaticObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:17:45. Modified on 08-03-03 16:28:18. Author: sfish

Representation of a planet.

Connections

- Access link from class *GravityStrategy <WorldPhysics>*
- Dependency link from class *CollisionStrategy <WorldPhysics>*
- Generalization link to class *StaticObject*

World::Projectile

Type: *public abstract* **Class**
Extends: *MovableObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:18:12. Modified on 08-03-03 16:28:19. Author: sfish

Contains common behavior and properties of weapon projectiles fired by a ship.

Connections

- Aggregation link to class *ProjectileFireEvent<WorldEvent>*
- Aggregation link from class *Ship*
- Dependency link from class *CollisionStrategy <WorldPhysics>*
- Generalization link from class *MissileProjectile*
- Generalization link from class *LaserProjectile*
- Generalization link to class *MovableObject*

World::Ship

Type: *public* **Class**
Extends: *DestroyableObject, MovableObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-08 11:18:01. Modified on 08-03-03 16:28:19. Author: sfish

Representation of the ship that a game player will control.

Connections

- Aggregation link to class *ItemPickupEvent<WorldEvent>*
- Aggregation link to class *Player<Player>*

- Aggregation link to class *Projectile*
- Aggregation link to class *World*
- Aggregation link from class *GameEventManager <Event>*
- Association link from class *Player <Player>*
- Instantiate link to class *LaserProjectile*. Uses internal world instance to do so.
- Instantiate link to class *MissileProjectile*. Uses internal world instance to do so.
- Instantiate link to class *ProjectileFireEvent<WorldEvent>*
- Dependency link from artifact *Status Bar Renderer(s)*
- Dependency link from artifact *Status Bar Renderer(s)*
- Dependency link from artifact *Status Bar Renderer(s)*
- Manipulate link from class *ShipLeftControl <Control>*
- Manipulate link from class *ShipRightControl <Control>*
- Manipulate link from class *ShipThrottleControl <Control>*
- Manipulate link from class *ShipLaserFireControl <Control>*
- Manipulate link from class *ShipMissileFireControl <Control>*
- Instantiate link from class *TwoPlayersPlayState <State>*. Use the same for both players.
- Instantiate link from class *SinglePlayerPlayState <State>*
- Manage link from class *CollisionStrategy <WorldPhysics>*
- Generalization link to class *DestroyableObject*
- Generalization link to class *MovableObject*
- Realization link to requirement *Affected movement affects orientation too*

World::SpawnPoint

Type: *public* **Class**
Extends: *StaticObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-28 14:51:57. Modified on 08-03-04 21:23:14. Author: Lukas Kalinski

Defines a point in which a world object may appear, telling whether the area is free of obstacles or not.

Connections

- Aggregation link to class *World*
- Manage link from class *CollisionStrategy <WorldPhysics>*. Toggles on availability on all spawn points before processing them.
- Generalization link to class *StaticObject*

World::StaticObject

Type: *public abstract* **Class**
Extends: *WorldObject.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-11 21:50:57. Modified on 08-03-03 16:28:20. Author: Lukas Kalinski

Contains common behavior and properties of static game world objects.

Connections

- Generalization link from class *SpawnPoint*
- Generalization link to class *WorldObject*
- Generalization link from class *Item*
- Generalization link from class *Planet*

World::World

Type: *public* **Class**
Implements: *GameEventListener, Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-10 22:05:16. Modified on 08-03-03 16:28:20. Author: Lukas Kalinski

Representation of the whole game world, containing all world objects that are supposed to exist at a certain moment during a game session.

Connections

- Aggregation link from class *WorldObject*
- Aggregation link to class *MapChoiceMenuState<State>*
- Aggregation link from class *MovableObject*
- Aggregation link from class *Asteroid*
- Aggregation link from class *Item*
- Aggregation link to class *Engine<Engine>*
- Aggregation link from class *Ship*
- Aggregation link from class *WorldObject*
- Aggregation link from class *BoundaryStrategy <WorldPhysics>*
- Aggregation link from interface *WorldStrategy*. Interface for a world strategy. A

world strategy is something that modifies/alters the world and/or its objects.
- Aggregation link from class *SpawnPoint*
- Association link from class *Player* <*Player*>. The world is fetched from engine during initialization.
- Use link from class *ExpirationStrategy* <*WorldLife*>. When an object expires, its removal is queued in the world.
- Call link to class *WorldObject*
- Manage link from class *Engine* <*Engine*>
- Instantiate link from class *WorldMapRegistry* <*Registry*>
- Use link from class *CollisionStrategy* <*WorldPhysics*>. On collisions leading to destruction, the concerned object(s) are queued for removal in the world.
- Manage link from class *TwoPlayersPlayState* <*State*>
- Manage link from class *ItemStrategy* <*WorldLife*>. Items may safely be inserted into the world, as they don't make a spawn point unavailable, as well as they can appear on a spawn point that is unavailable.
- Access link from class *ItemStrategy* <*WorldLife*>. Reads the world's self-defined timestamp in order to calculate when an insertion should be made.
- Access link from class *ExpirationStrategy* <*WorldLife*>. Reads the world's self-defined timestamp in order to calculate expirations.
- Access link from class *AsteroidStrategy* <*WorldLife*>. Reads the world's self-defined timestamp in order to calculate when an insertion should be made.
- Manage link from class *AsteroidStrategy* <*WorldLife*>
- Manage link from class *SinglePlayerPlayState* <*State*>
- Realization link to interface *GameEventListener*<*Event*>
- Realization link to interface *Tickable*<*Util*>

World::WorldObject

Type: *public abstract* **Class**
Implements: *Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: World

Details: Created on 08-02-11 21:49:28. Modified on 08-03-03 16:28:20. Author: Lukas Kalinski

Contains common behavior and properties of game world objects.

Connections

- Aggregation link to class *CollisionEvent*<*WorldEvent*>
- Aggregation link from class *Vector2d* <*Util*>
- Aggregation link to class *World*
- Aggregation link to class *World*
- Aggregation link from class *Shape* <*Util*>
- Aggregation link from class *Coord2d* <*Util*>

- Aggregation link to class *CollisionEvent<WorldEvent>*
- Aggregation link to class *InsertionEvent<WorldEvent>*
- Aggregation link to class *RemovalOrderEvent<WorldEvent>*
- Aggregation link to class *DestructionEvent<WorldEvent>*
- Call link from class *World*
- Manage link from class *BoundaryStrategy <WorldPhysics>*. Repositions world object if found beyond world boundaries.
- Access link from class *BoundaryStrategy <WorldPhysics>*
- Dependency link from class *GravityStrategy <WorldPhysics>*
- Dependency link from class *CollisionStrategy <WorldPhysics>*
- Generalization link from class *StaticObject*
- Generalization link from class *MovableObject*
- Realization link to requirement *World objects shall not do their own cleanup.*
- Realization link to interface *Tickable<Util>*

5.1.14 Registry

Responsible for holding, managing persistence for and providing global data to other modules.

Registry::ConfigRegistry

Type: *public «singleton»* **Class**
Extends: *Registry.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Registry

Details: Created on 08-02-10 23:12:56. Modified on 08-03-09 19:07:25. Author: Lukas Kalinski

Responsible to provide and handle persistence for the game configuration.

Connections

- Access link from class *KeyboardListener <Controller>*
- Use link to class *KeyboardListener<Controller>*
- Generalization link to class *Registry*

Registry::HighScoreRegistry

Type: *public «singleton»* **Class**
Extends: *Registry.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Registry

Details: Created on 08-02-11 14:20:22. Modified on 08-03-09 18:55:44. Author: Lukas Kalinski

Responsible for storing high scores in a file, as well as deciding what scores should be considered being high scores.

Connections

- Use link from class *SinglePlayerPlayState <State>*
- Generalization link to class *Registry*

Registry::Registry

Type: *public abstract* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Registry

Details: Created on 08-03-09 18:57:20. Modified on 08-03-09 18:58:13. Author: Lukas Kalinski

Abstract registry class, holding file management functions.

Connections

- Generalization link from class *HighScoreRegistry*
- Generalization link from class *WorldMapRegistry*
- Generalization link from class *ConfigRegistry*

Registry::WorldMapRegistry

Type: *public «singleton»* **Class**
Extends: *Registry.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: Registry

Details: Created on 08-02-11 22:34:50. Modified on 08-03-06 11:23:10. Author: Lukas Kalinski

Responsible for generating game worlds while being provided a map name.

Connections

- Use link from class *MapChoiceMenuState <State>*
- Access link from class *MapChoiceMenuState <State>*
- Instantiate link to class *World<World>*

- Generalization link to class *Registry*

5.1.15 Util

Contains common utilities, such as coordinate representations etc.

Util::CircularShape

Type: *public* **Class**
Extends: *Shape*.

Status: Proposed. Version 1.0. Phase 1.0.

Package: Util

Details: Created on 08-03-02 16:12:06. Modified on 08-03-03 16:28:15. Author: Lukas Kalinski

A circular shape.

Connections

- Generalization link to class *Shape*

Util::Coord2d

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Util

Details: Created on 08-02-27 21:46:56. Modified on 08-03-03 16:28:15. Author: Lukas Kalinski

Represents a coordinate in the absolute 2D-space.

Connections

- Aggregation link to class *WorldObject<World>*
- Aggregation link to class *RectangularBoundaryStrategy<WorldPhysics>*
- Aggregation link to class *RectangularBoundaryStrategy<WorldPhysics>*

Util::Shape

Type: *public abstract* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Util

Details: Created on 08-03-02 16:11:34. Modified on 08-03-03 16:28:19. Author: Lukas Kalinski

An abstrac geometric shape.

Connections

- Aggregation link to class *WorldObject<World>*
- Generalization link from class *CircularShape*

Util::Vector2d

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: Util

Details: Created on 08-02-11 21:34:04. Modified on 08-03-03 16:28:20. Author: Lukas Kalinski

Defines a vector in 2D space by combining a coordinate and a length.

Connections

- Aggregation link to class *WorldObject<World>*
- Aggregation link to class *MovableObject<World>*

5.1.16 View

Responsible for drawing graphics for the game. Does so by monitoring the Game module and associating elements in it with own graphical objects, which then will be painted.

AnimationSprite

Type: *public* **Class**
Extends: *Sprite.* Implements: *Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 15:19:07. Modified on 08-03-06 15:19:11. Author: Lukas Kalinski

Connections

- Generalization link to class *Sprite*
- Realization link to interface *Tickable<Util>*

AsteroidSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:39:01. Modified on 08-03-10 14:31:22. Author: Lukas Kalinski

Responsible for managing sprites for an asteroid.

Connections

- Generalization link to class *SpriteManager*

FuelItemSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:46:26. Modified on 08-03-10 14:31:33. Author: Lukas Kalinski

Responsible for managing sprites for a fuel item.

Connections

- Generalization link to class *SpriteManager*

LaserSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:46:09. Modified on 08-03-10 14:31:43. Author: Lukas Kalinski

Responsible for managing sprites for a laser.

Connections

- Generalization link to class *SpriteManager*

MenuButtonSpriteManager

Type: *public* **Class**
Extends: *TextSpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 15:20:03. Modified on 08-03-06 15:20:11. Author: Lukas Kalinski

Connections

- Generalization link to class *TextSpriteManager*

MissileItemSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:49:03. Modified on 08-03-10 14:31:59. Author: Lukas Kalinski

Responsible for managing sprites for a missile item.

Connections

- Generalization link to class *SpriteManager*

MissileSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:46:19. Modified on 08-03-10 14:31:38. Author: Lukas Kalinski

Responsible for managing sprites for a missile.

Connections

- Generalization link to class *SpriteManager*

OpenGLRenderer

Type: *public* **Class**
Implements: *Renderer, Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 13:25:08. Modified on 08-03-06 13:25:46. Author: Lukas Kalinski

Responsible for rendering graphics by using OpenGL.

Connections

- Aggregation link from class *SpriteManager*
- Aggregation link from class *SpriteManager*
- Access link to class *Game<Game>*
- Realization link to interface *Tickable<Util>*
- Realization link to interface *Renderer*

PlanetSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:49:35. Modified on 08-03-10 14:32:07. Author: Lukas Kalinski

Responsible for managing sprites for a planet.

Connections

- Generalization link to class *SpriteManager*

ShipSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 13:42:27. Modified on 08-03-10 14:31:15. Author: Lukas Kalinski

Responsible for managing sprites for a ship.

Connections

- Generalization link to class *SpriteManager*

SpawnPointSpriteManager

Type: *public* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 23:50:01. Modified on 08-03-10 14:32:12. Author: Lukas Kalinski

Responsible for managing sprites for a spawn point.

Connections

- Generalization link to class *SpriteManager*

Sprite

Type: *public* **Class**

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 15:21:40. Modified on 08-03-06 15:21:45. Author: Lukas Kalinski

Connections

- Aggregation link to class *SpriteManager*
- Aggregation link to class *SpriteManager*
- Aggregation link to class *SpriteManager*
- Generalization link from class *AnimationSprite*

SpriteManager

Type: *public abstract* **Class**
Implements: *GameEventListener, Tickable.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-09 21:24:51. Modified on 08-03-10 14:33:11. Author: Lukas Kalinski

Abstract sprite manager.

Connections

- Aggregation link from class *Sprite*
- Aggregation link to class *OpenGLRenderer*
- Aggregation link to class *OpenGLRenderer*
- Aggregation link from class *Sprite*
- Aggregation link from class *Sprite*
- Generalization link from class *LaserSpriteManager*
- Generalization link from class *MissileSpriteManager*
- Generalization link from class *FuelItemSpriteManager*
- Generalization link from class *MissileItemSpriteManager*
- Generalization link from class *PlanetSpriteManager*
- Generalization link from class *ShipSpriteManager*
- Generalization link from class *TextSpriteManager*
- Generalization link from class *AsteroidSpriteManager*
- Generalization link from class *SpawnPointSpriteManager*
- Realization link to interface *GameEventListener<Event>*
- Realization link to interface *Tickable<Util>*

TextSpriteManager

Type: *public abstract* **Class**
Extends: *SpriteManager.*

Status: Proposed. Version 1.0. Phase 1.0.

Package: View

Details: Created on 08-03-06 15:20:33. Modified on 08-03-10 14:33:03. Author: Lukas Kalinski

Abstract sprite manager that will manage texts.

Connections

- Generalization link to class *SpriteManager*
- Generalization link from class *MenuButtonSpriteManager*

5.2 Class Diagram

Diagram: Controller

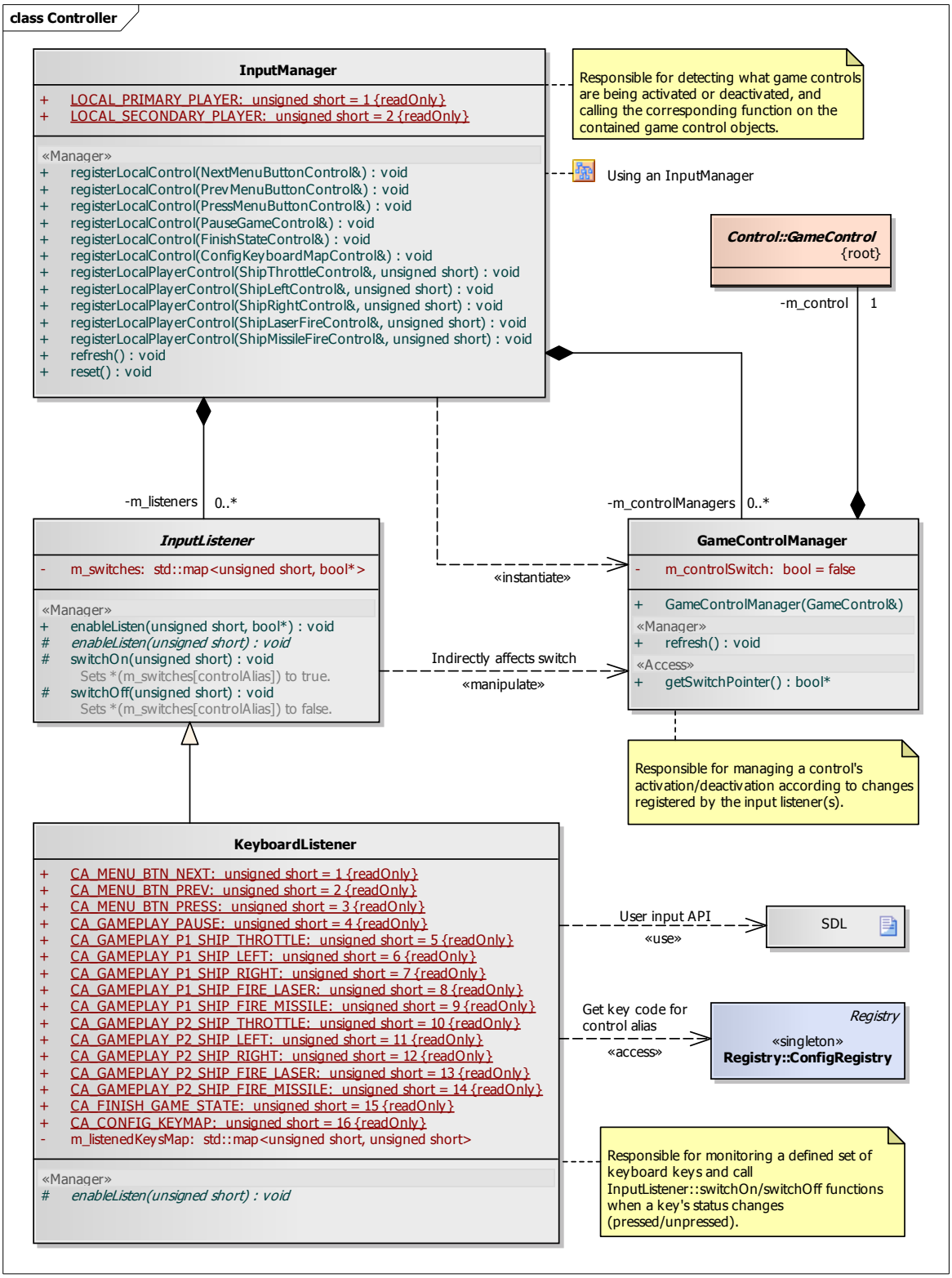


Diagram: Game

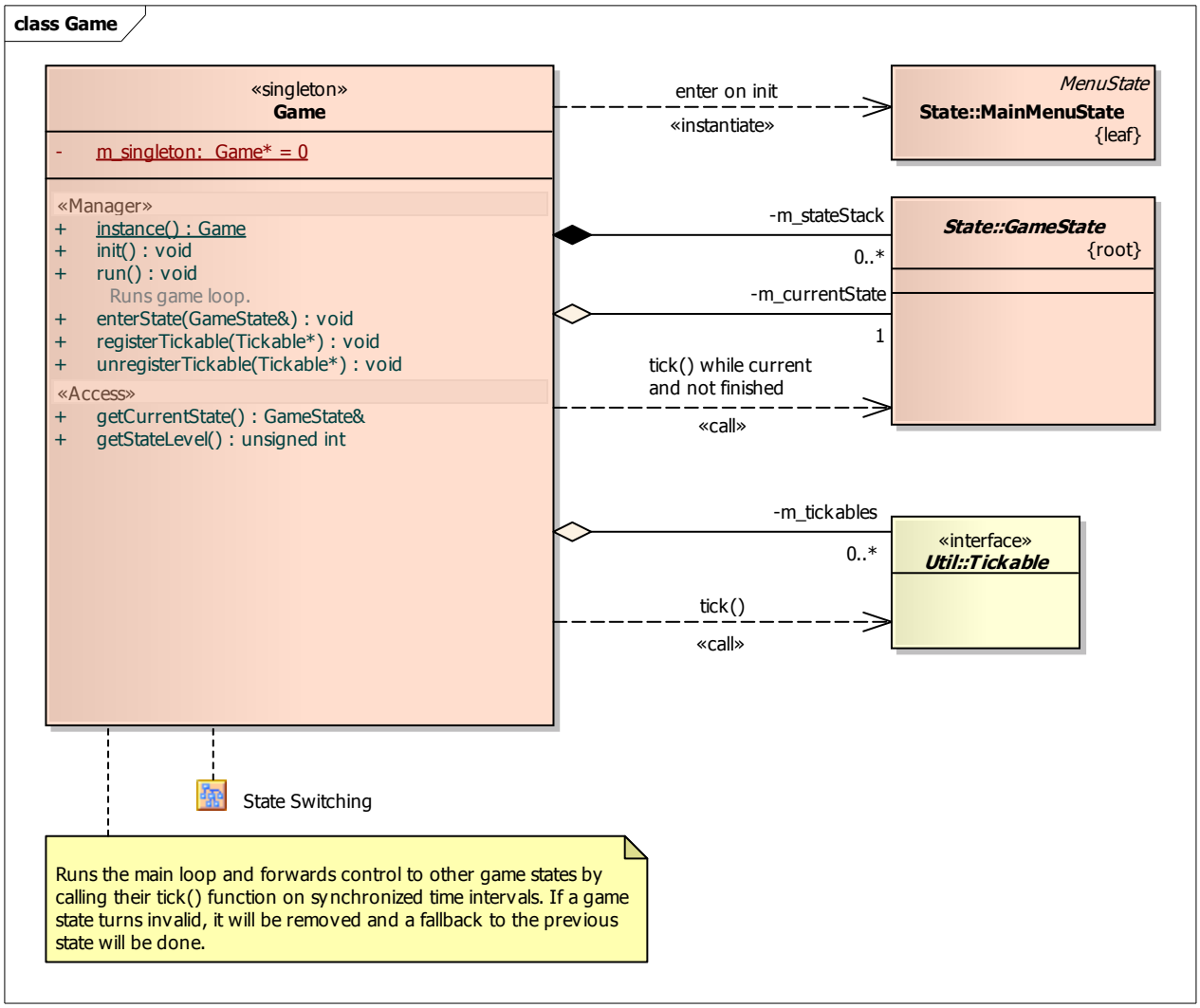


Diagram: Game Engine

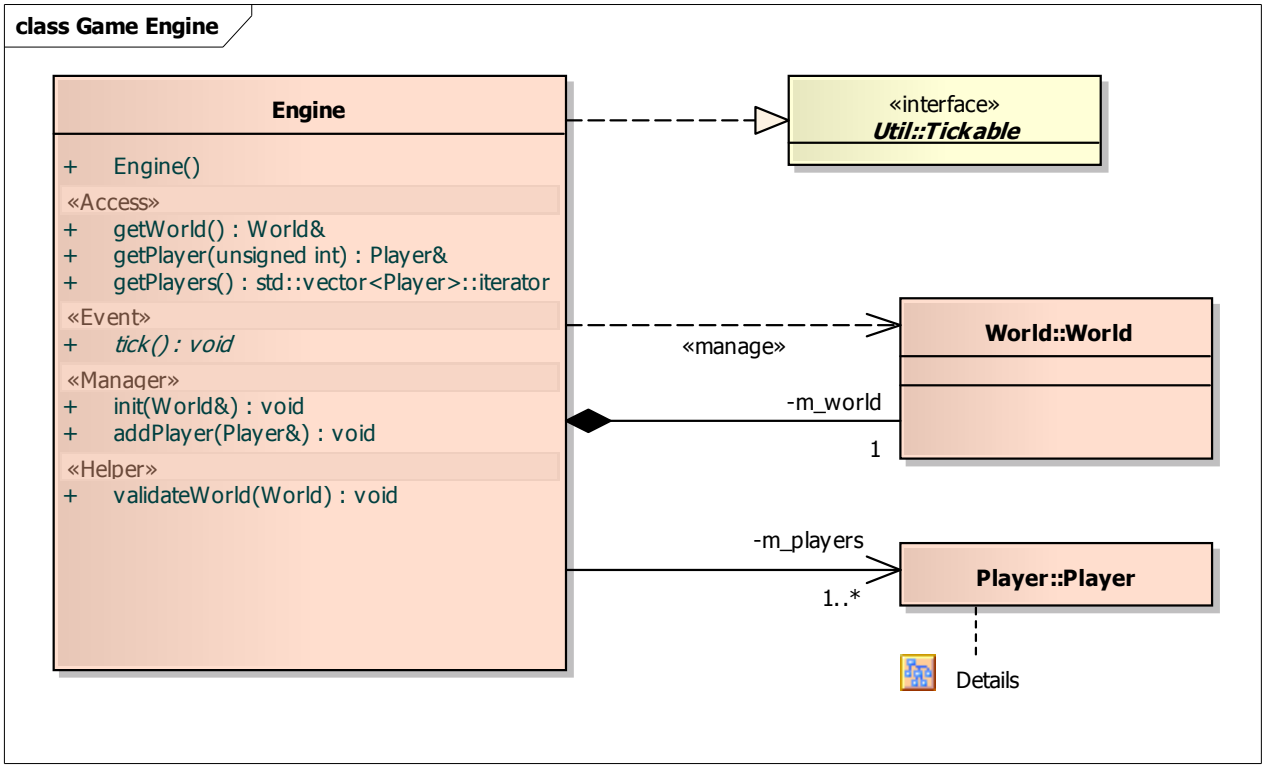


Diagram: Collision Strategy

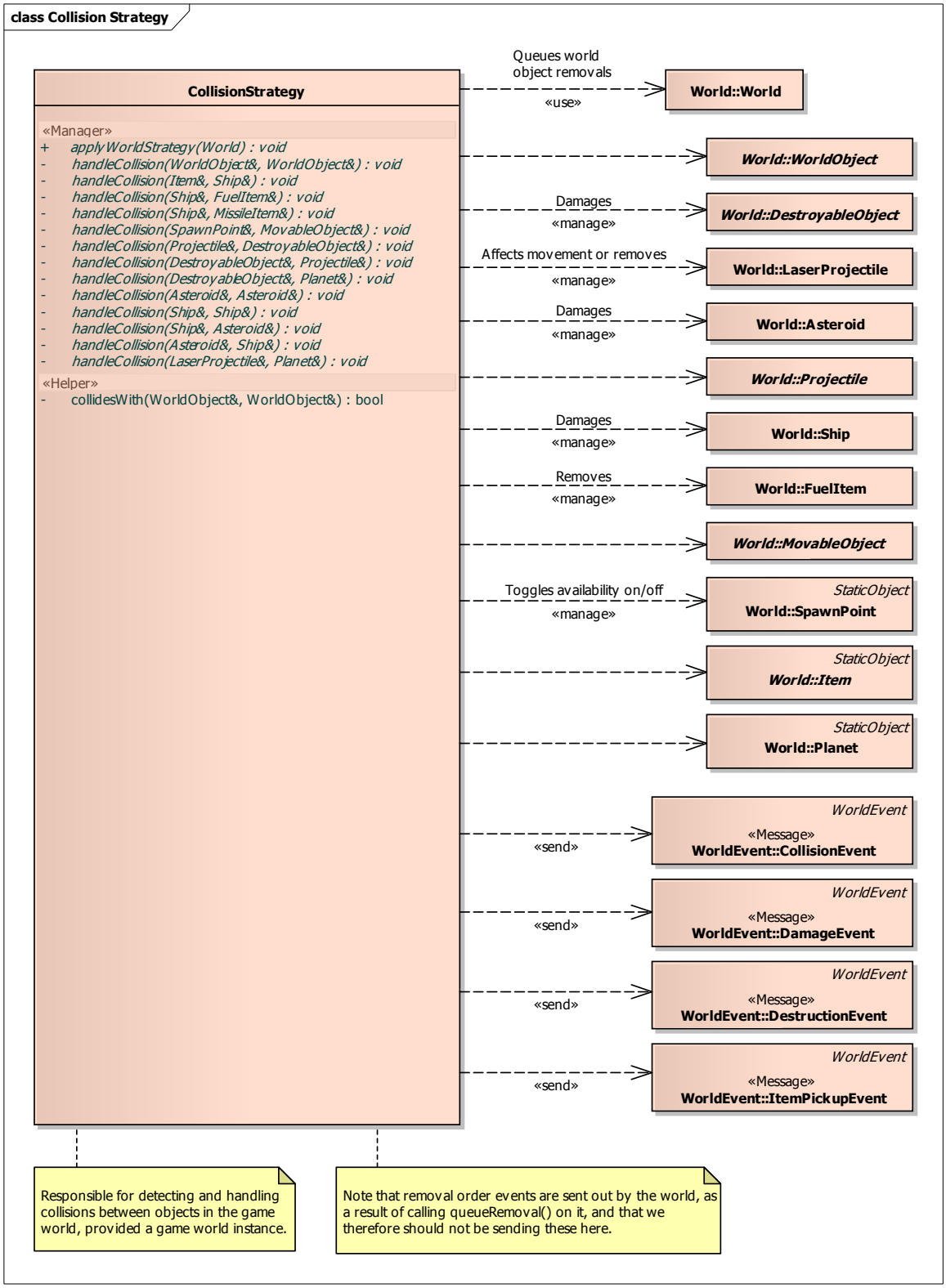


Diagram: GravityStrategy

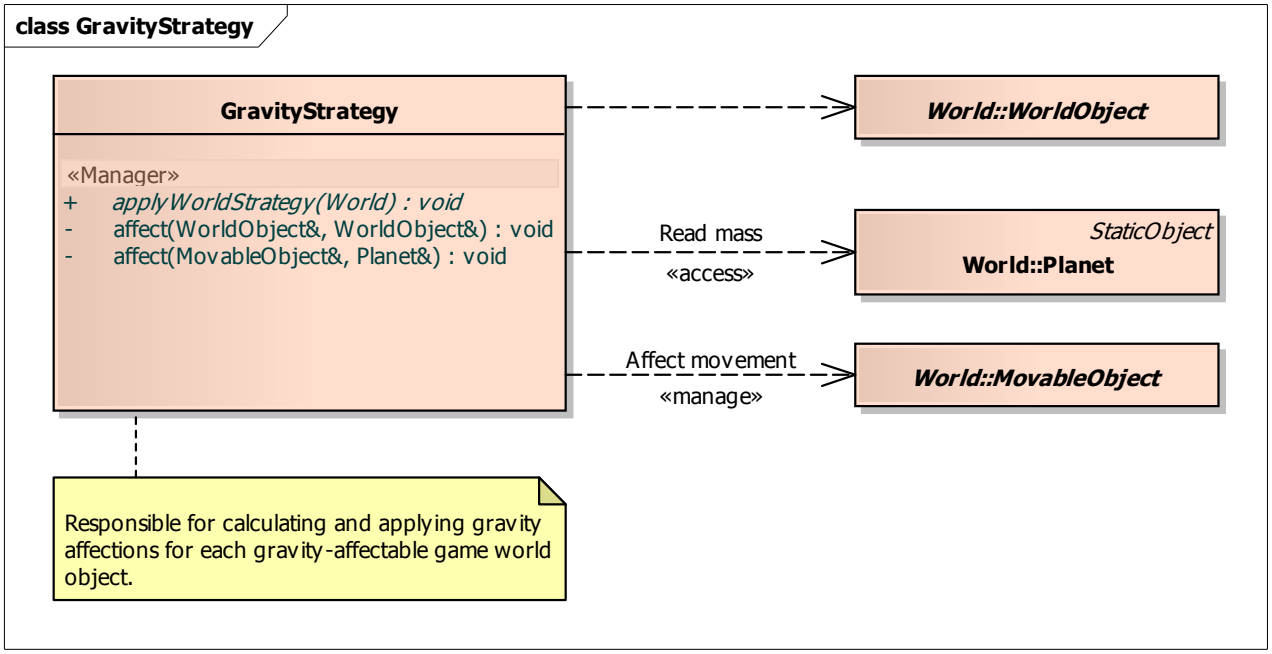


Diagram: World Physics

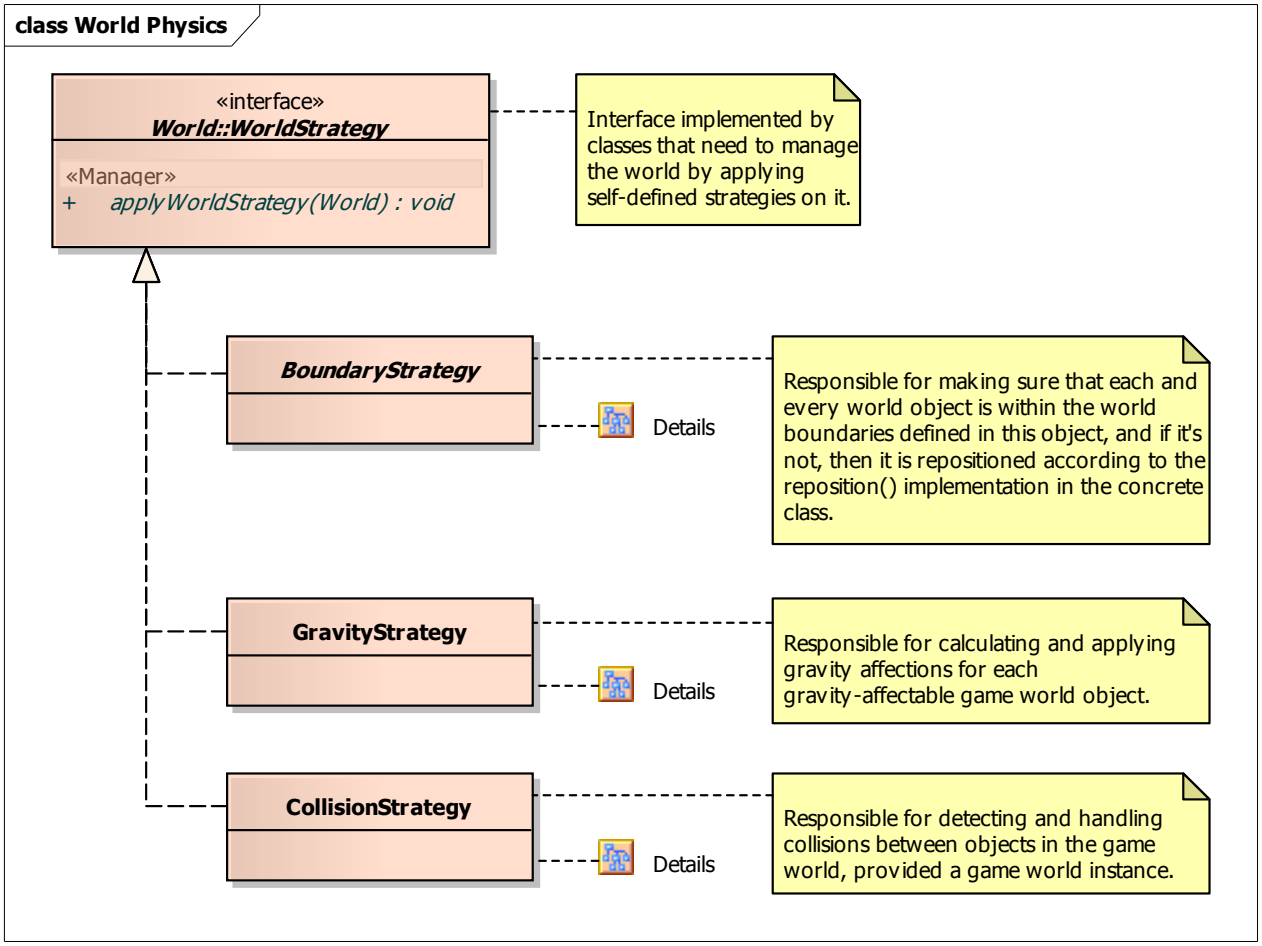


Diagram: World Physics - Boundary Strategies

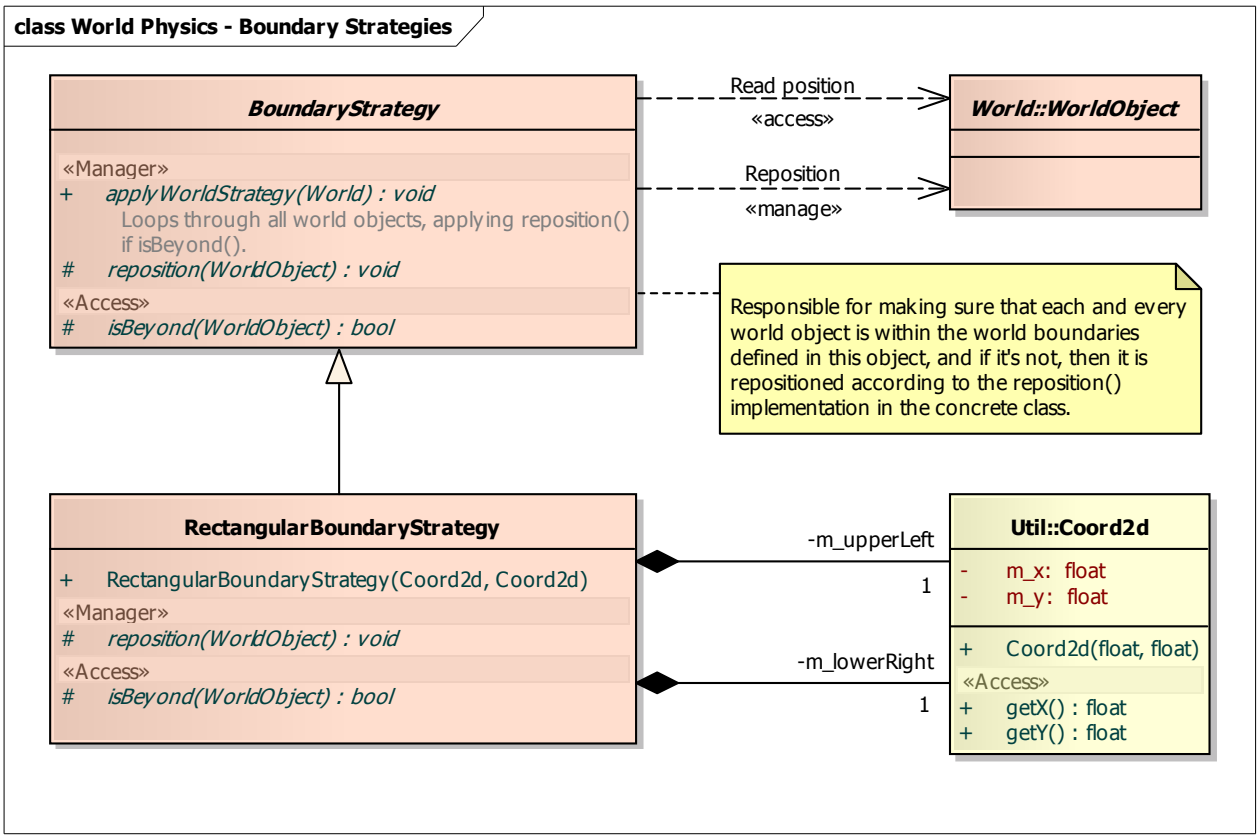


Diagram: AsteroidStrategy

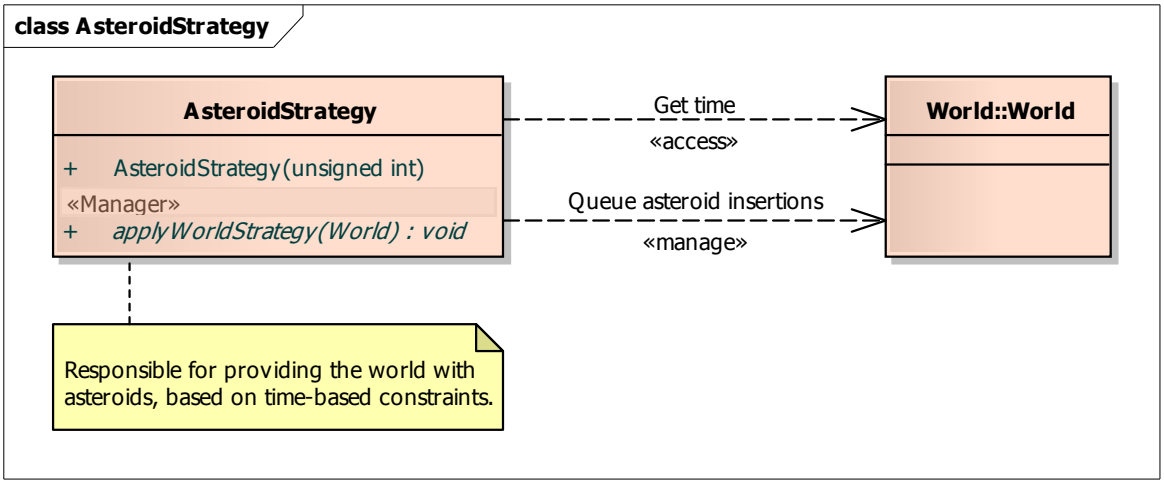


Diagram: ExpirationStrategy

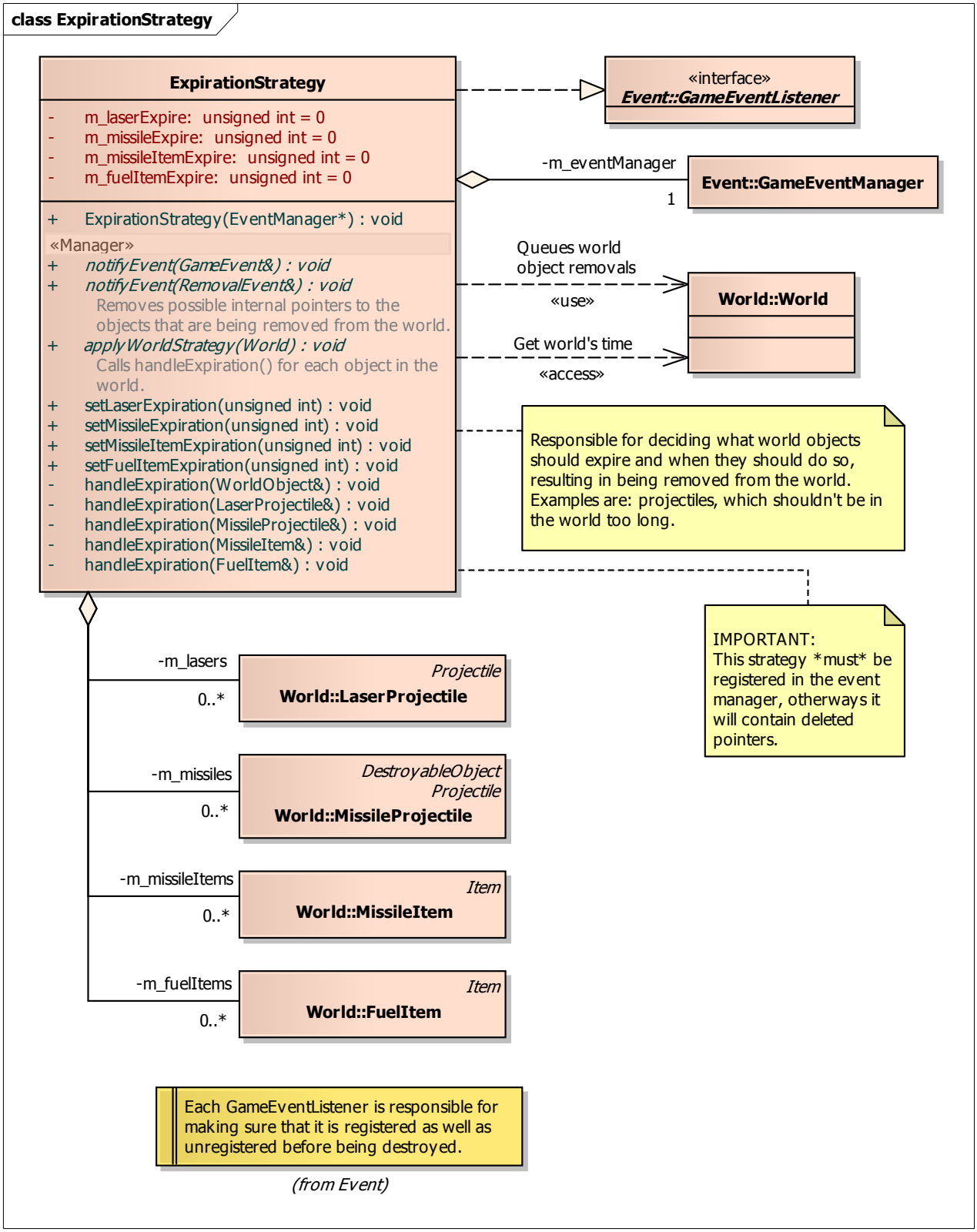


Diagram: ItemStrategy

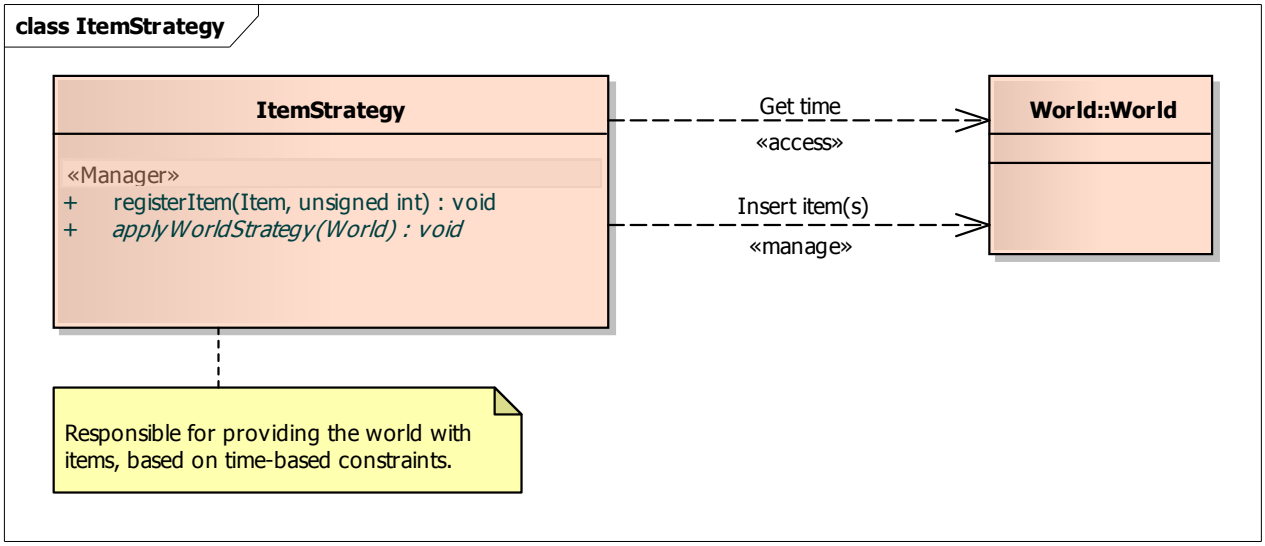


Diagram: World Life

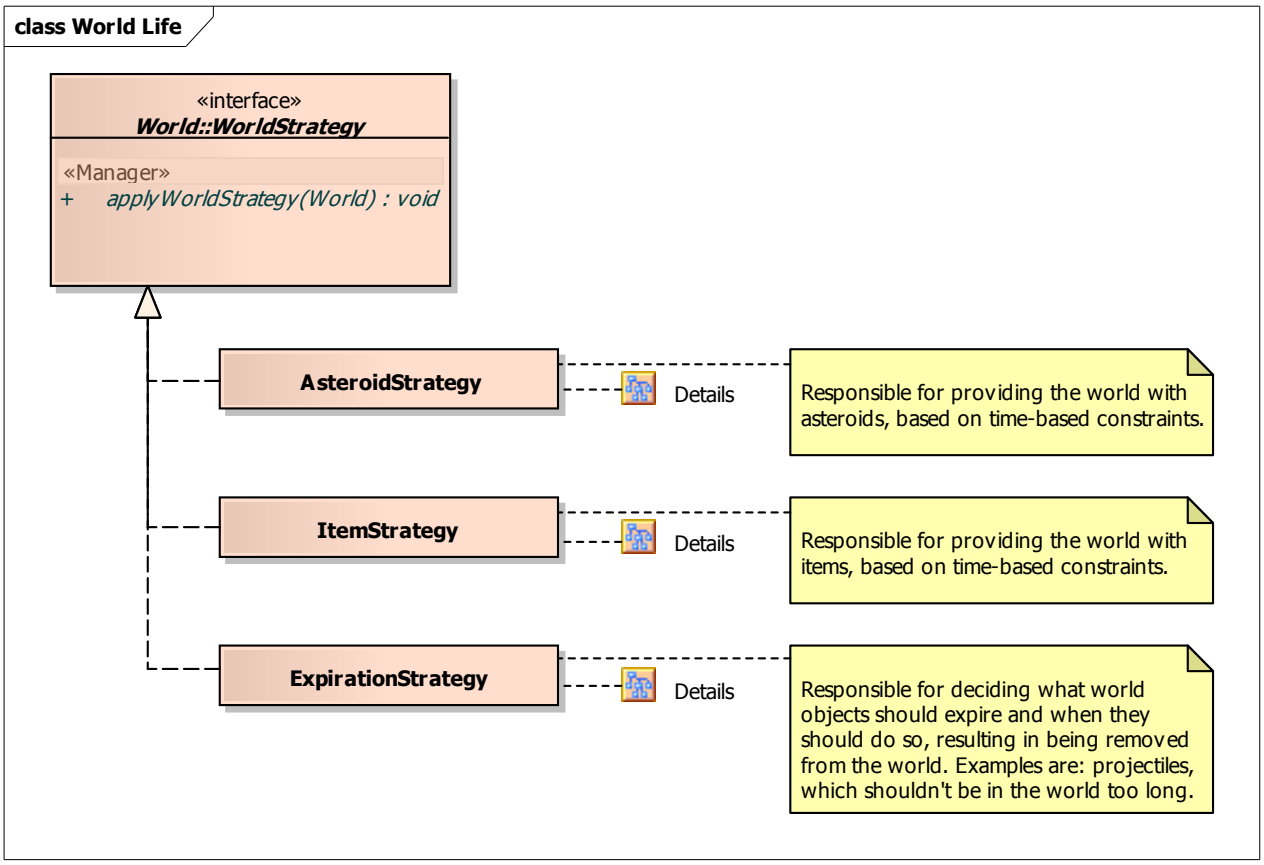


Diagram: CollisionEvent

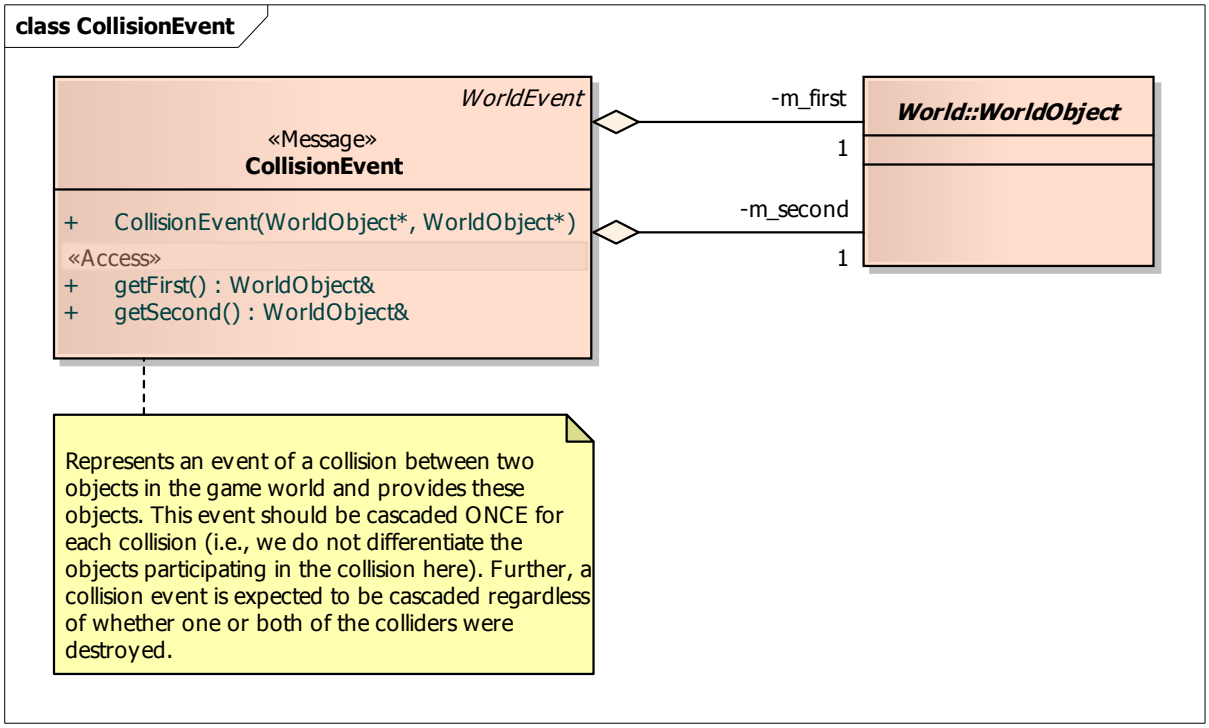


Diagram: DamageEvent

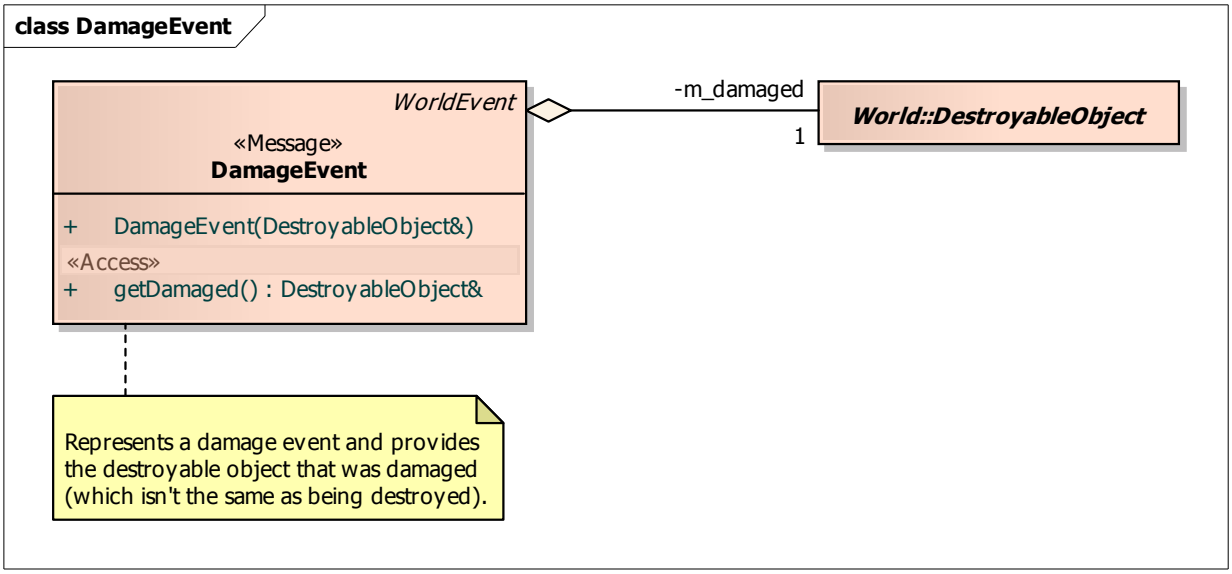


Diagram: DestructionEvent

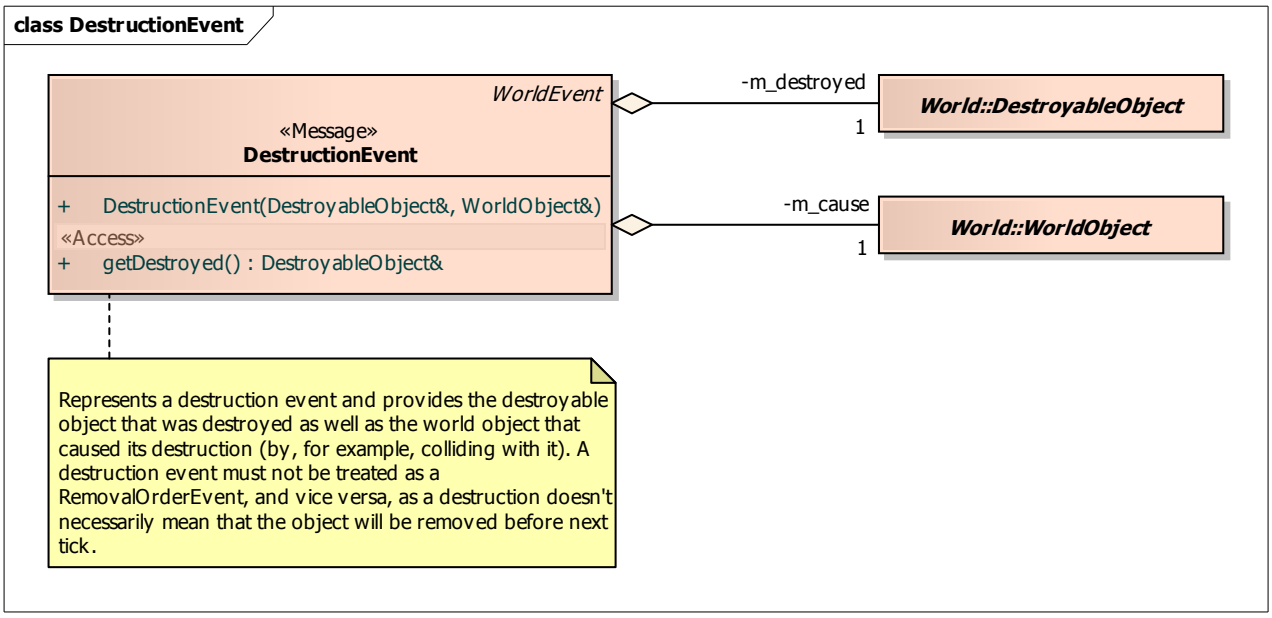


Diagram: InsertionEvent

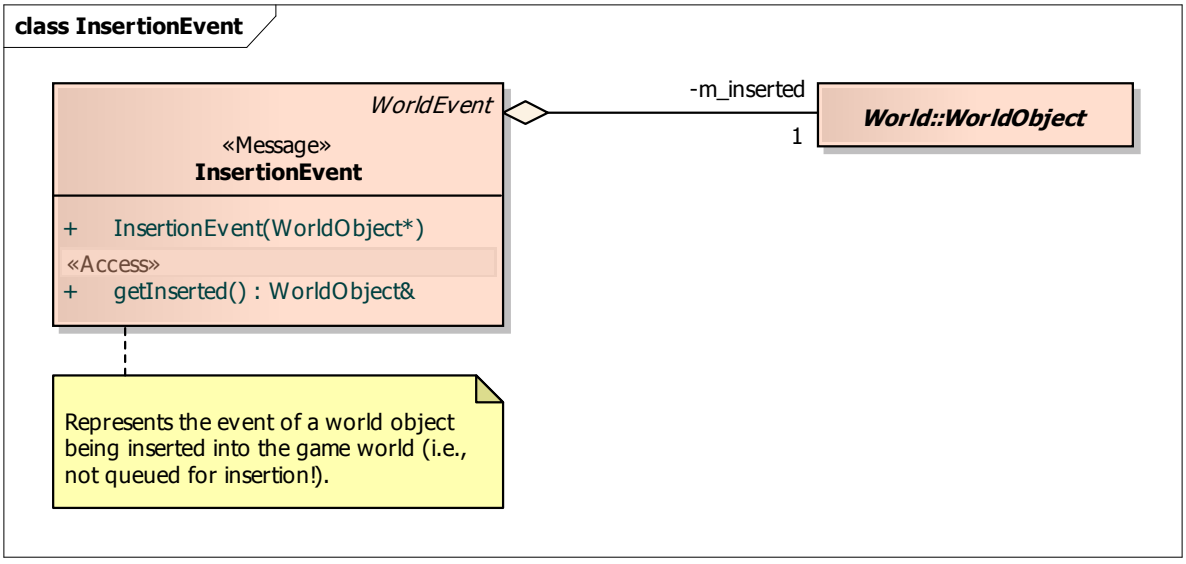


Diagram: ItemPickupEvent

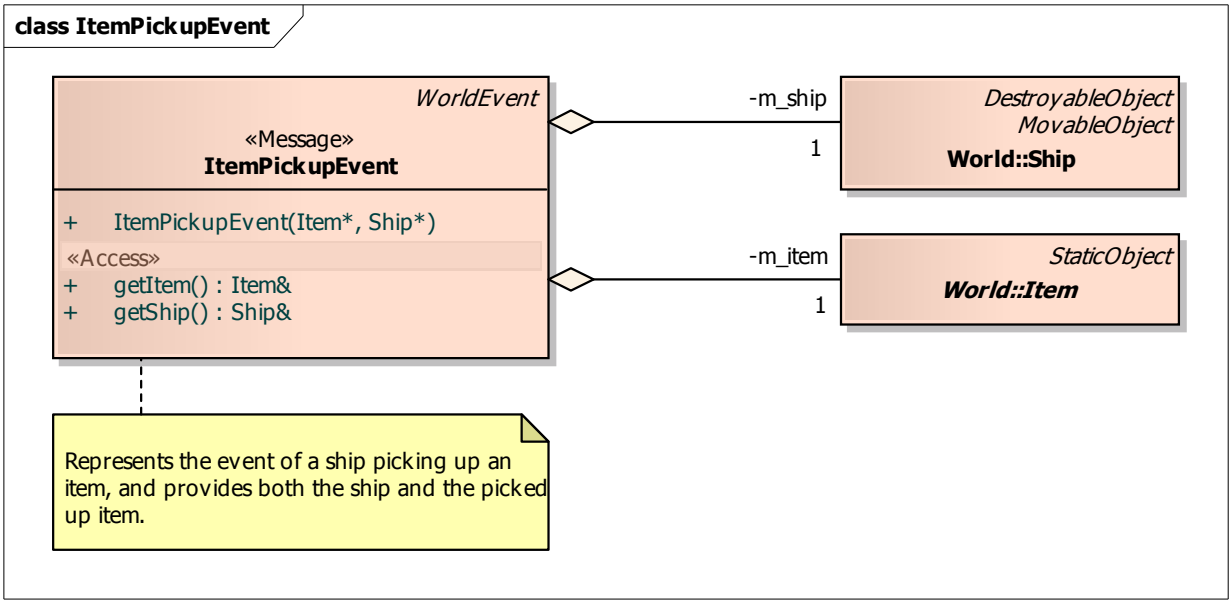


Diagram: ProjectileFireEvent

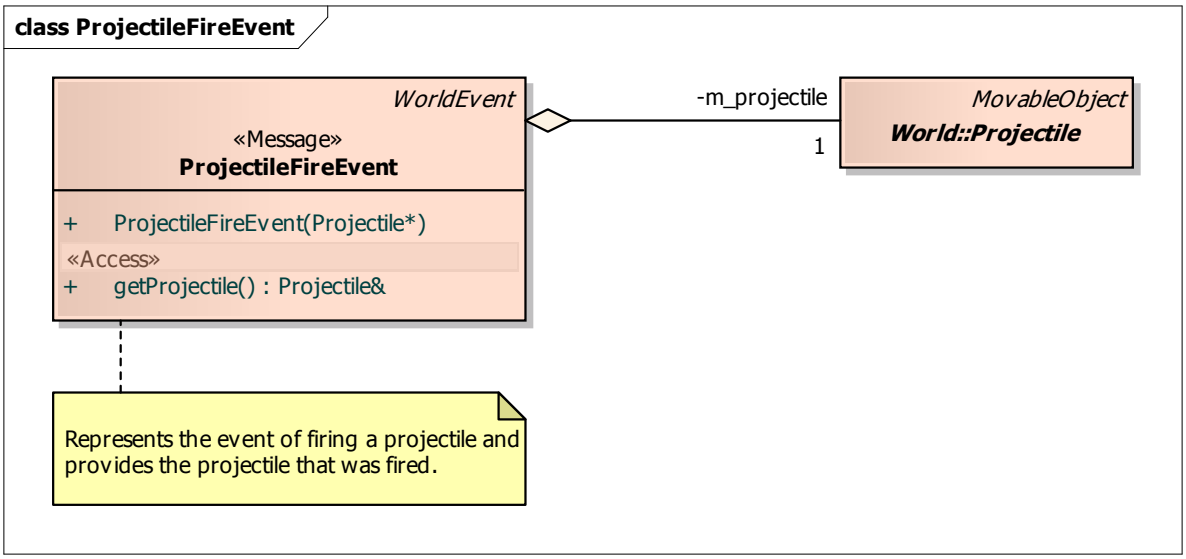


Diagram: RemovalEvent

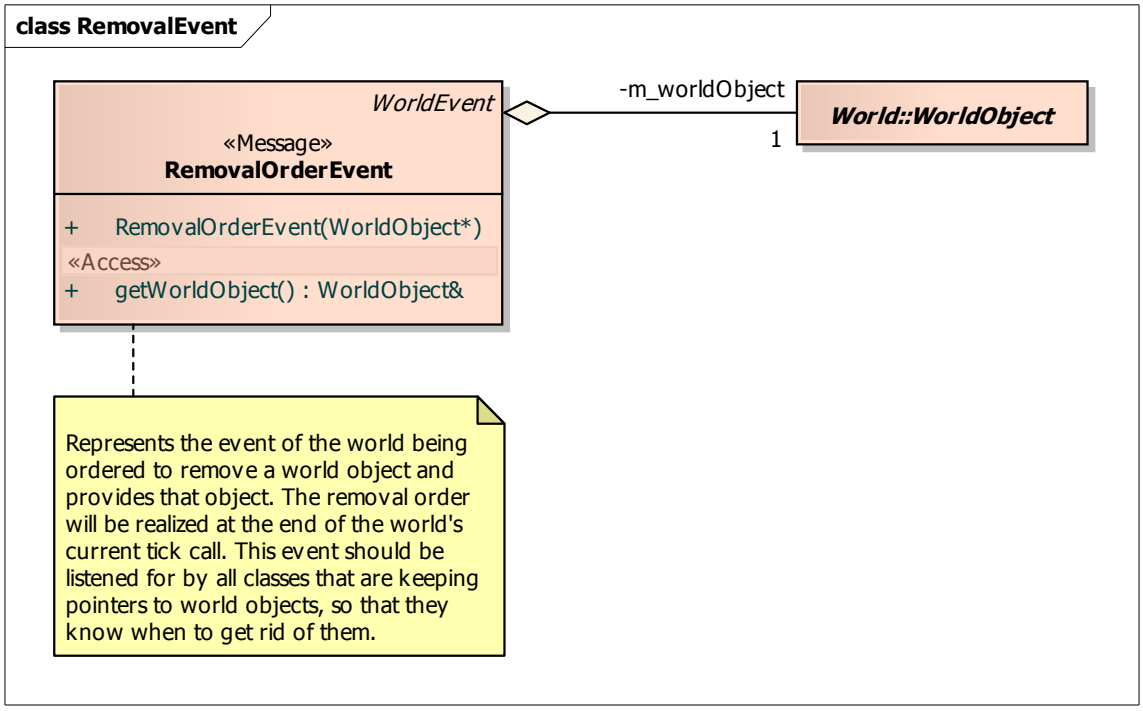


Diagram: World Event

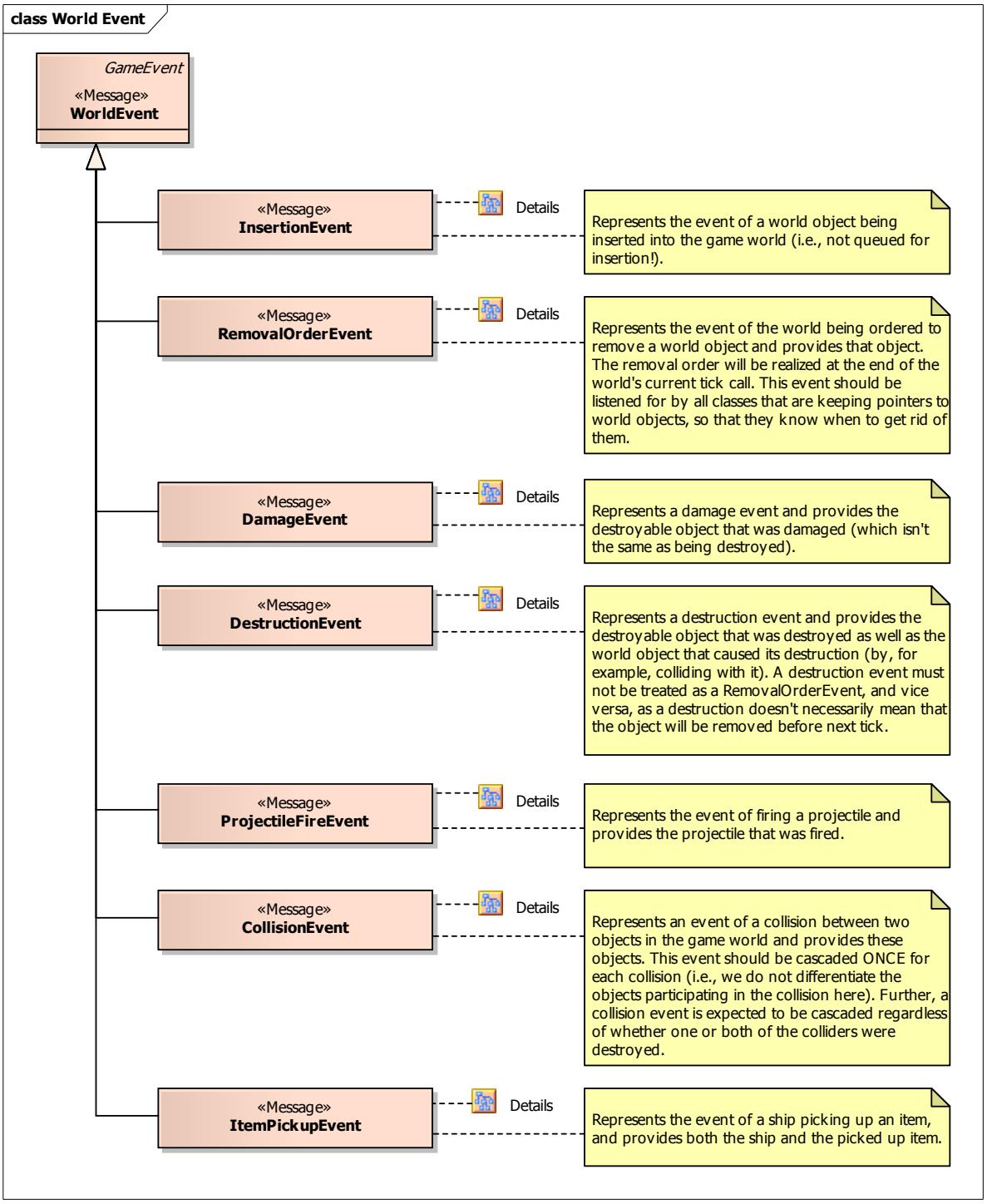


Diagram: ControlsConfigMenuState

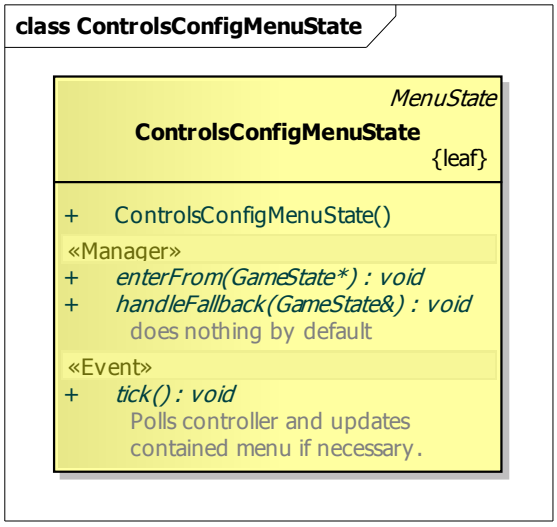


Diagram: ControlsMenuState

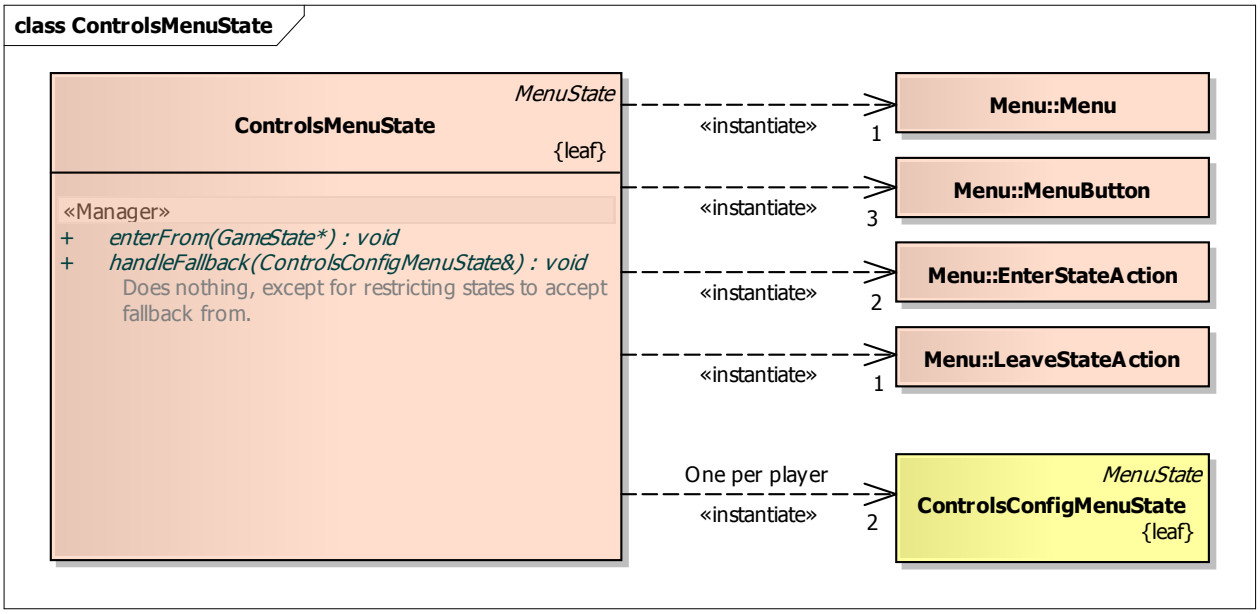


Diagram: Game State

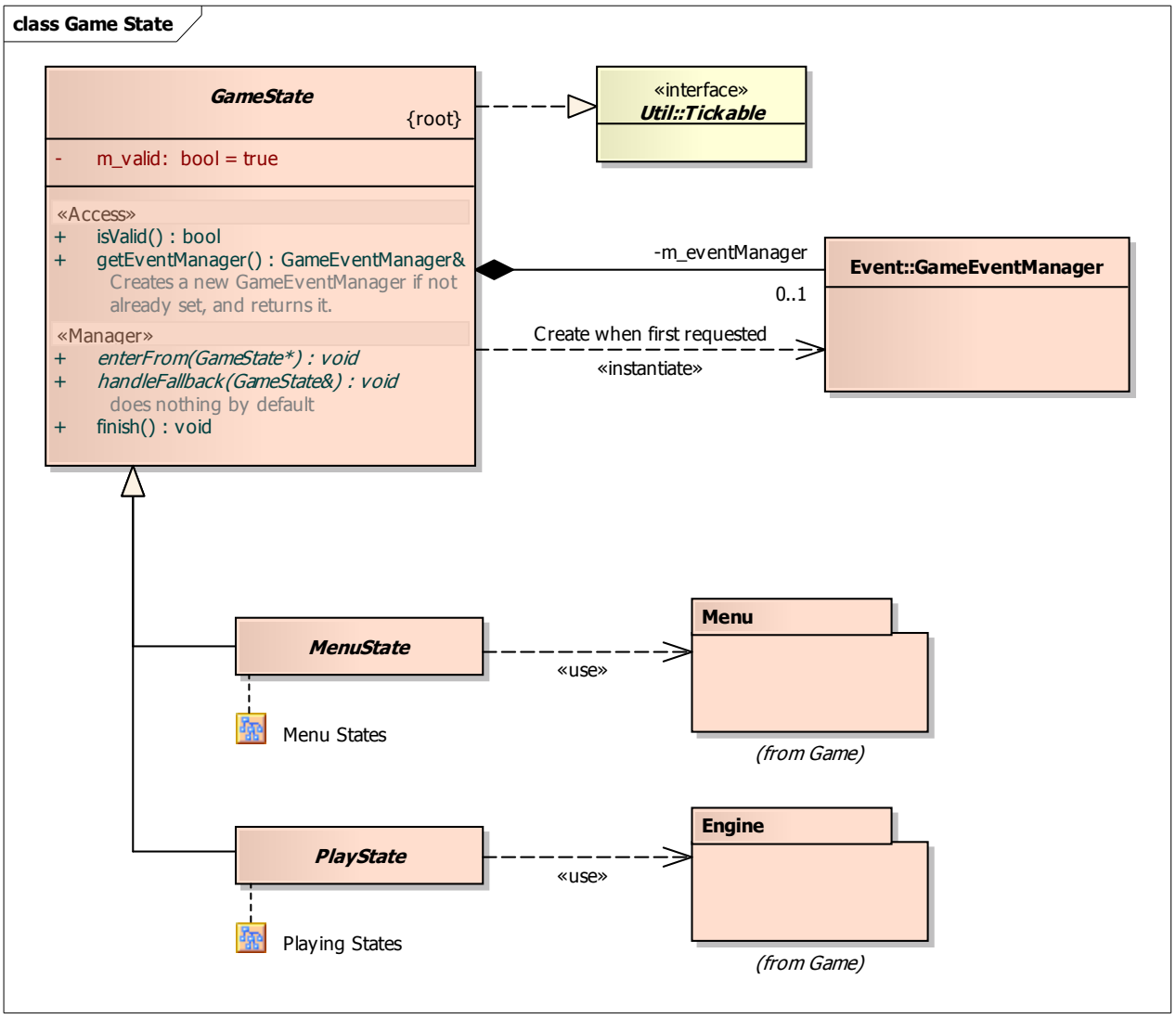


Diagram: HelpMenuState

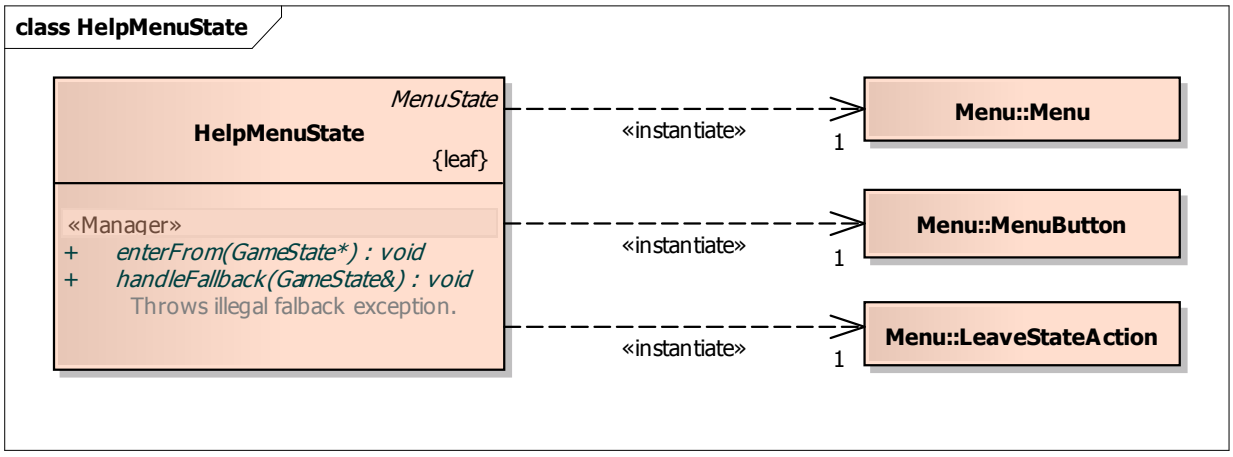


Diagram: HighScoreMenuState

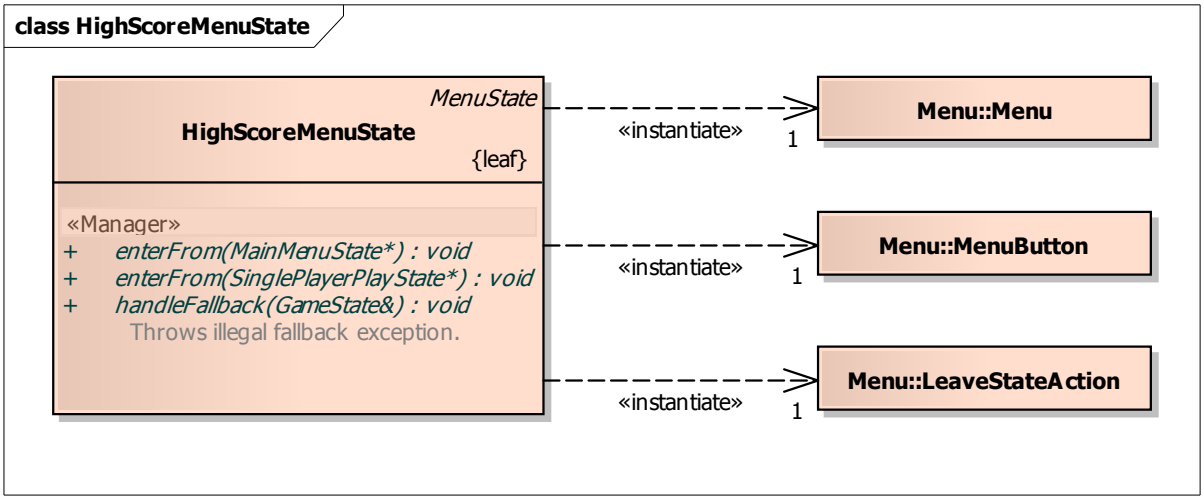


Diagram: MainMenuState

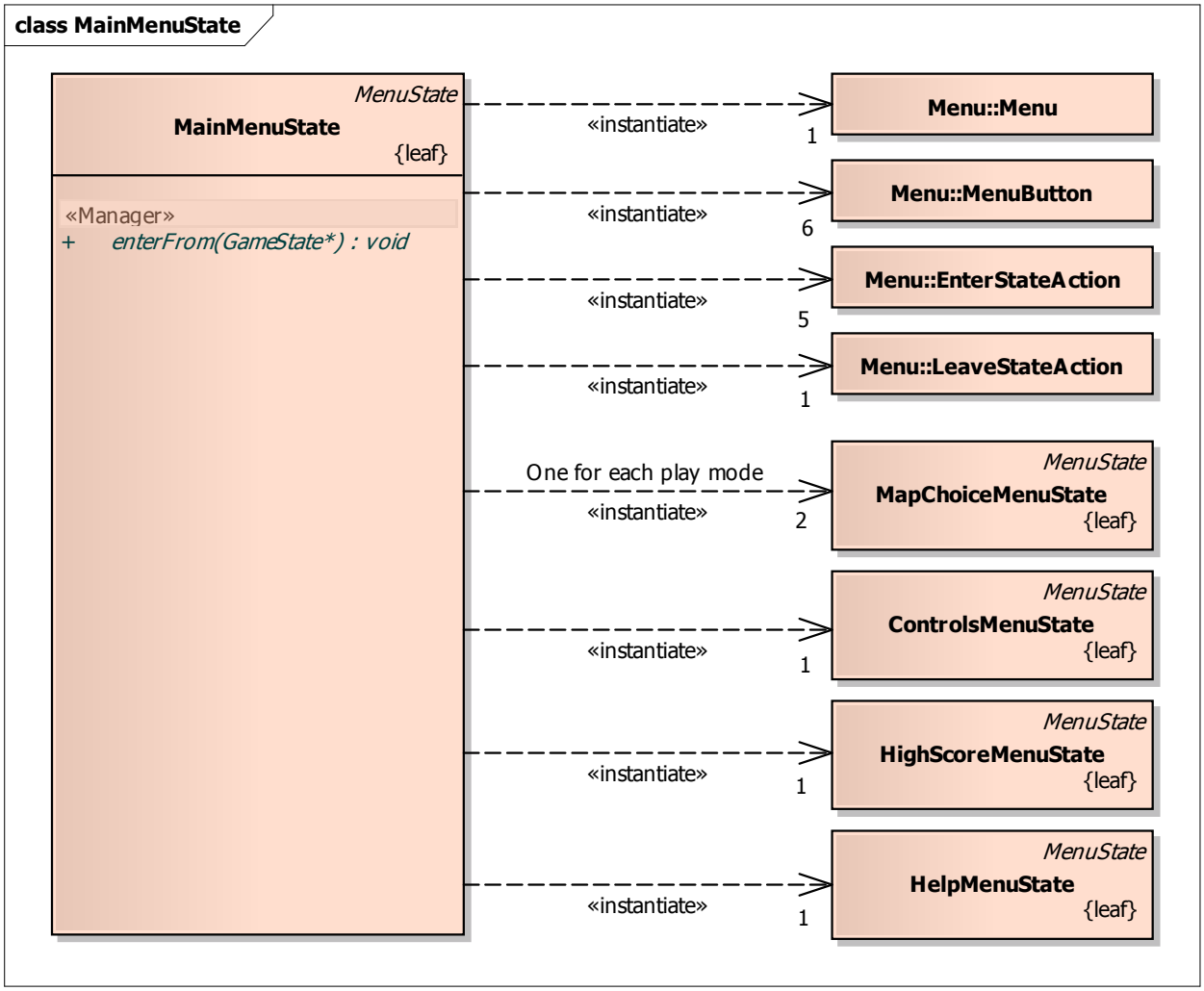


Diagram: MapChoiceMenuState

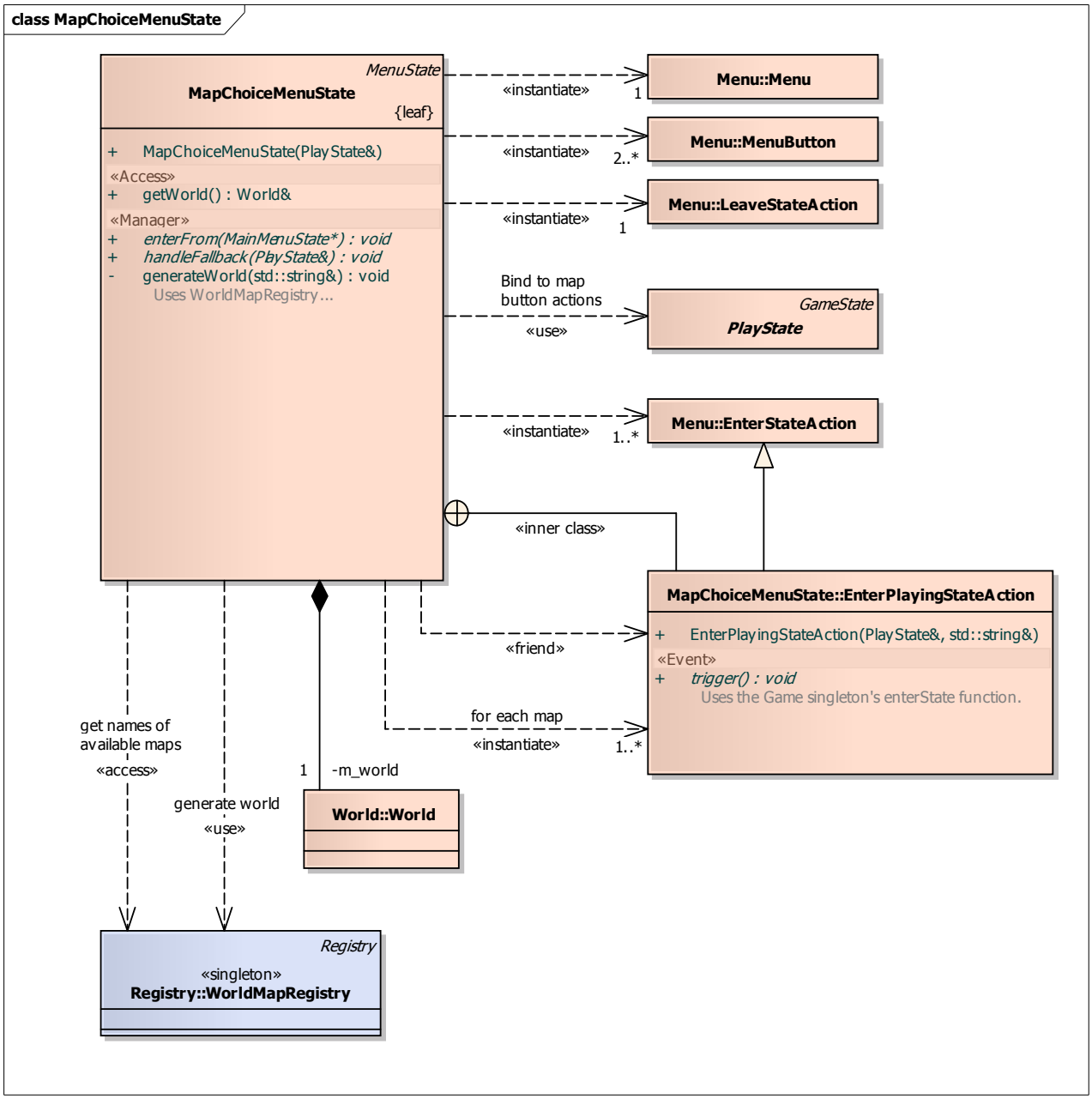


Diagram: Menu Game States

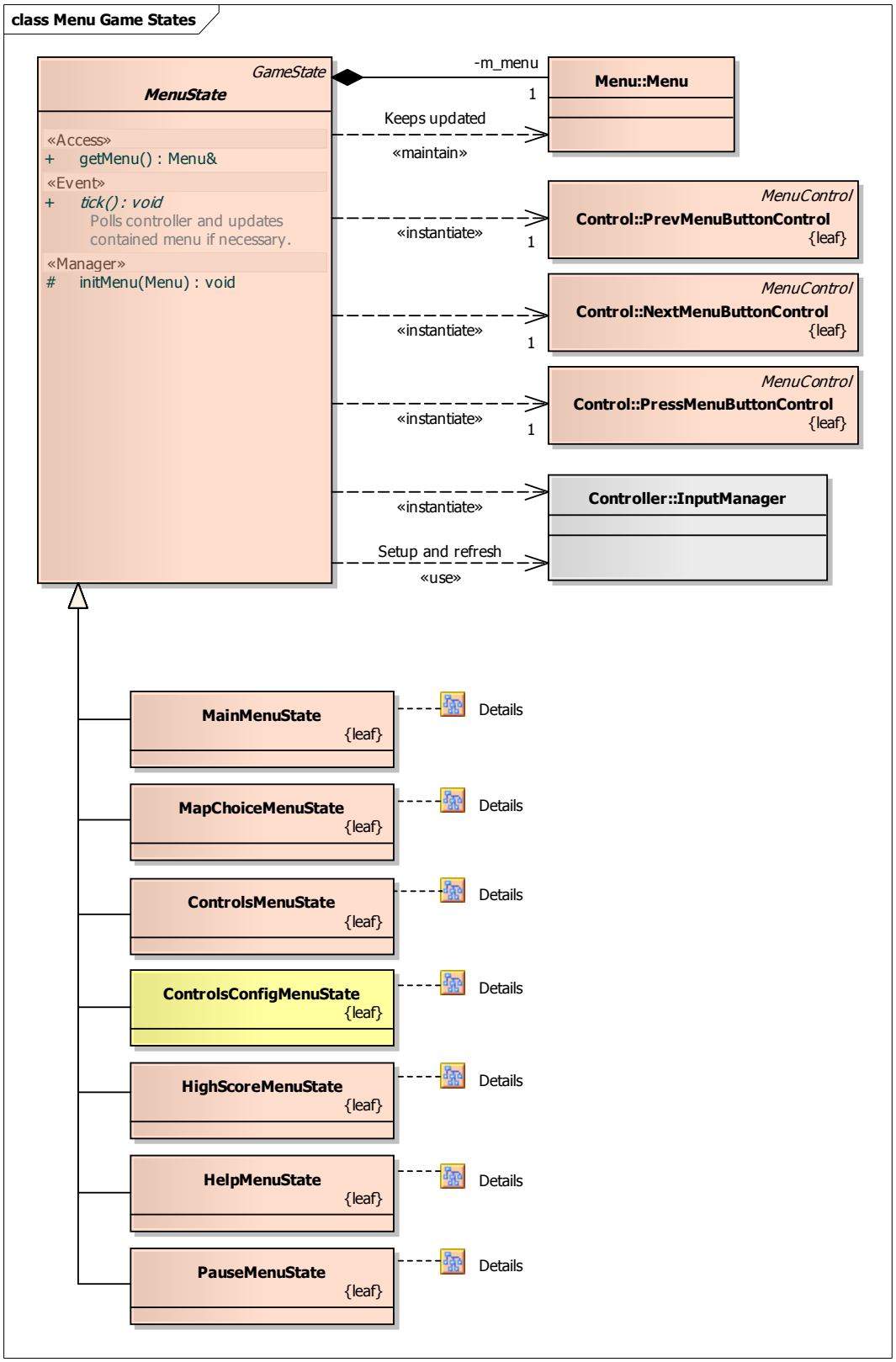


Diagram: PauseMenuState

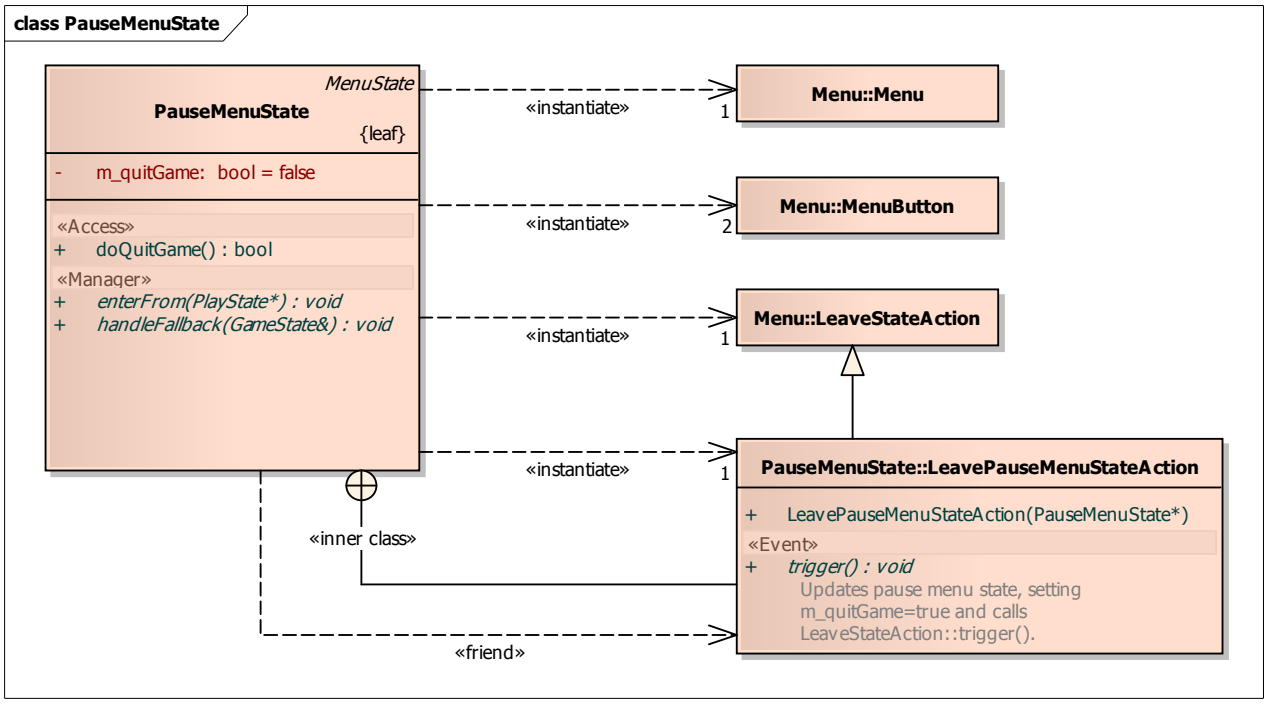


Diagram: Play Game States

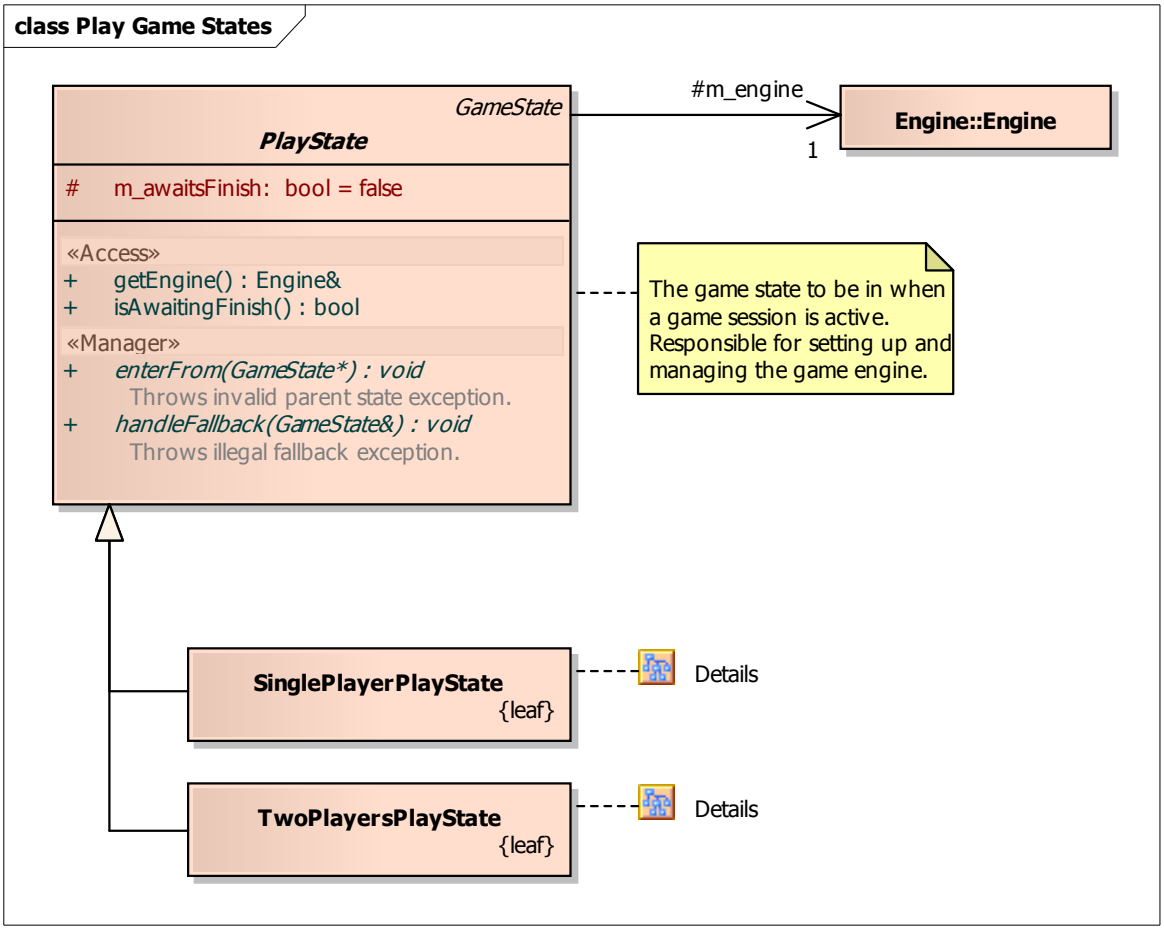


Diagram: SinglePlayerPlayState 1

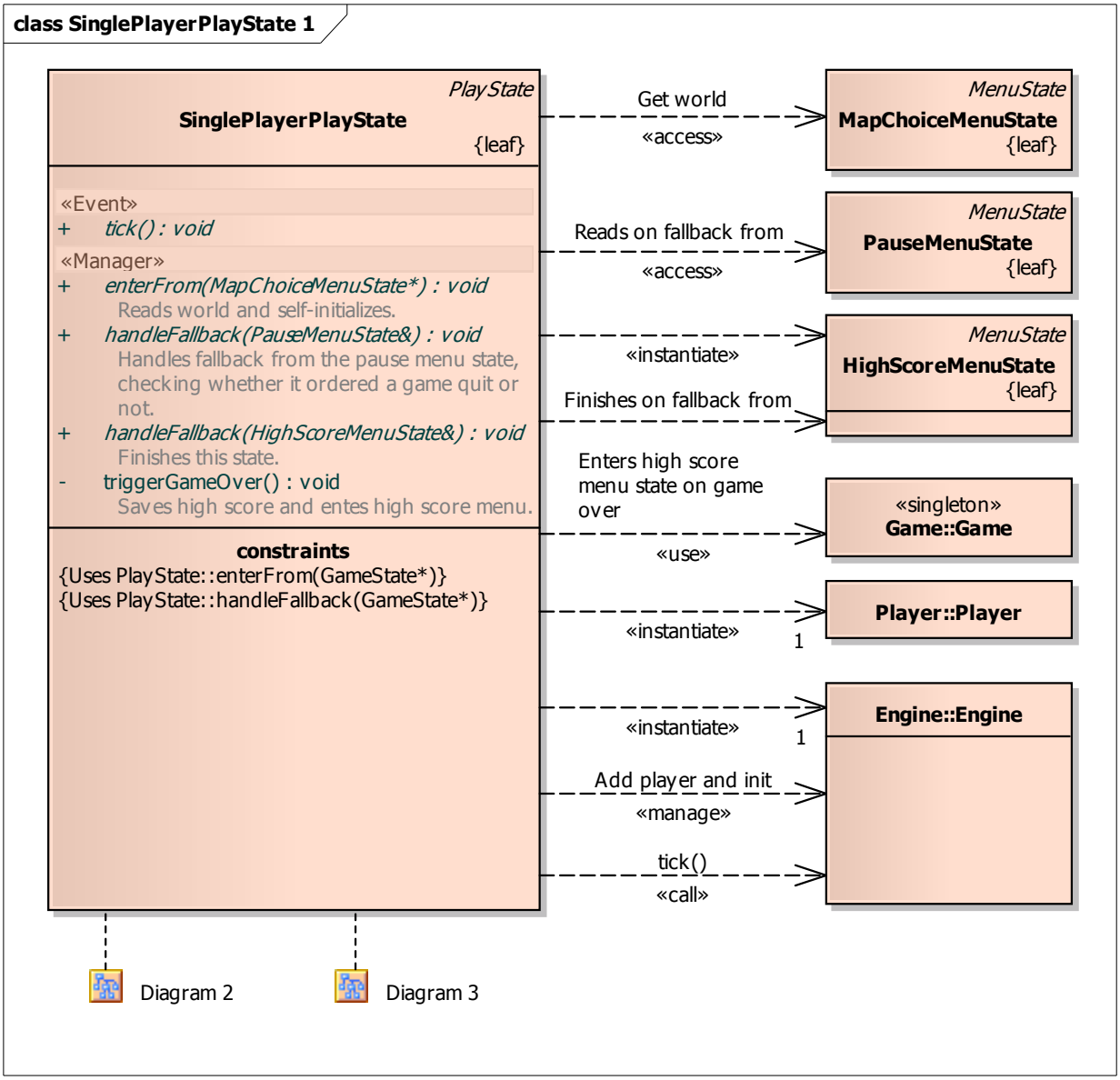


Diagram: SinglePlayerPlayState 2

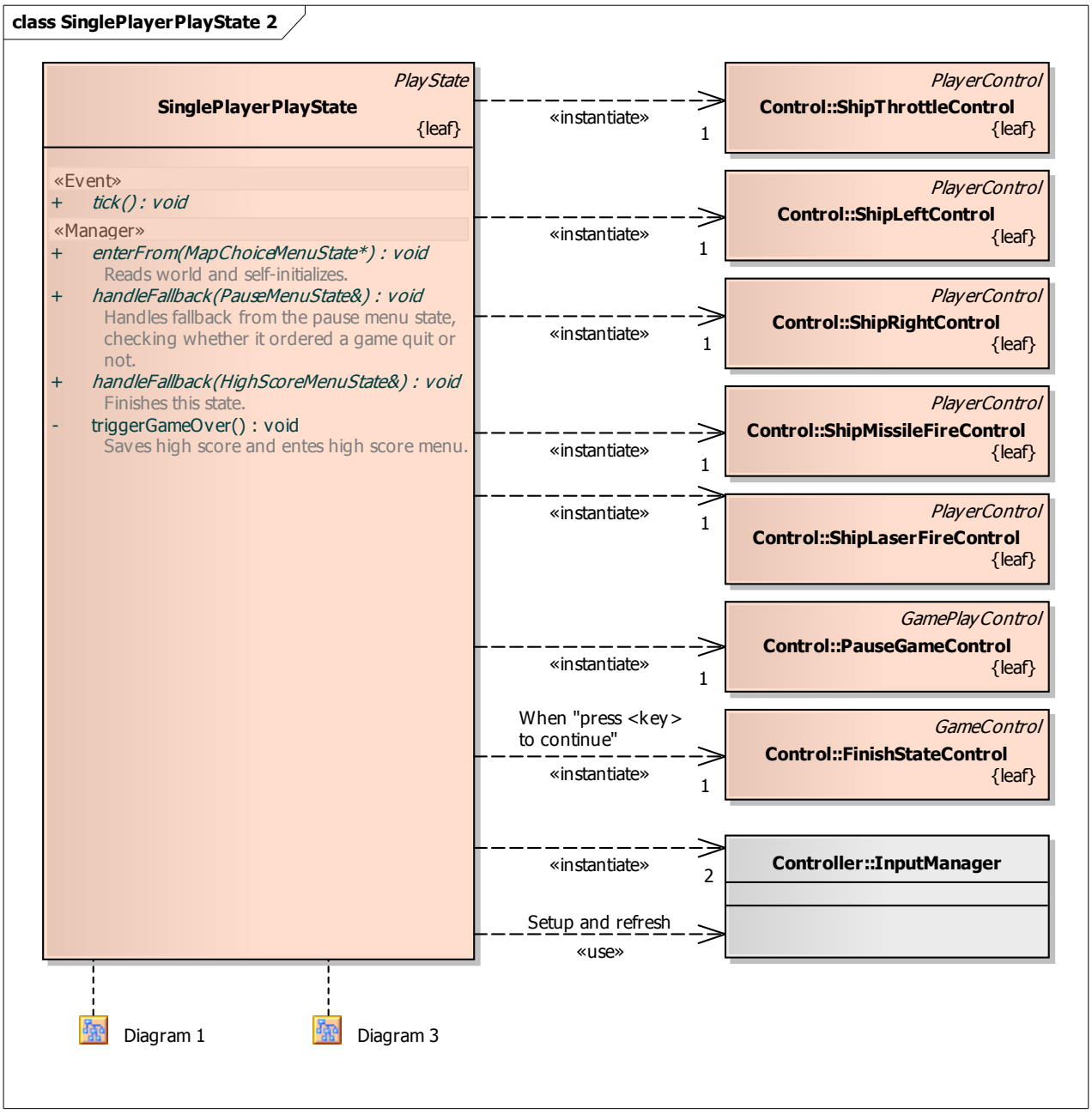


Diagram: SinglePlayerPlayState 3

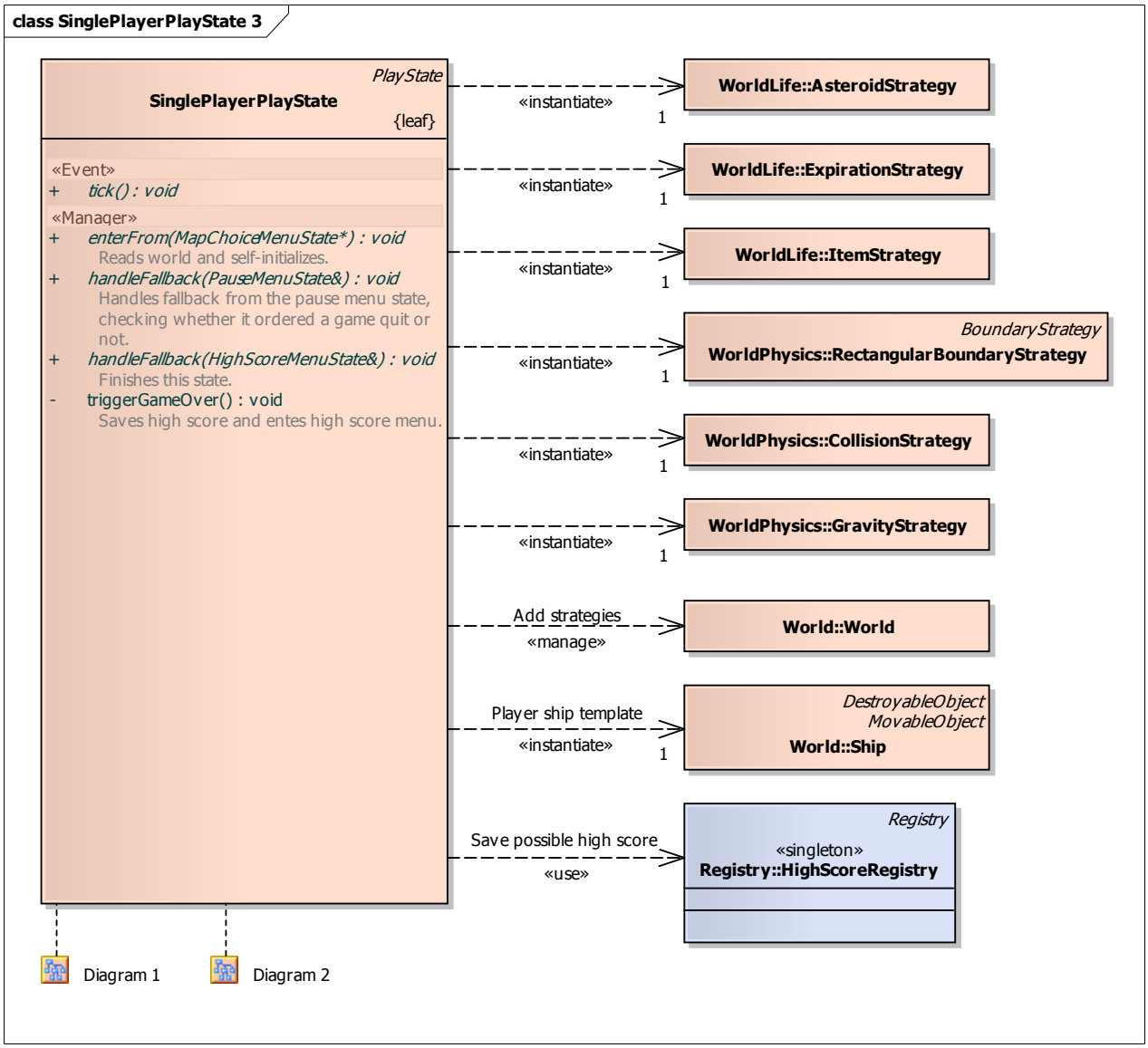


Diagram: TwoPlayersPlayState 1

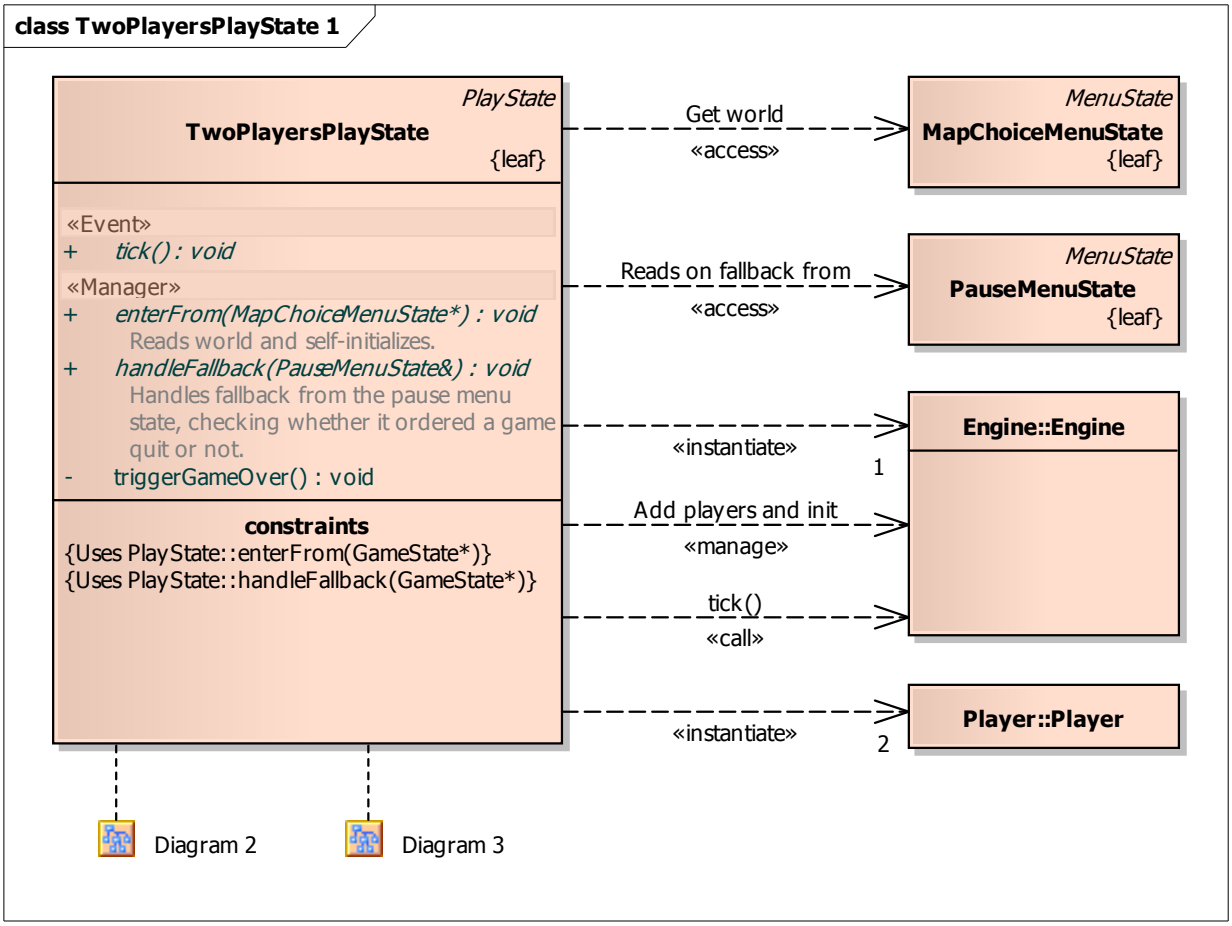


Diagram: TwoPlayersPlayState 2

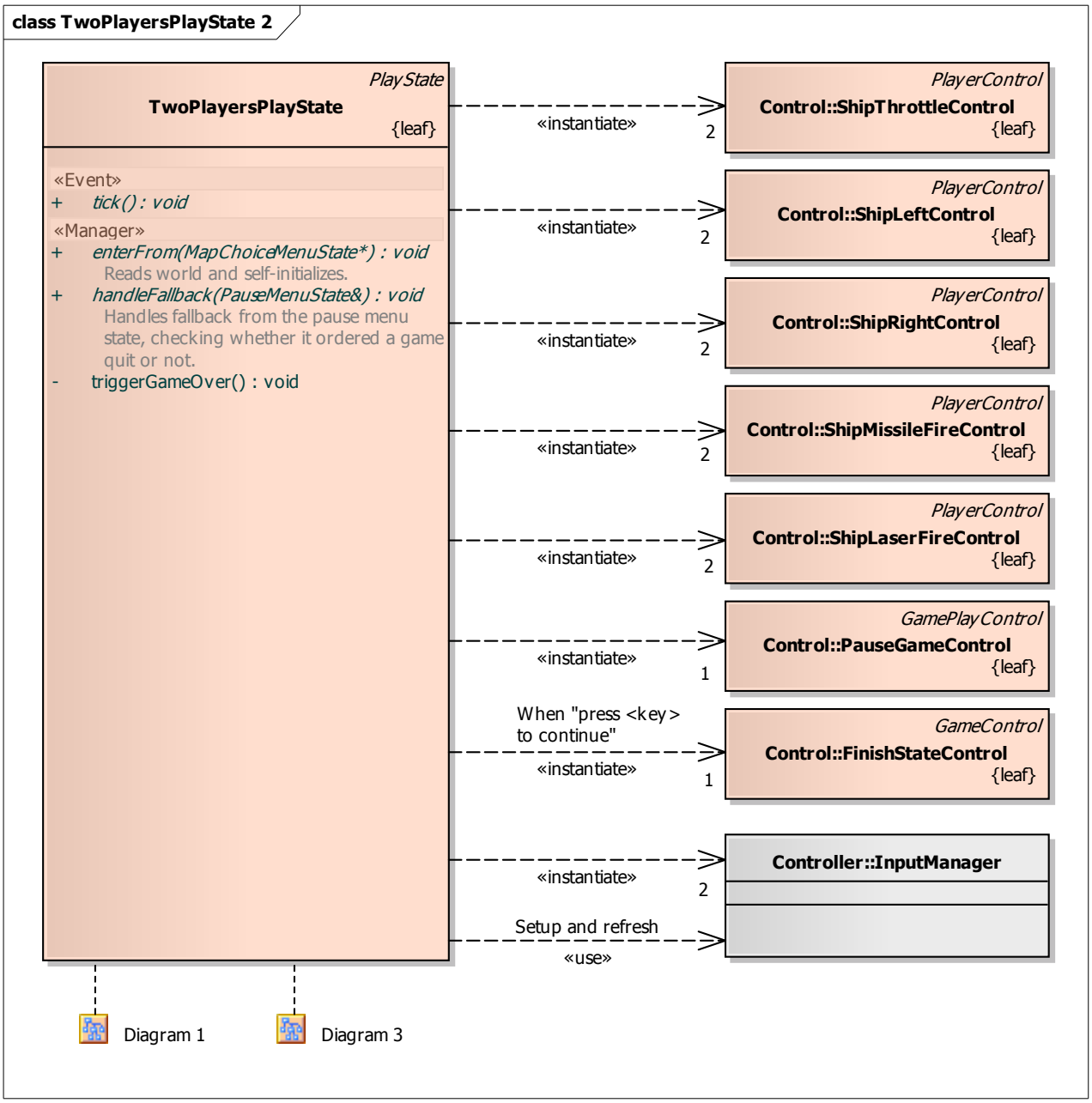


Diagram: TwoPlayersPlayState 3

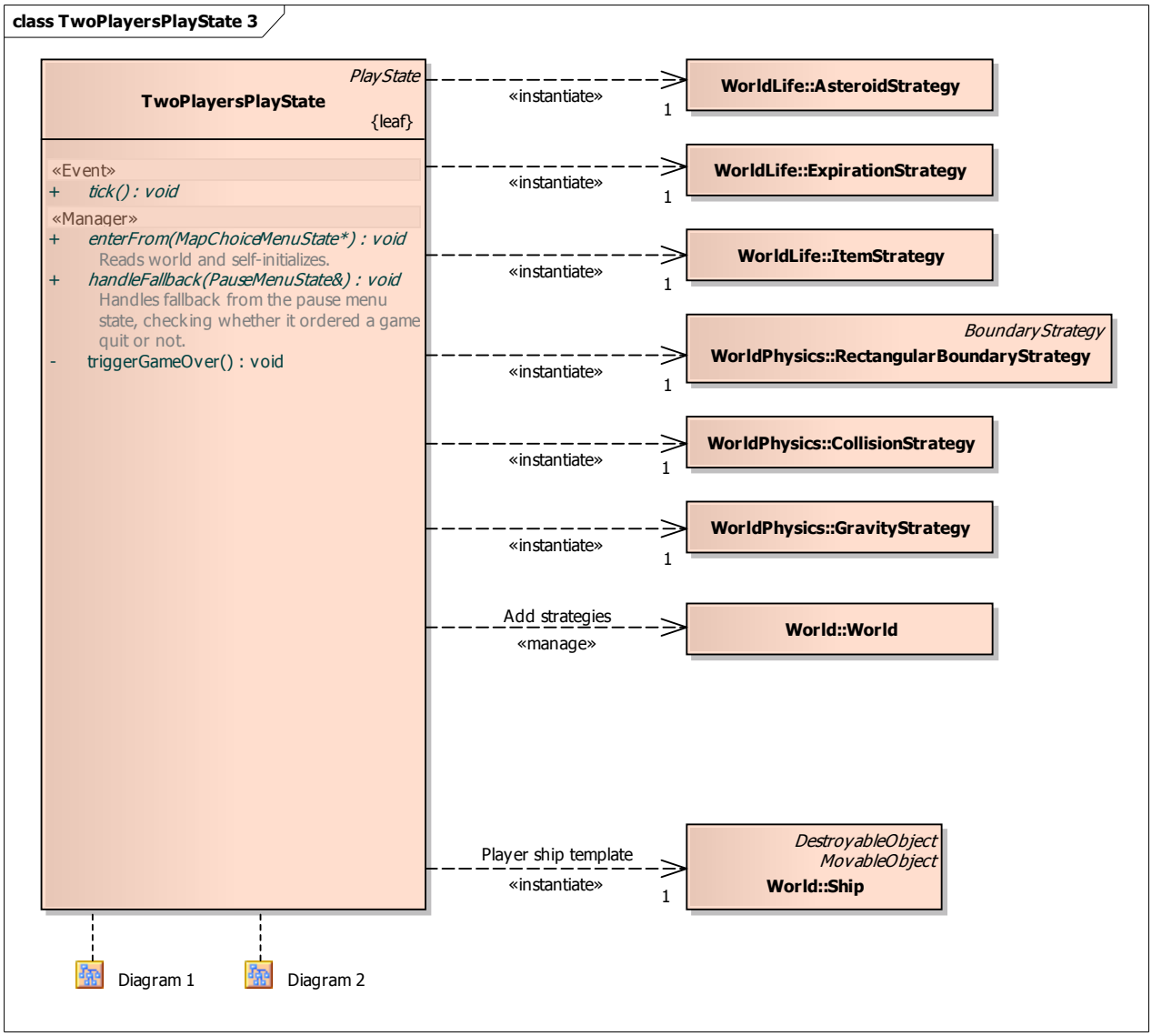


Diagram: Asteroid

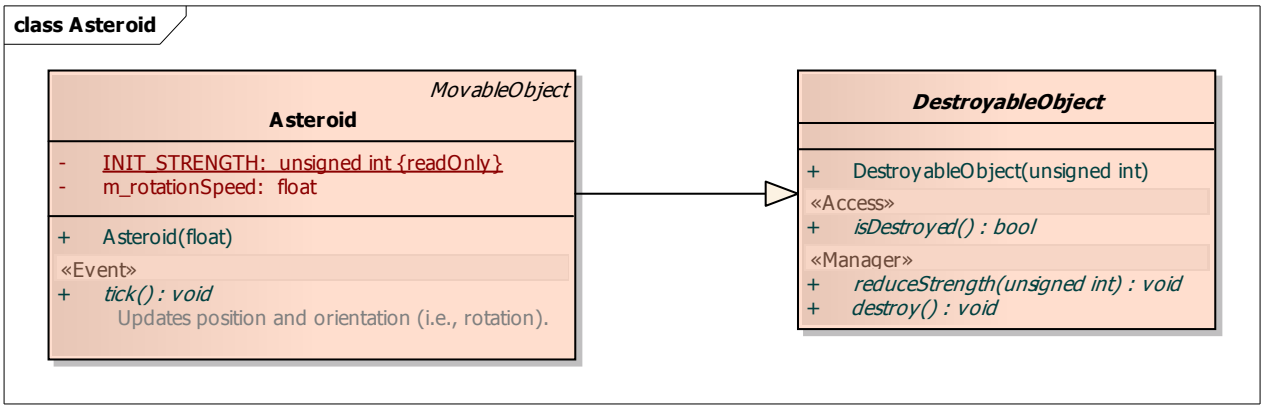


Diagram: Game World

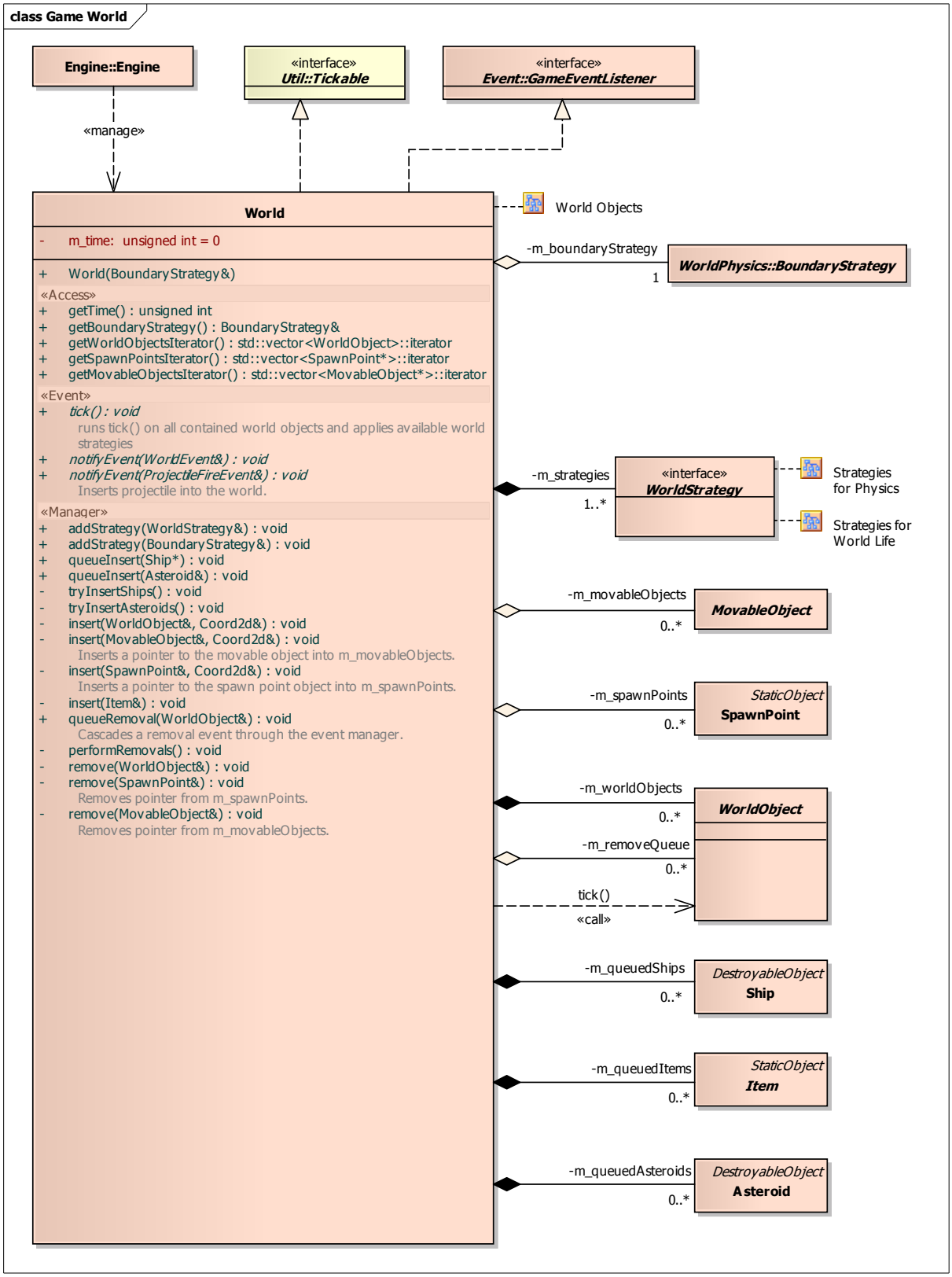


Diagram: Game World Object Destroying

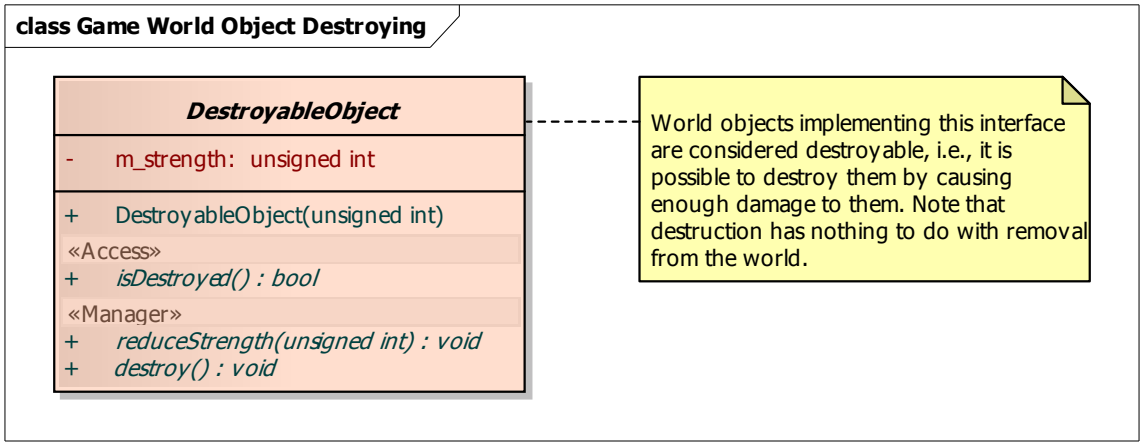


Diagram: Game World Objects

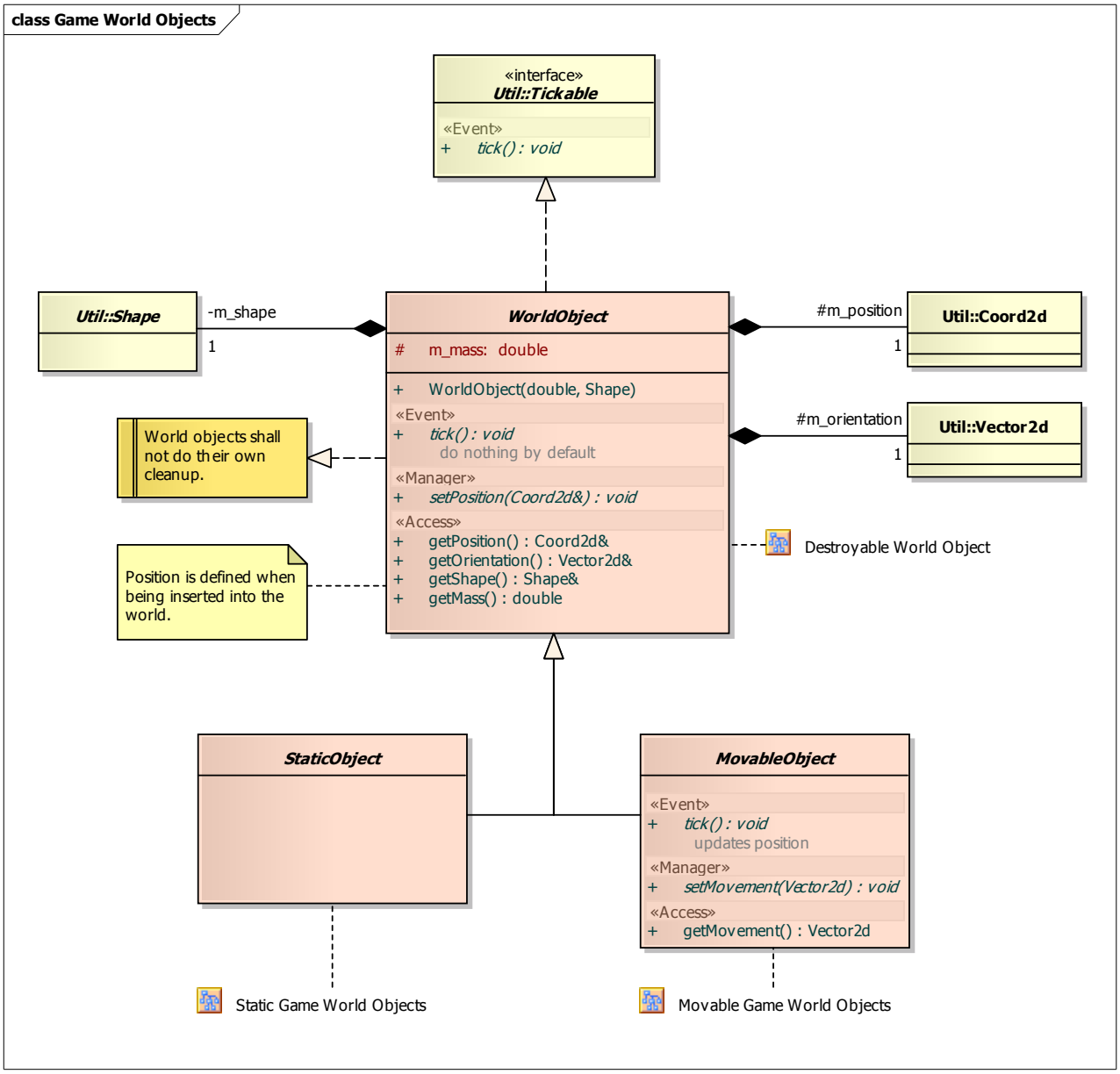


Diagram: Movable Game World Objects

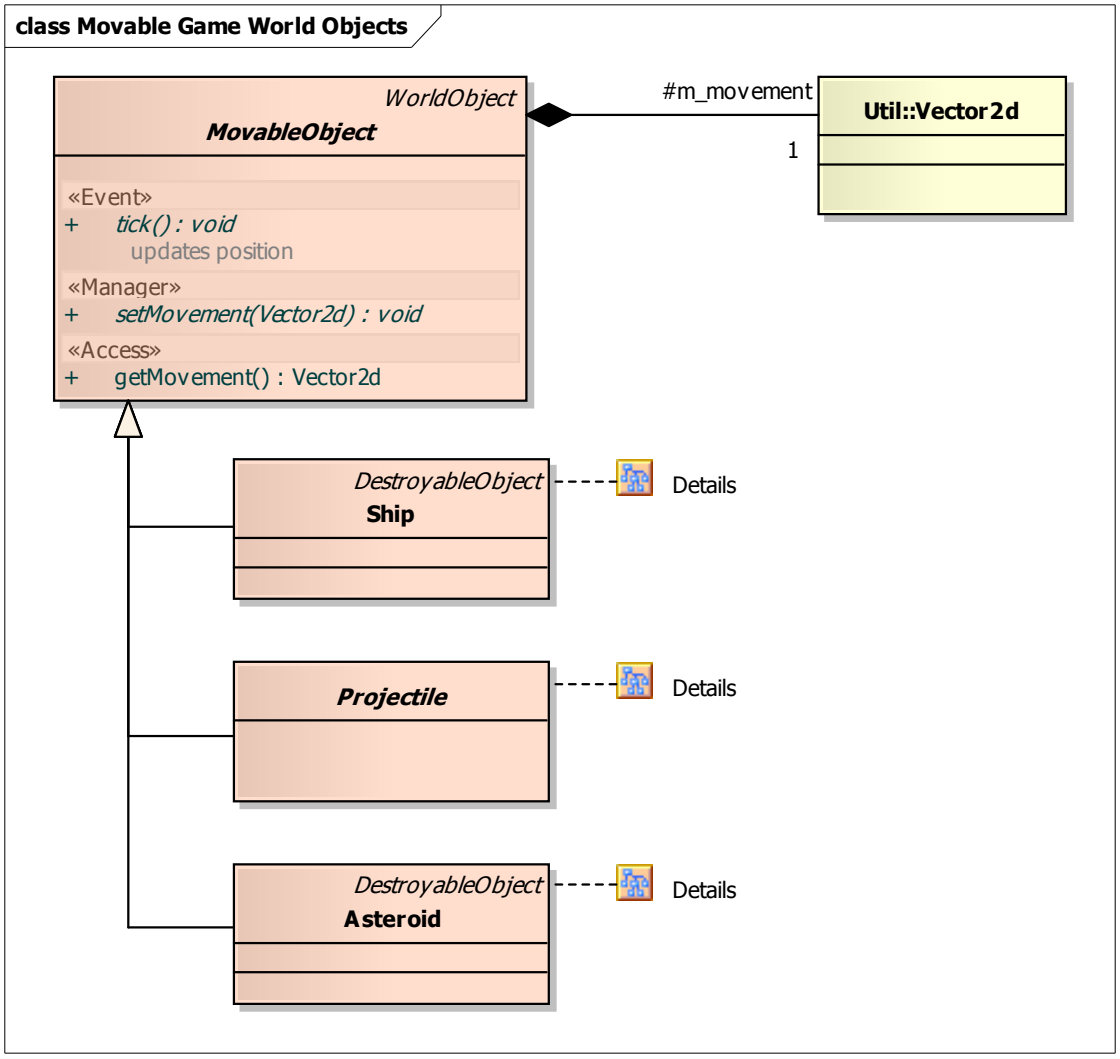


Diagram: Projectile

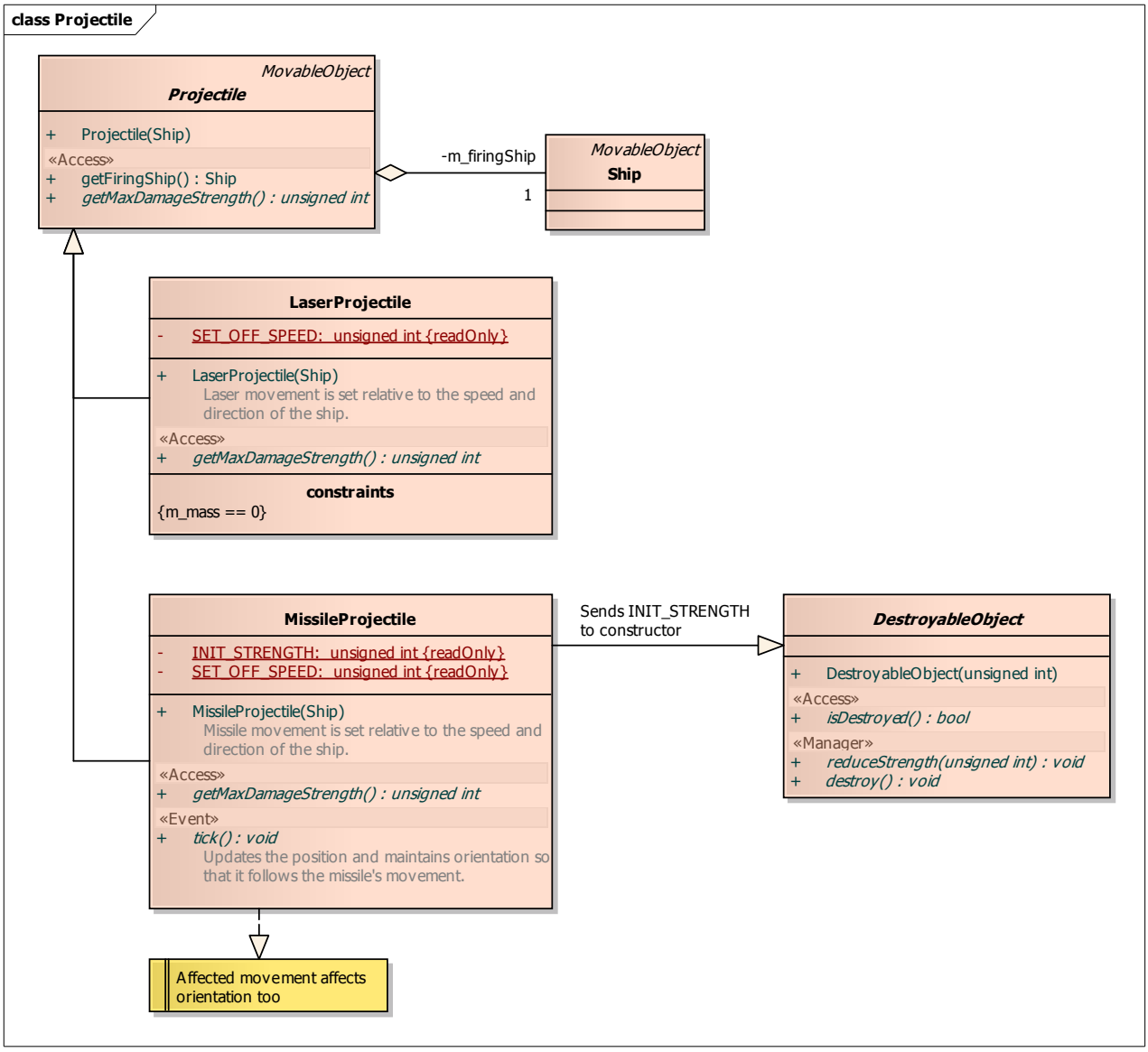


Diagram: Ship

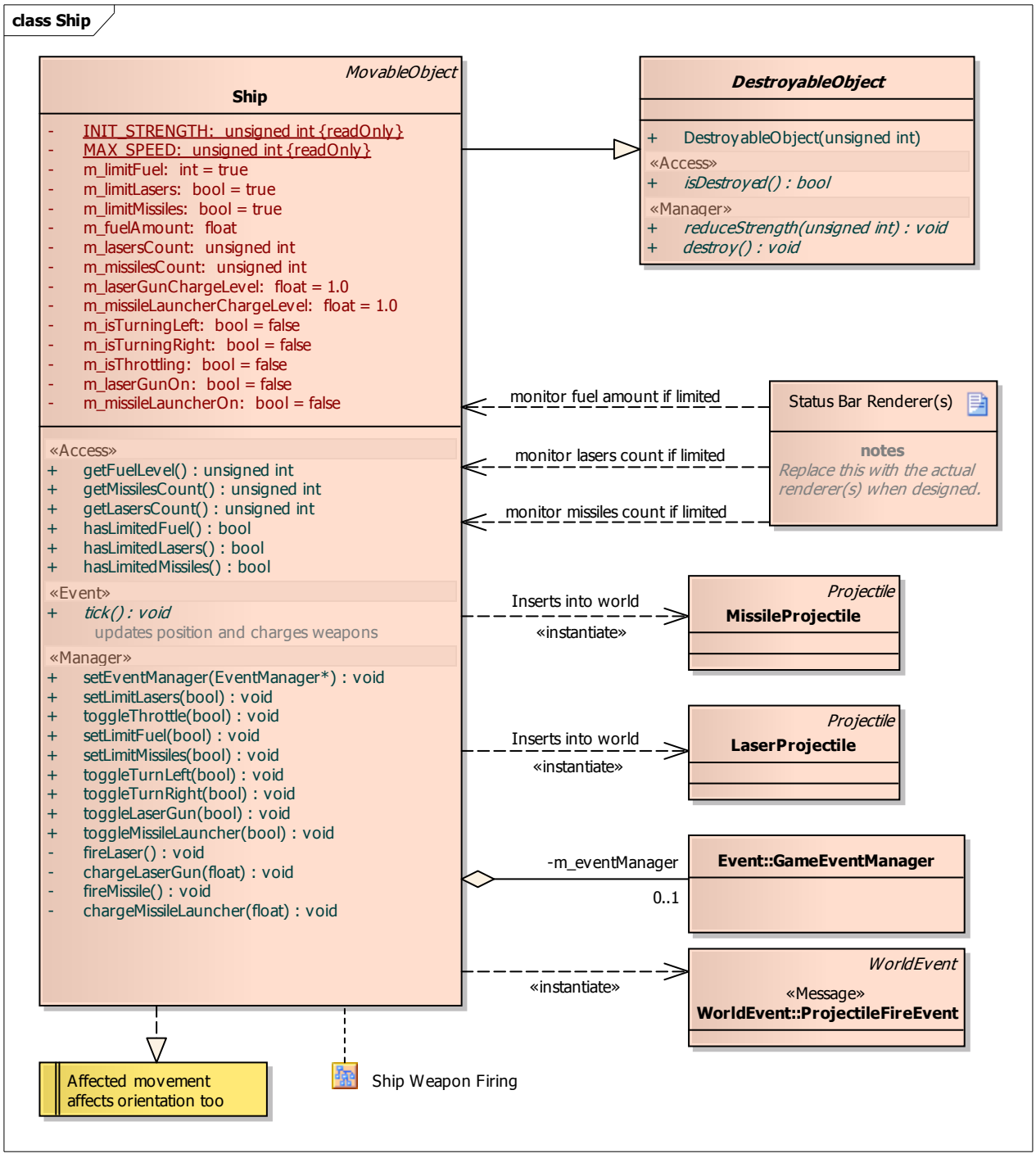


Diagram: Static Game World Objects

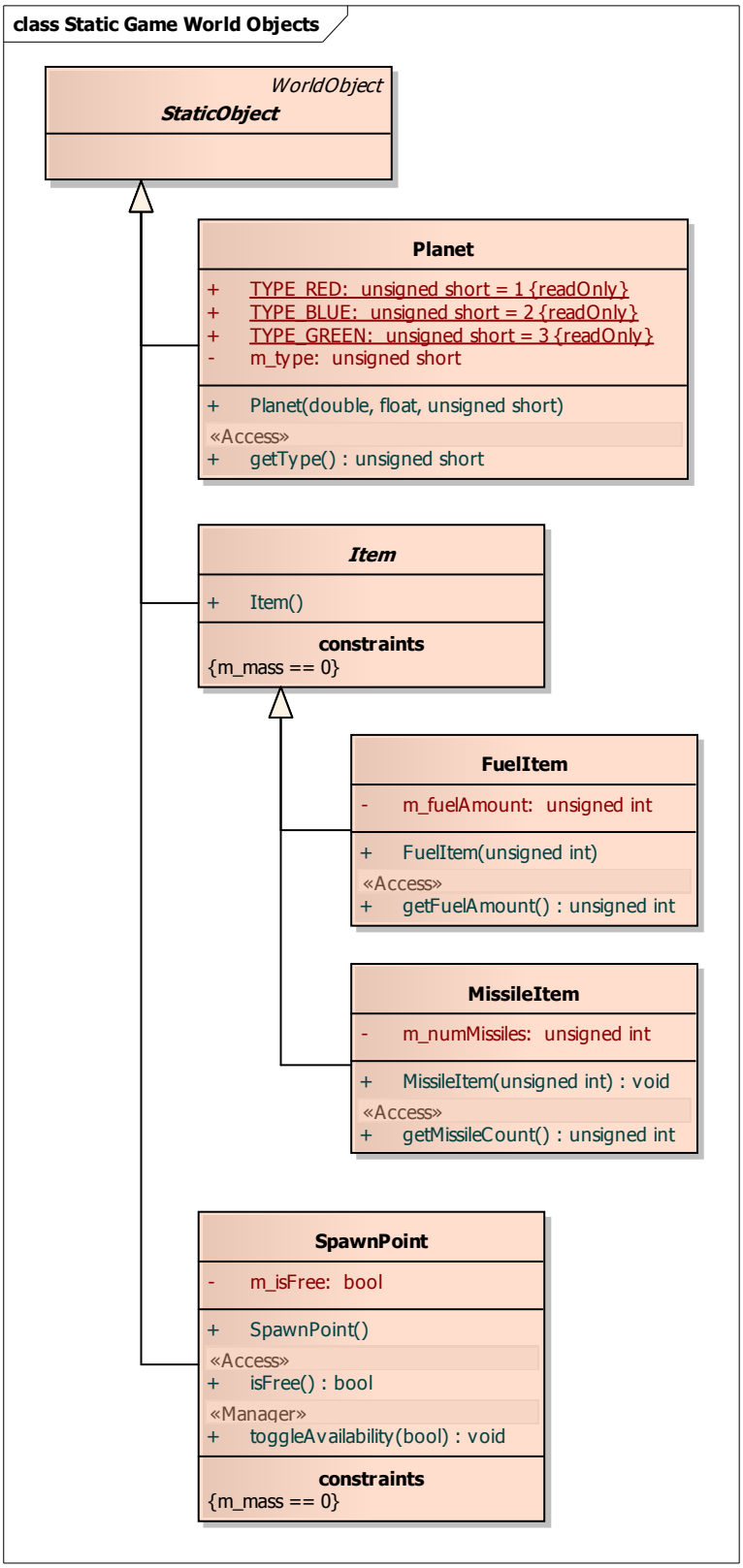


Diagram: EnterStateAction

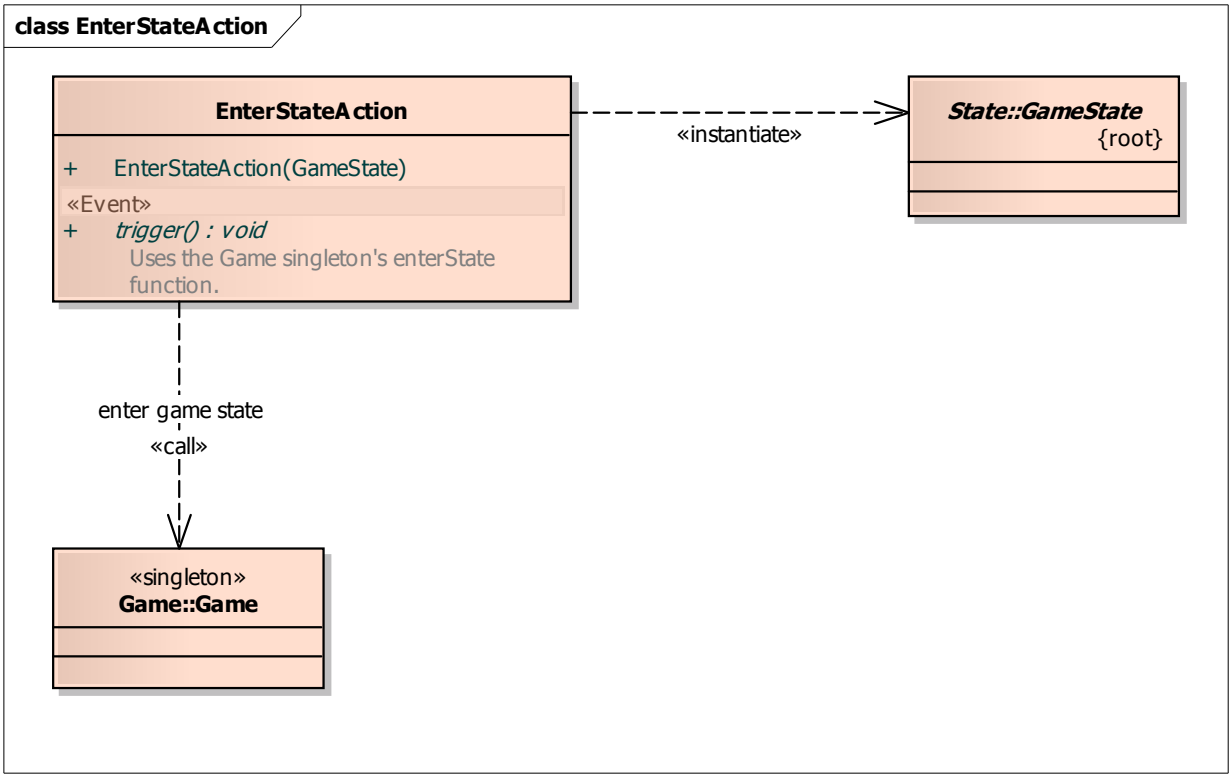


Diagram: LeaveStateAction

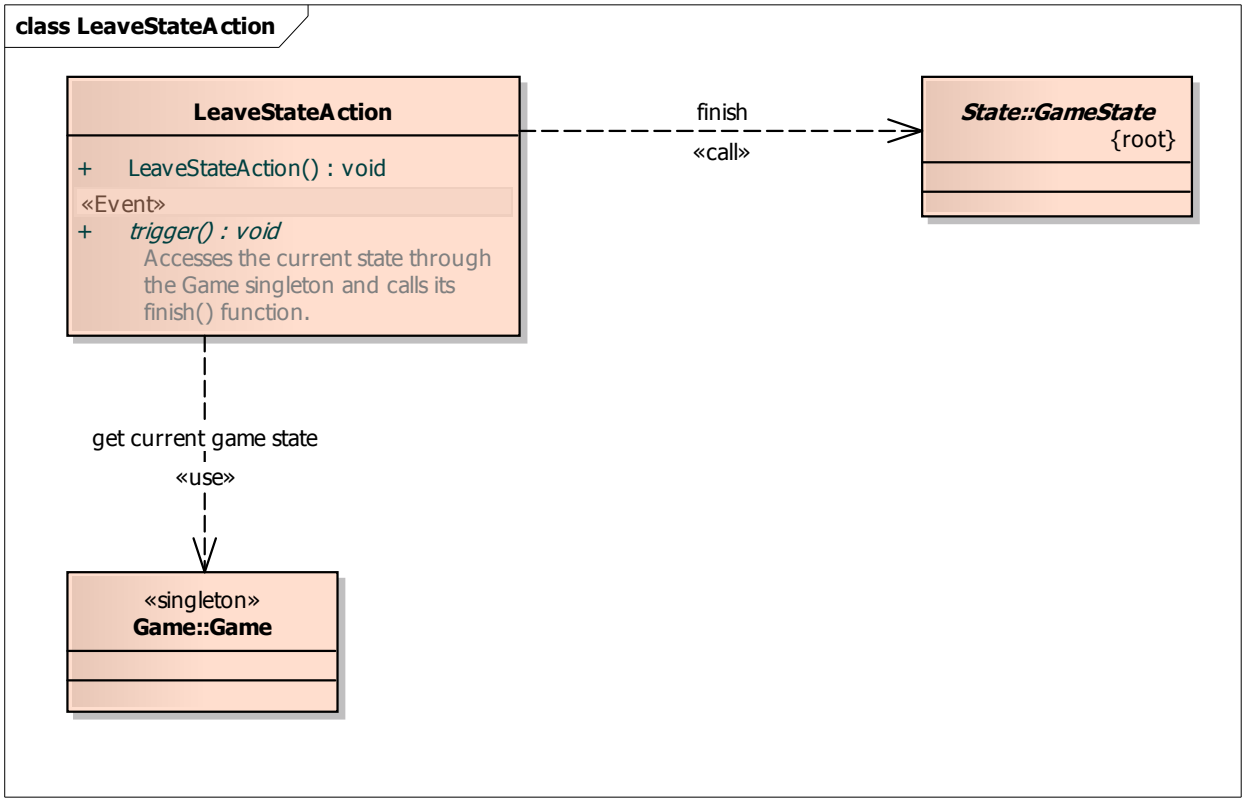


Diagram: Menu

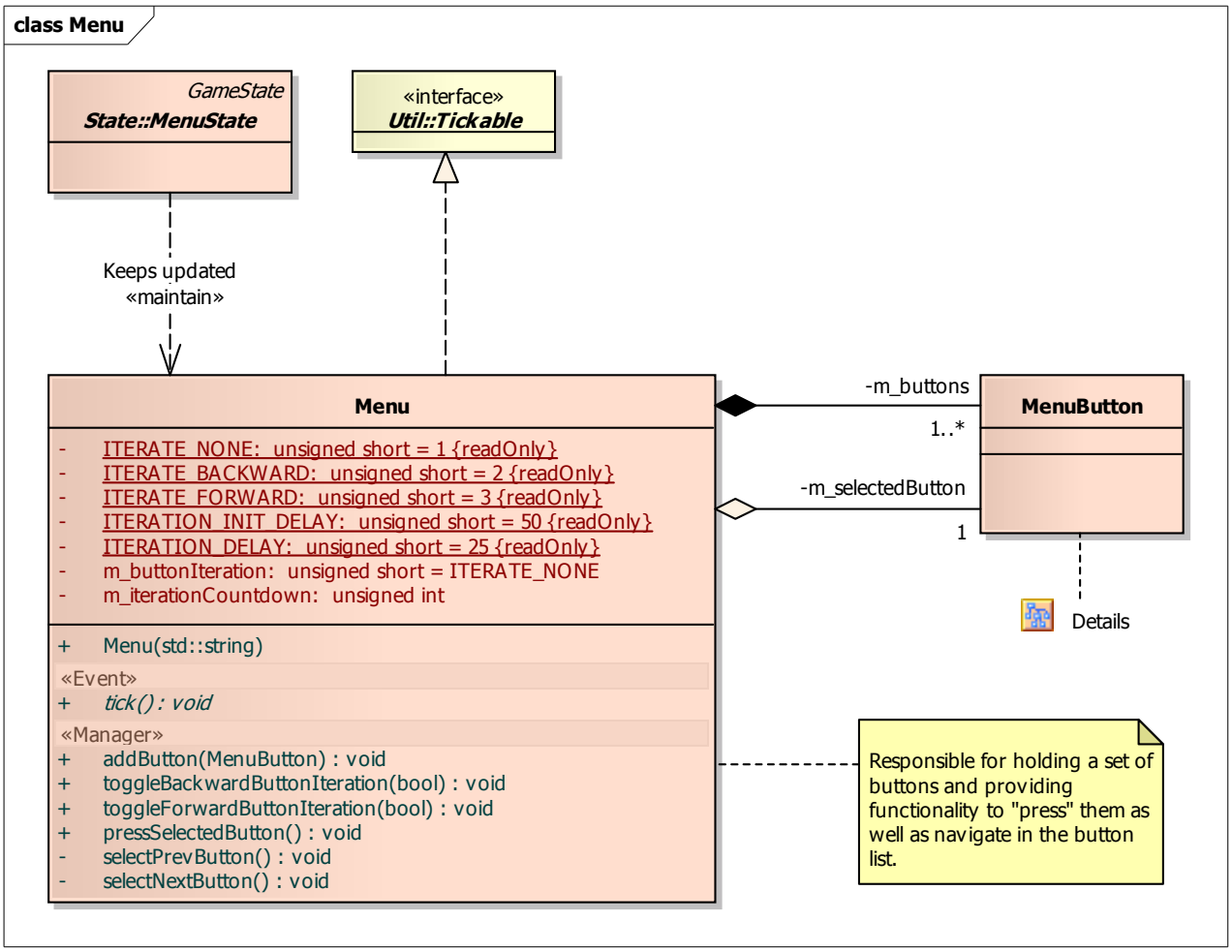


Diagram: Menu Button

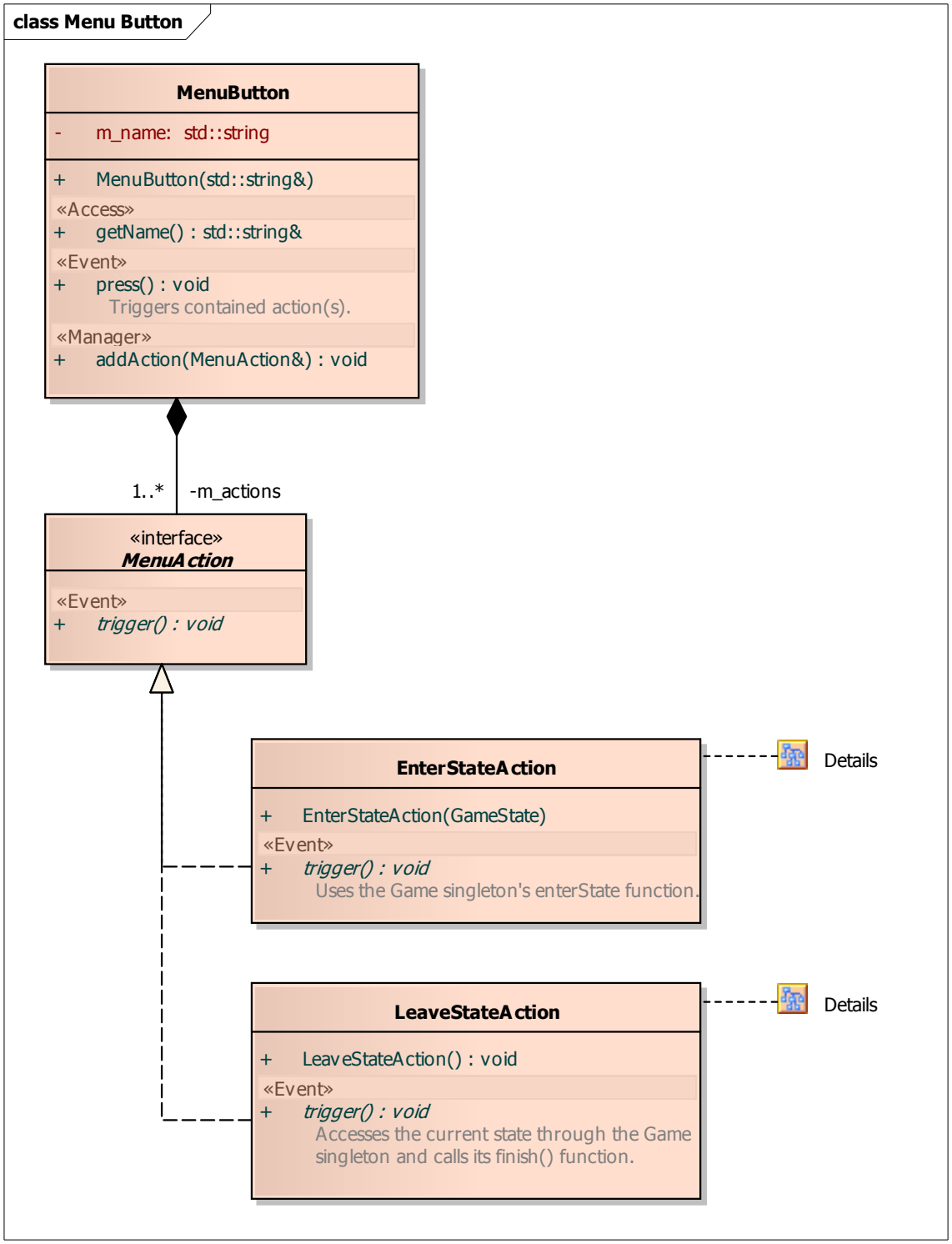


Diagram: FinishStateControl

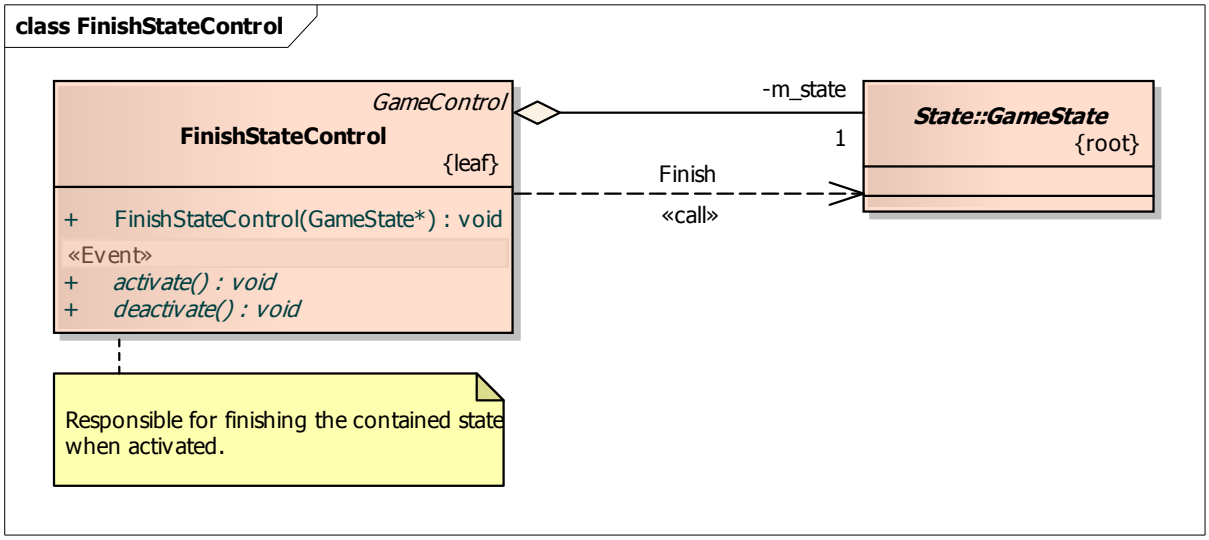


Diagram: Game Controls

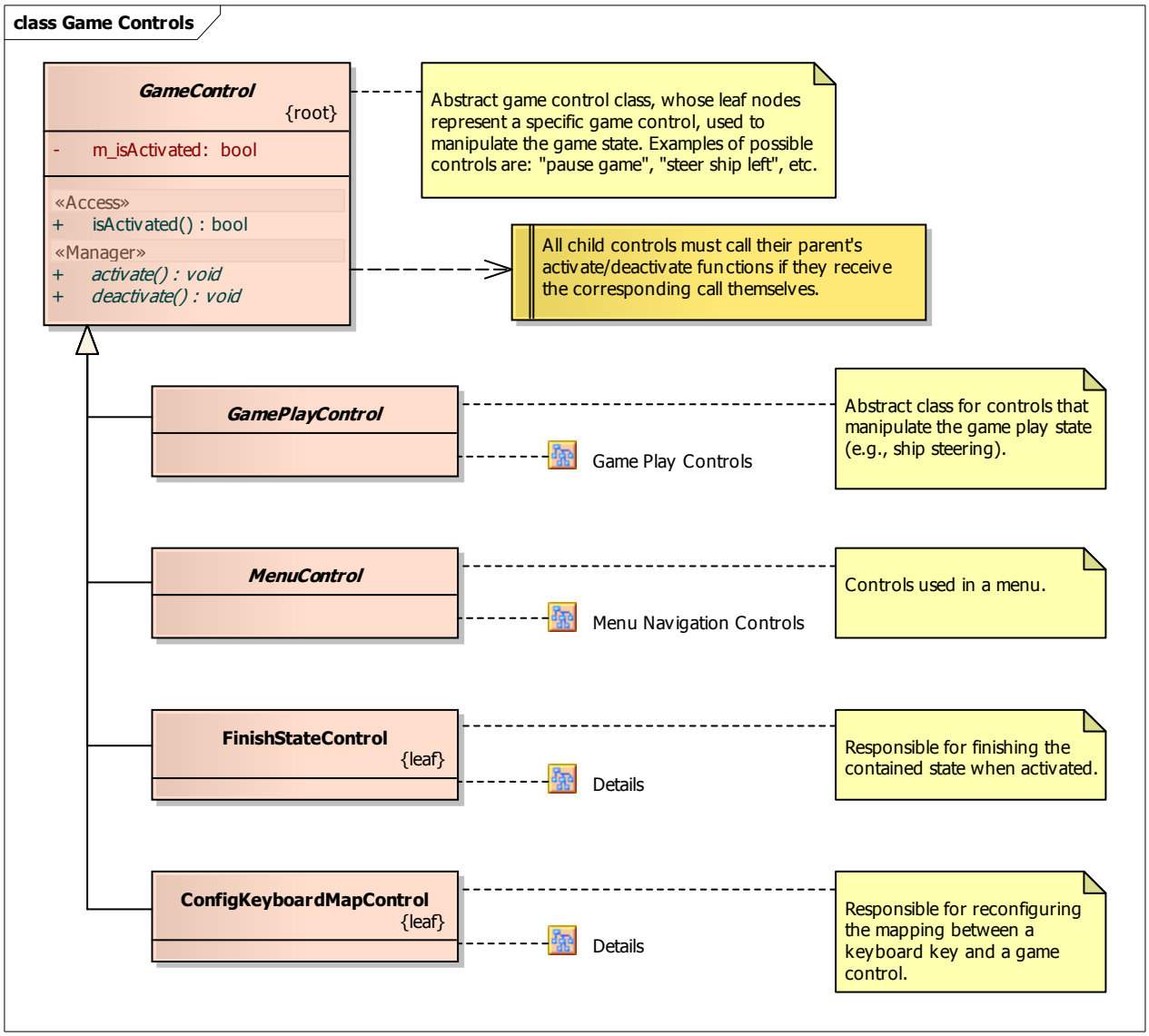


Diagram: Game Controls - Game Play

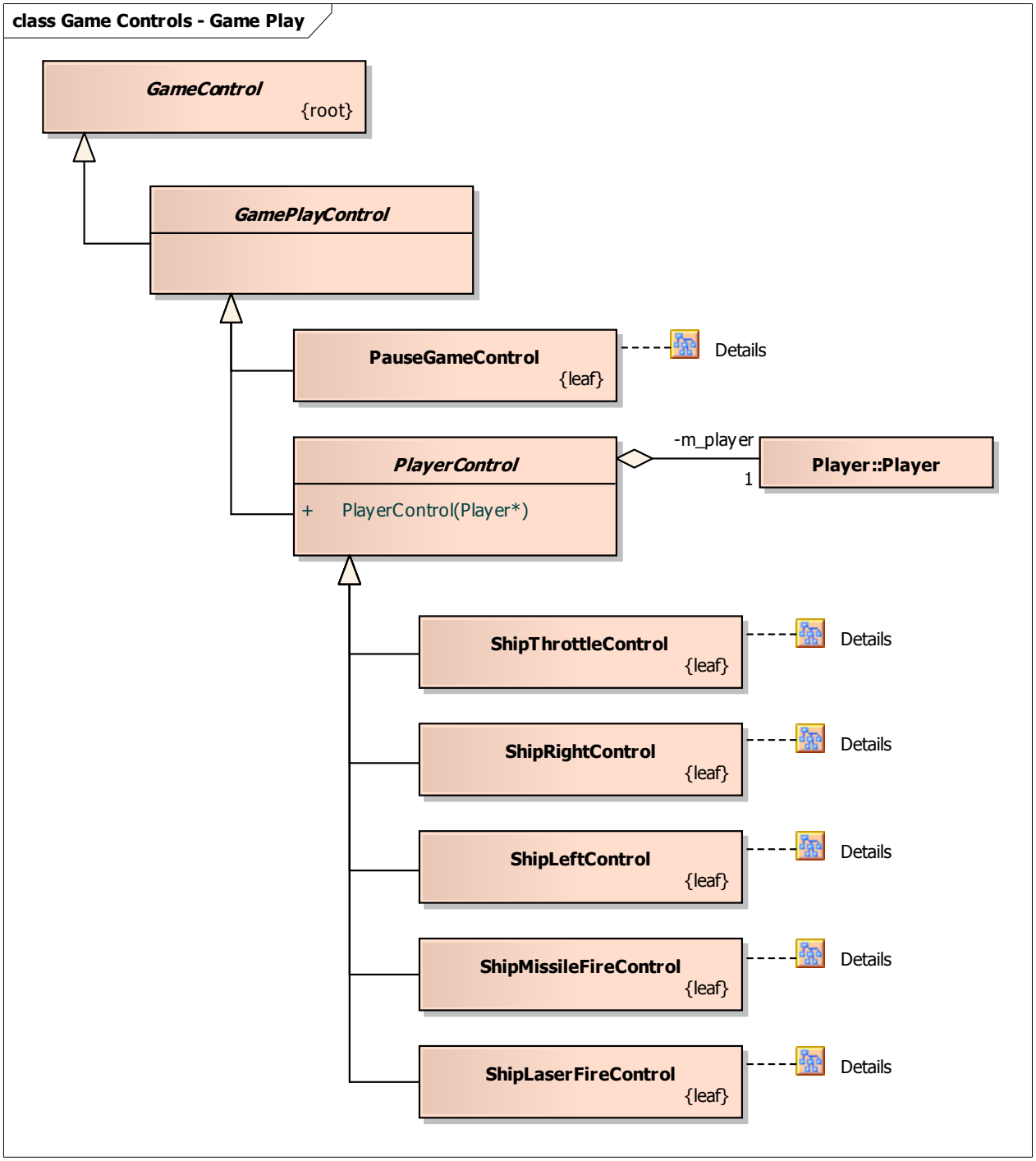


Diagram: Game Controls - Menu Navigation

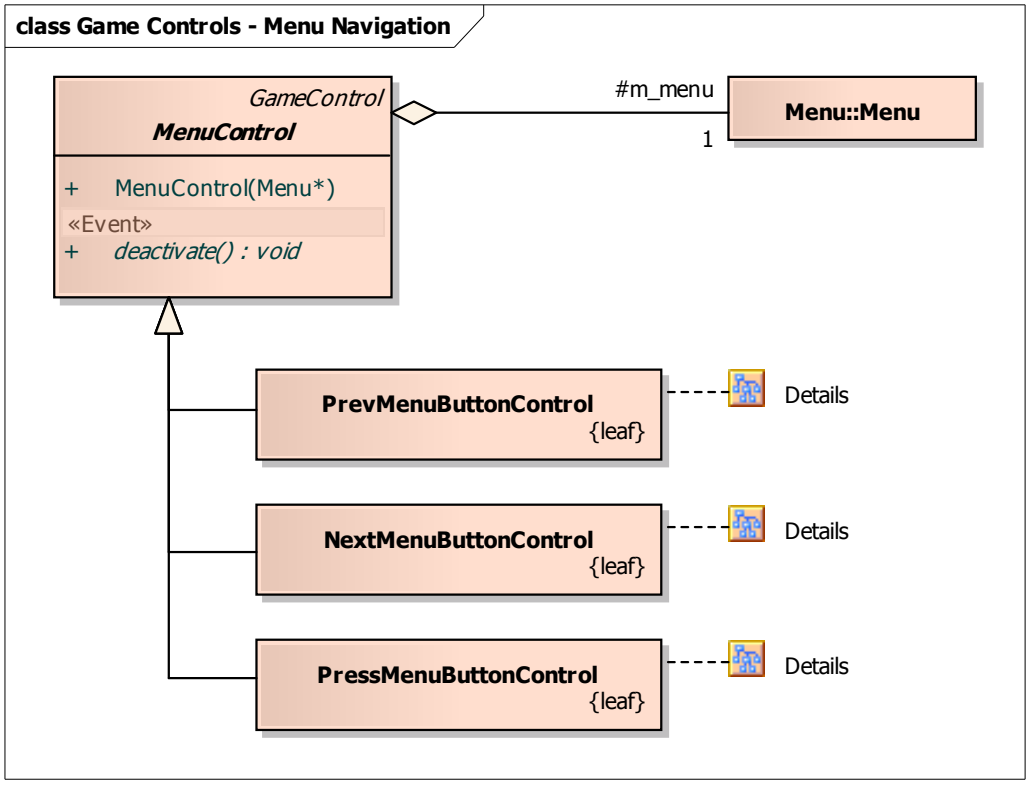


Diagram: MenuNavigateDownAction

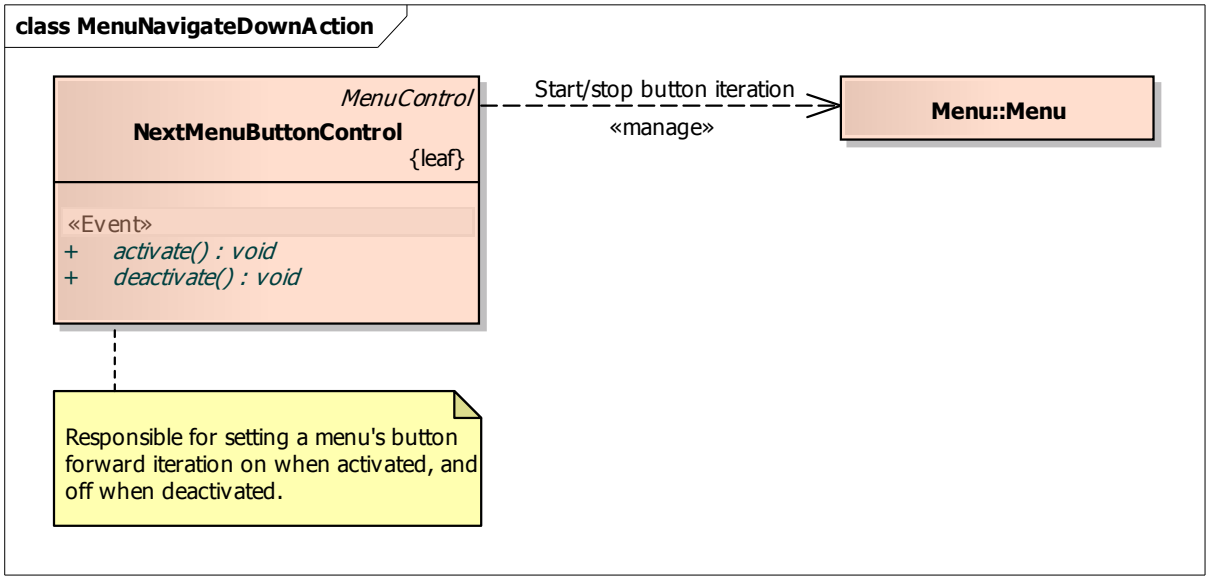


Diagram: MenuNavigateUpAction

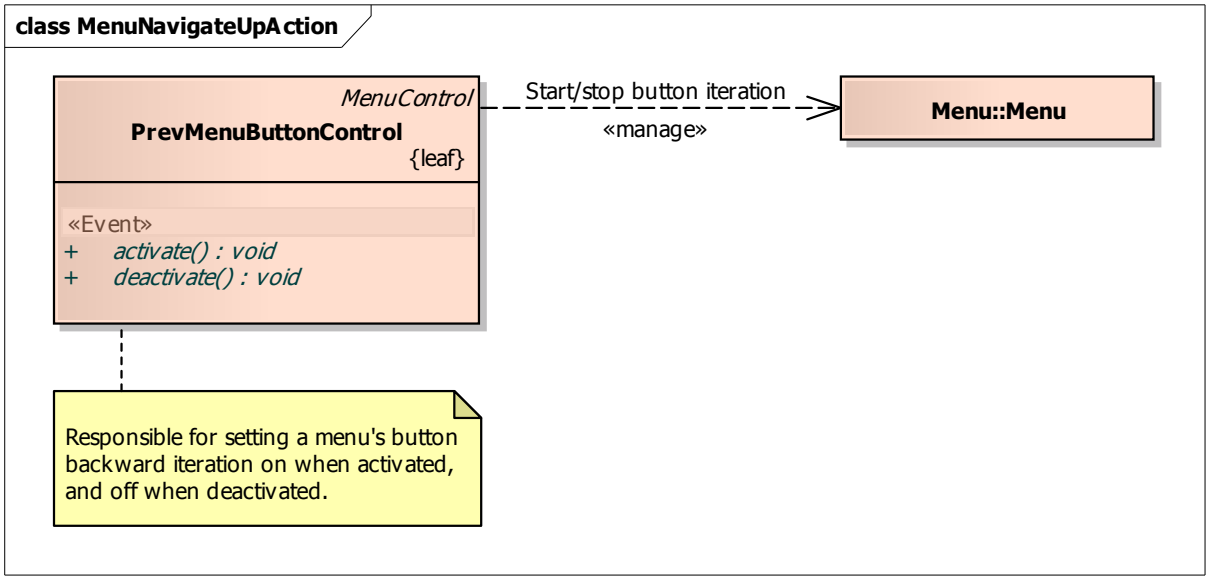


Diagram: MenuPressAction

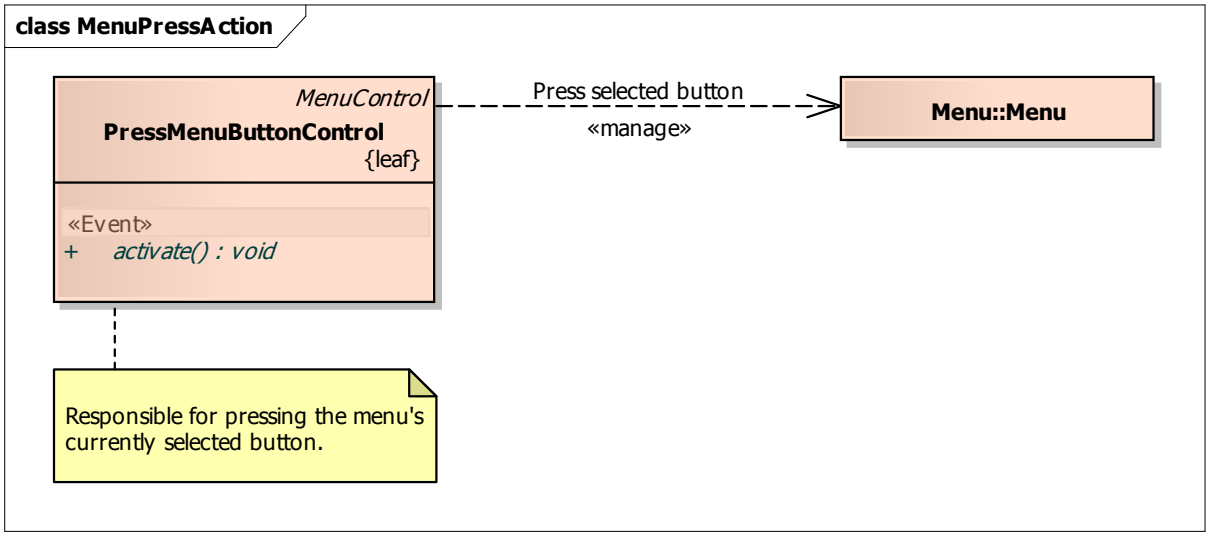


Diagram: PauseGameControl

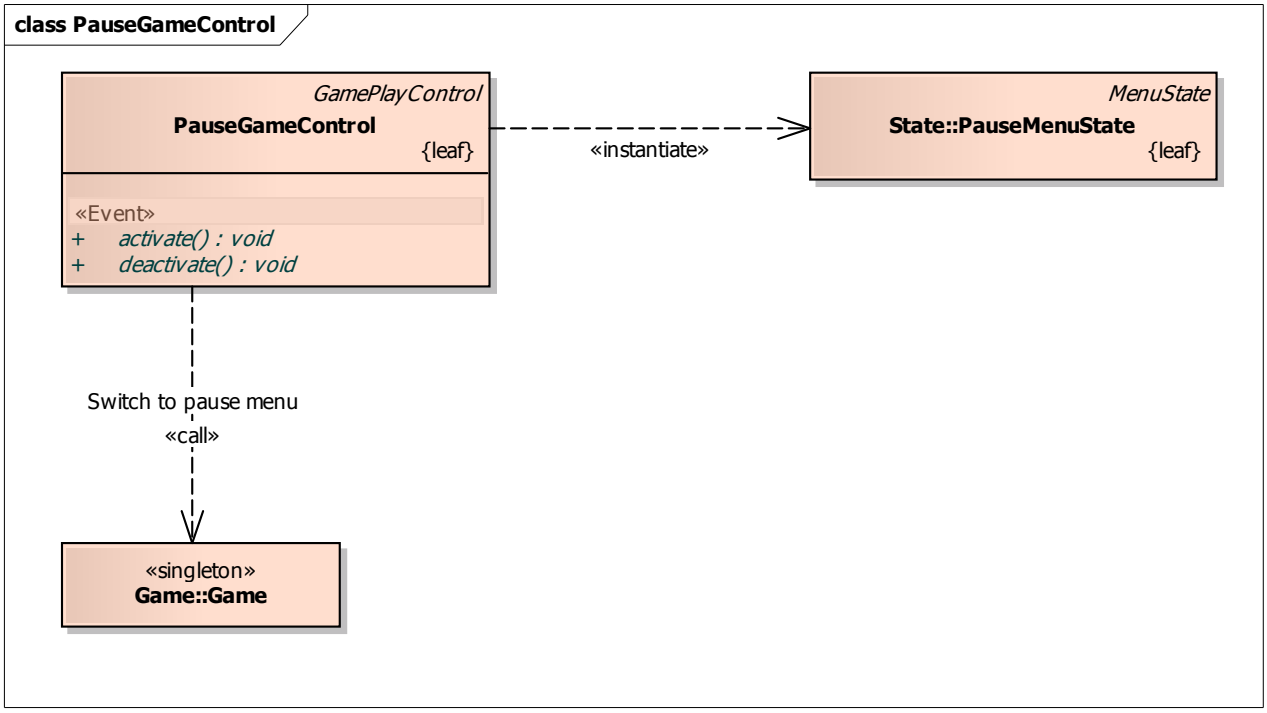


Diagram: ReconfigKeyboardMapControl

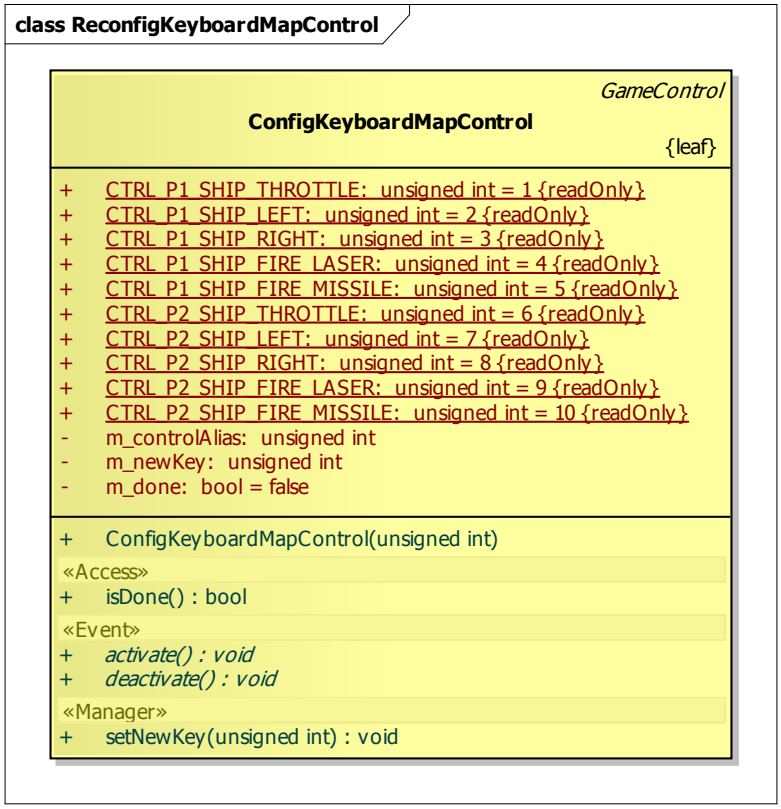


Diagram: ShipLaserFireControl

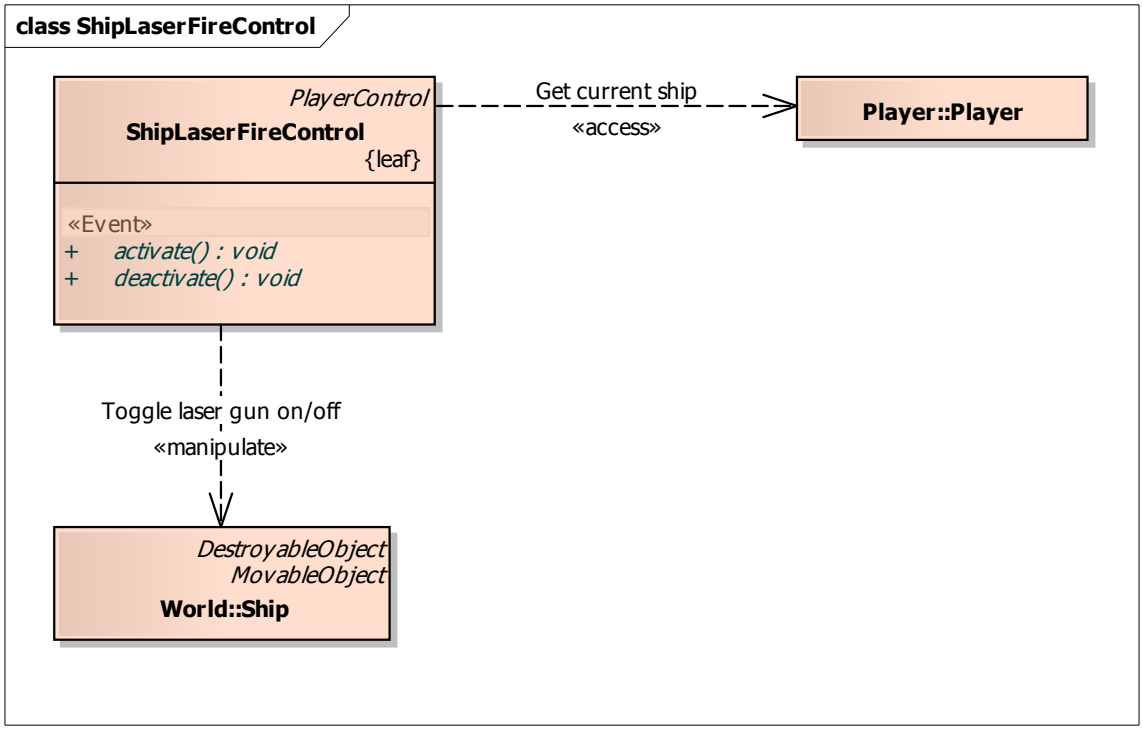


Diagram: ShipLeftControl

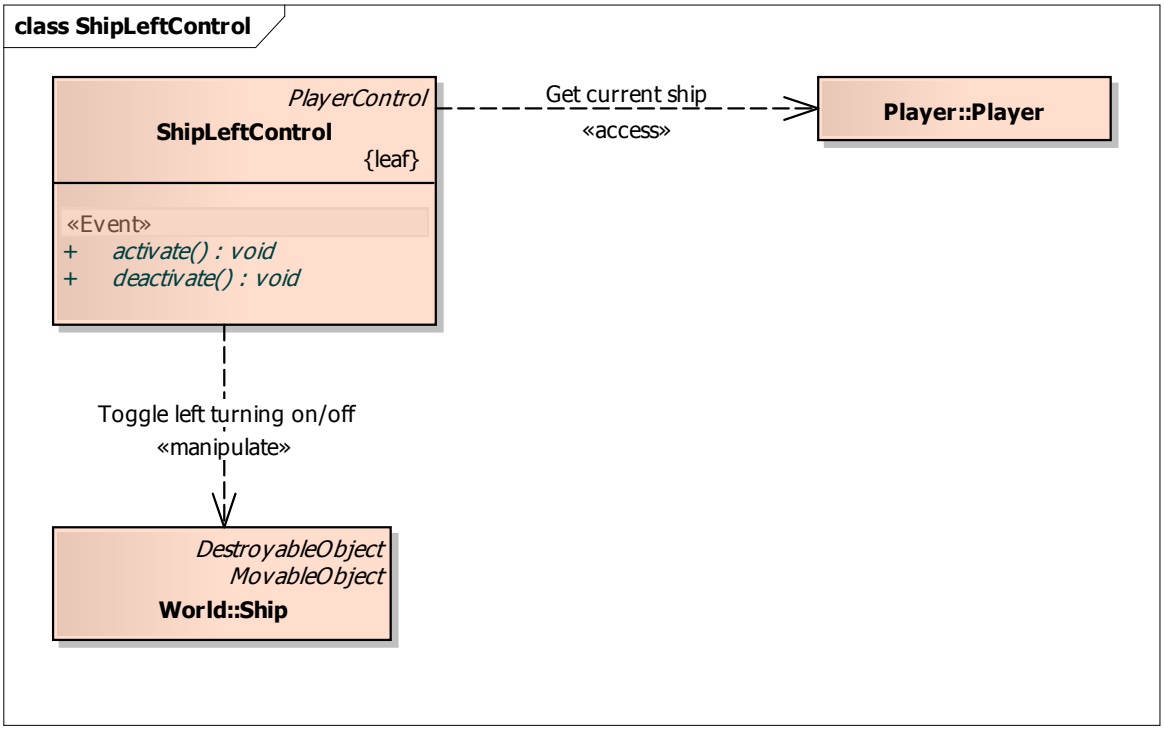


Diagram: ShipMissileFireControl

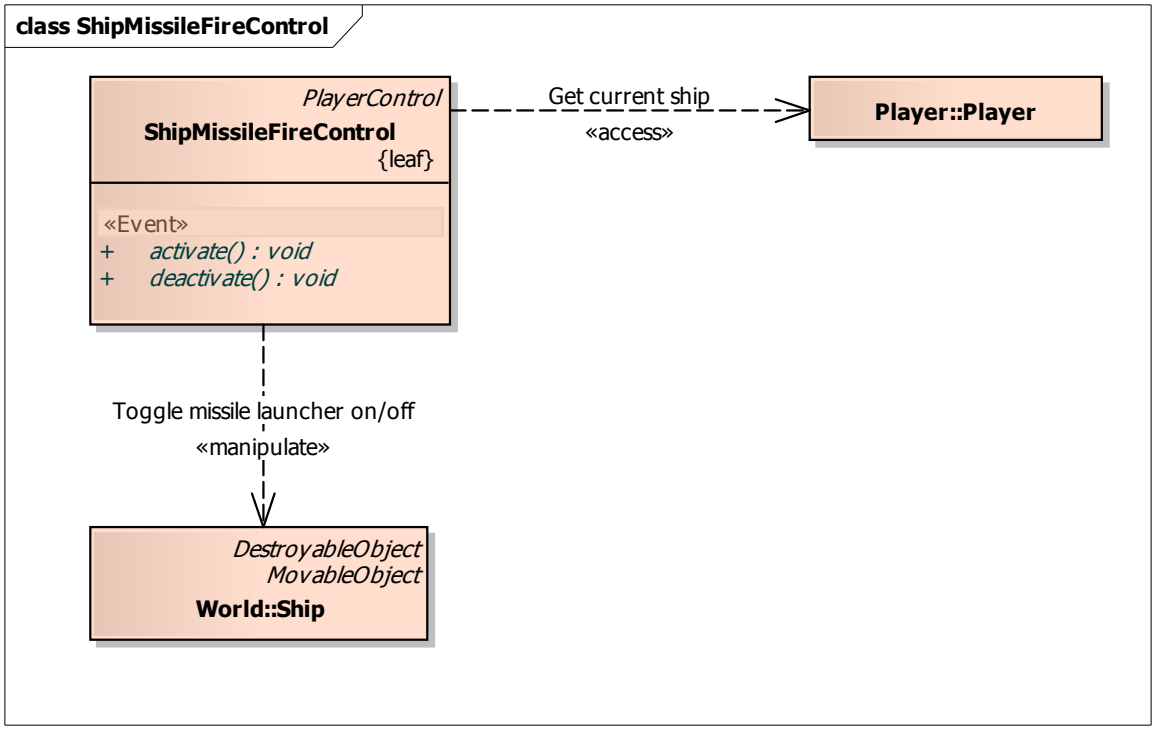


Diagram: ShipRightControl

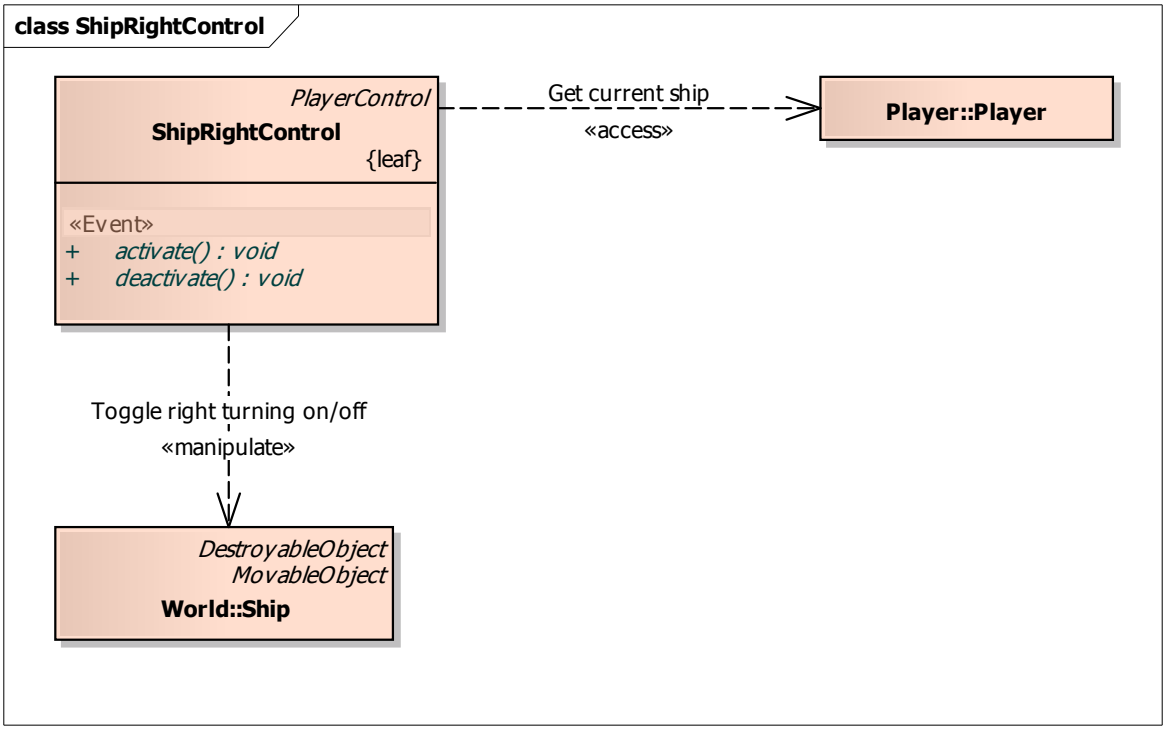


Diagram: ShipThrottleControl

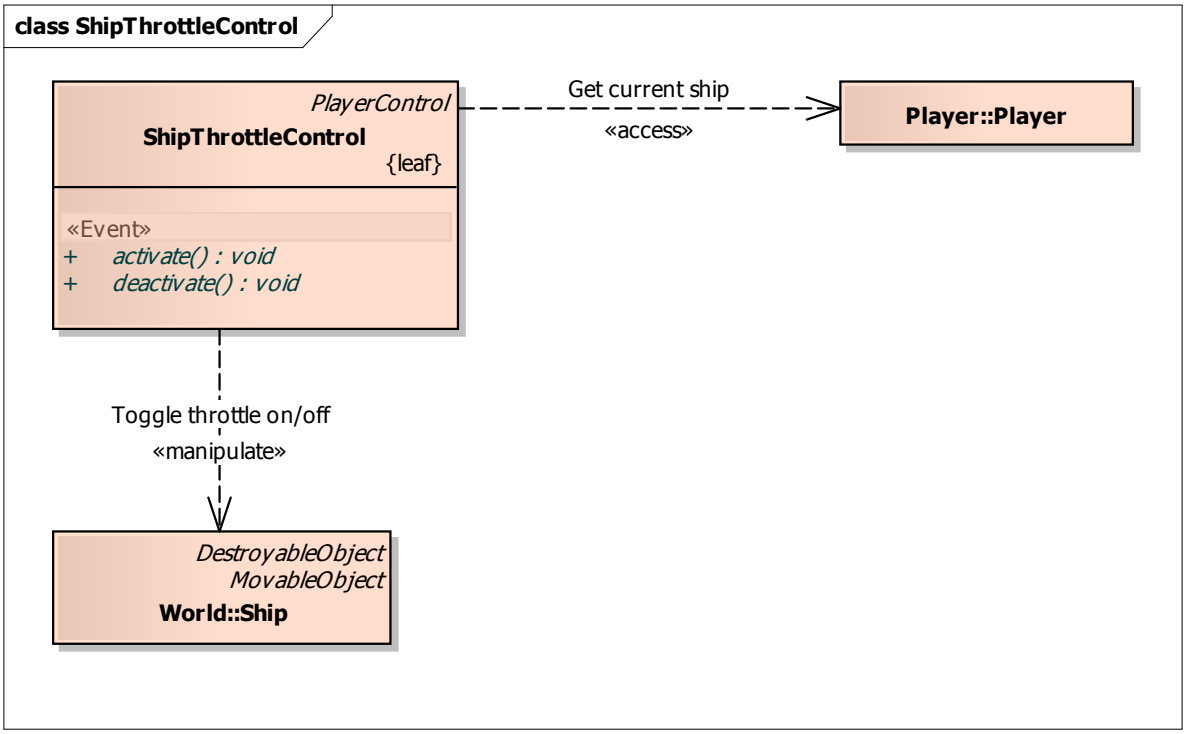


Diagram: Player

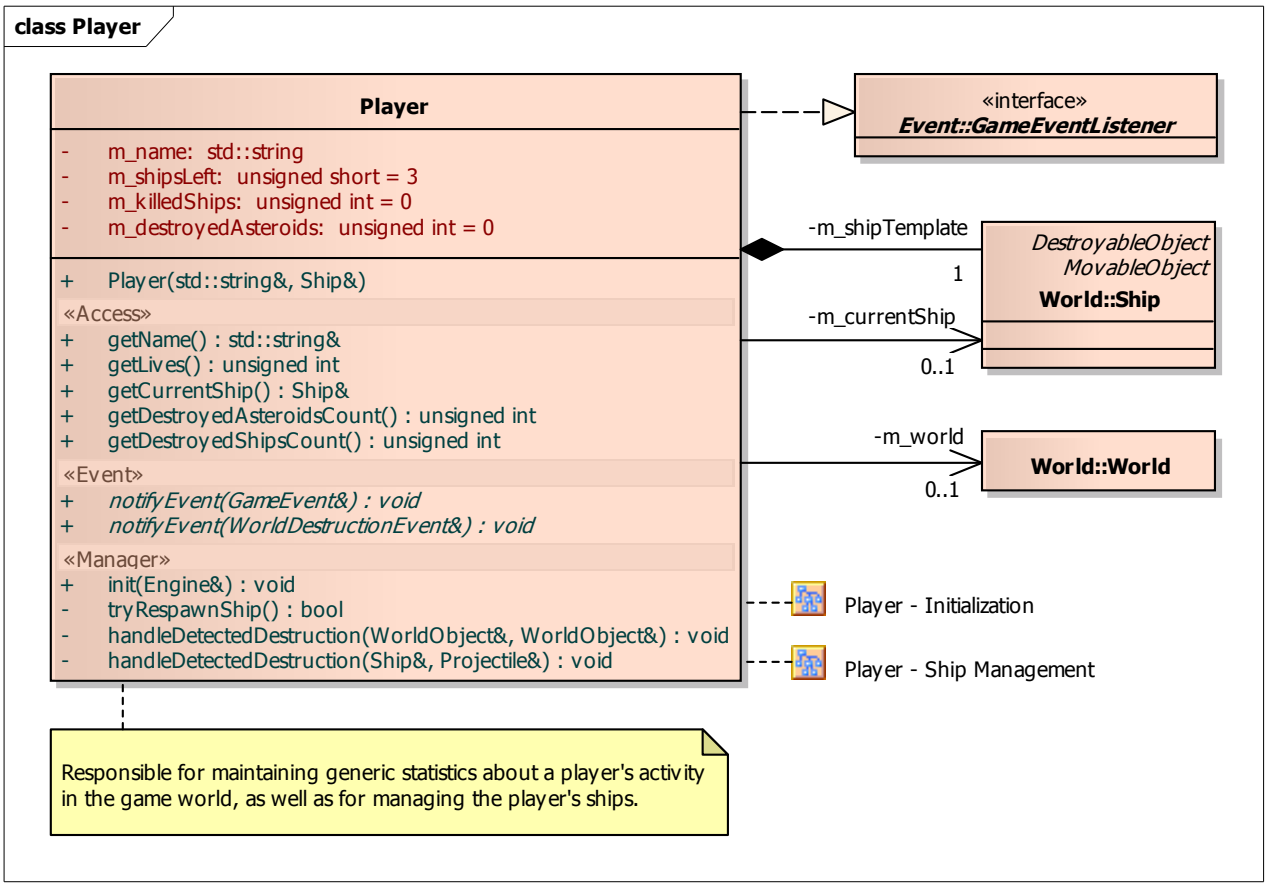


Diagram: Game Event

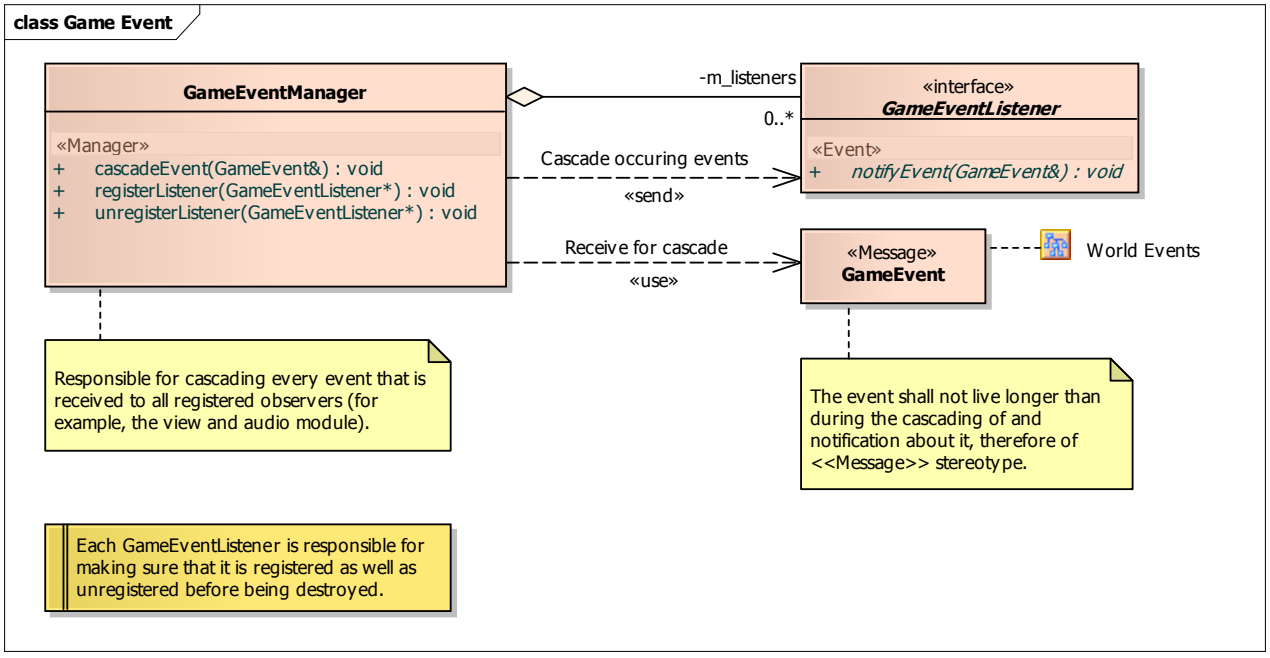


Diagram: Audio

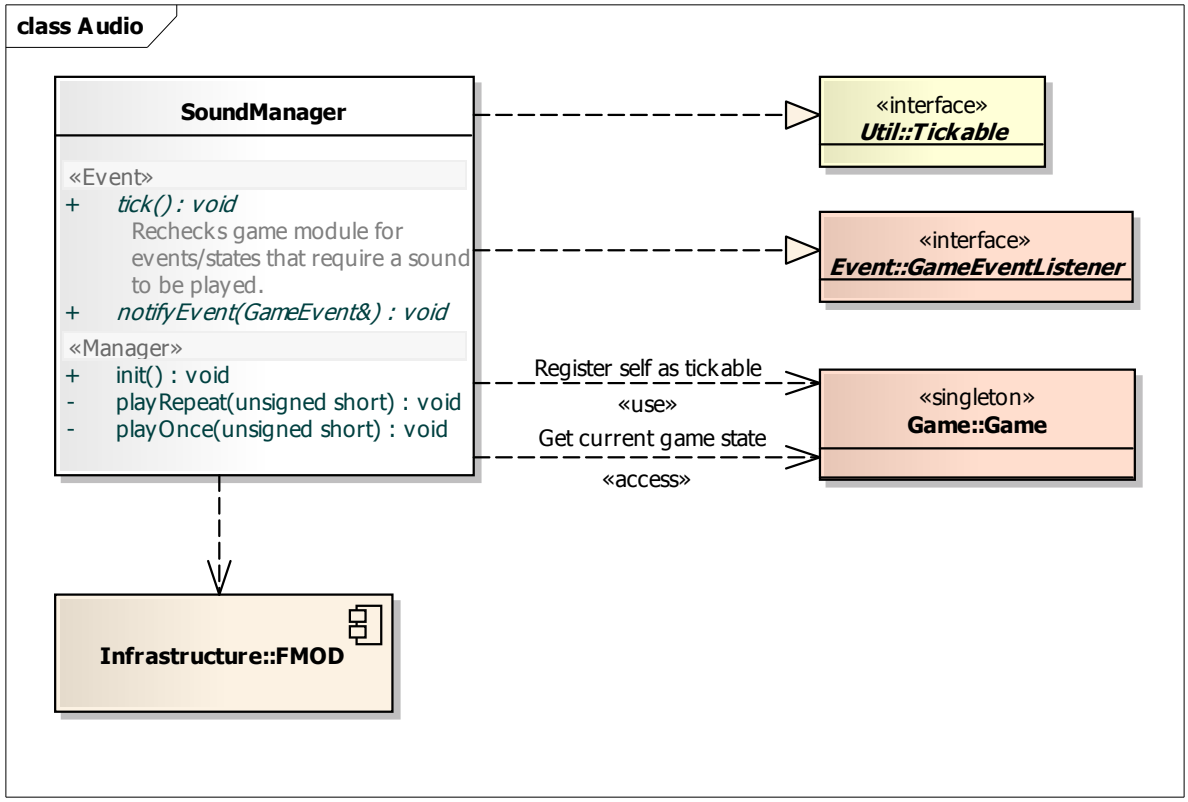


Diagram: View

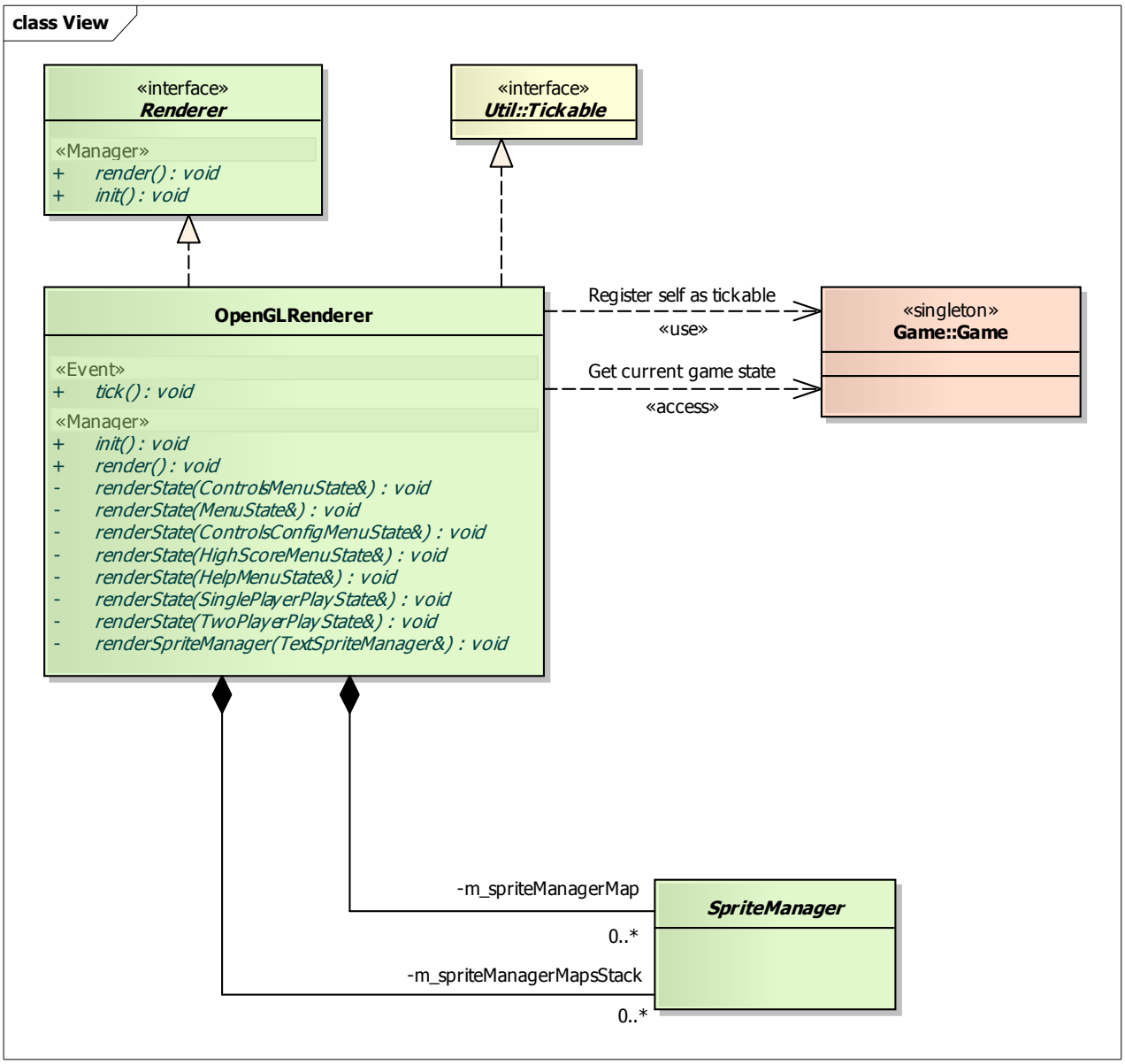


Diagram: View - Sprite Managers

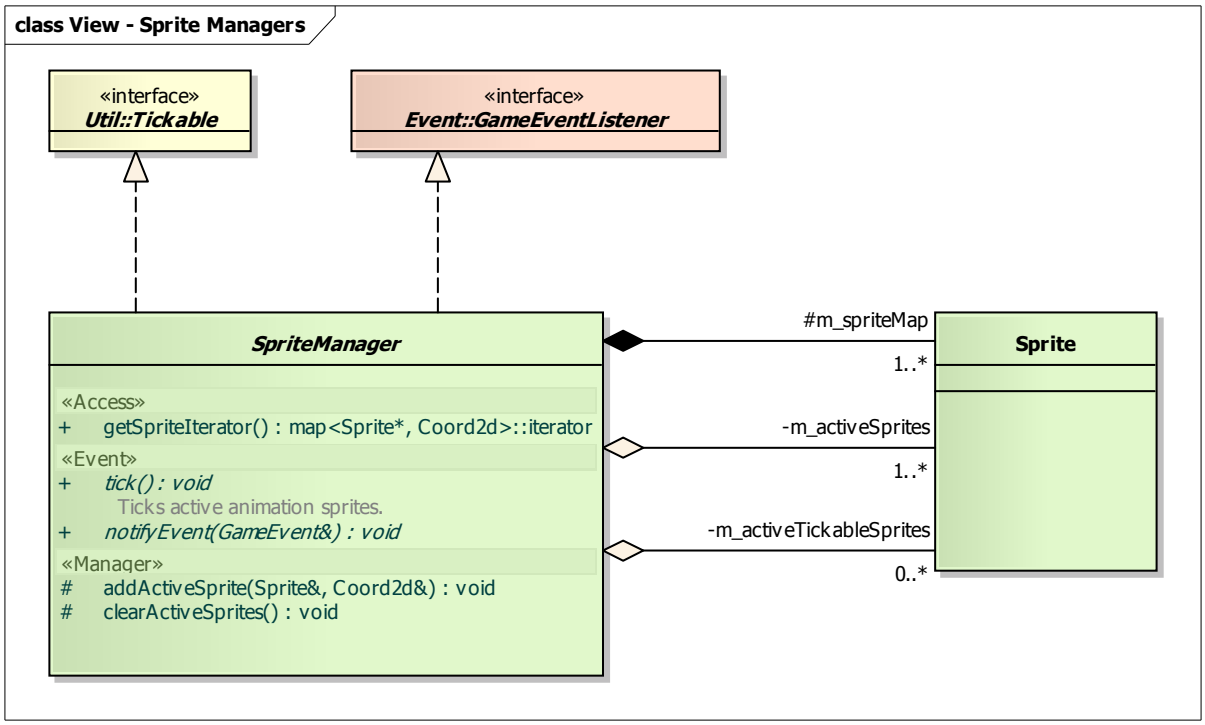


Diagram: View - Sprites

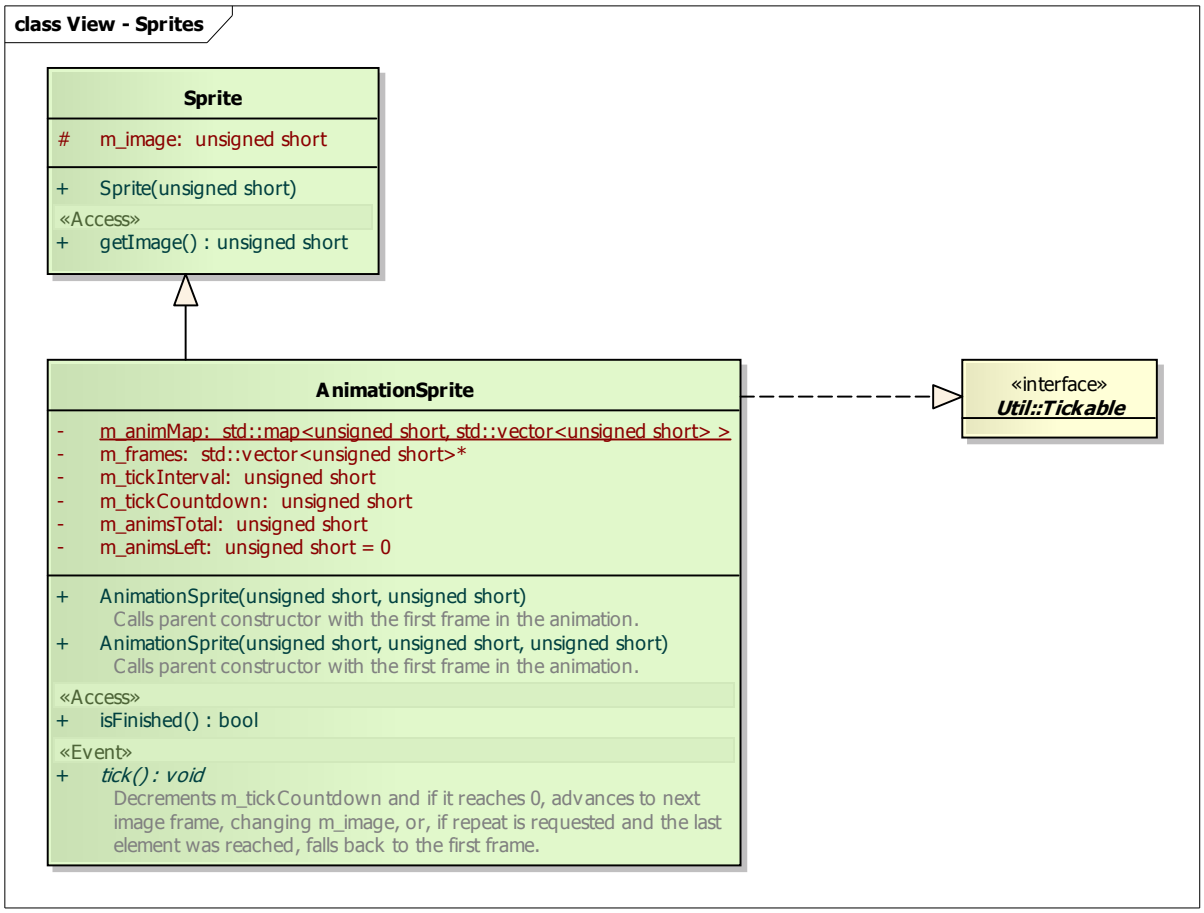


Diagram: View - World Sprite Managers

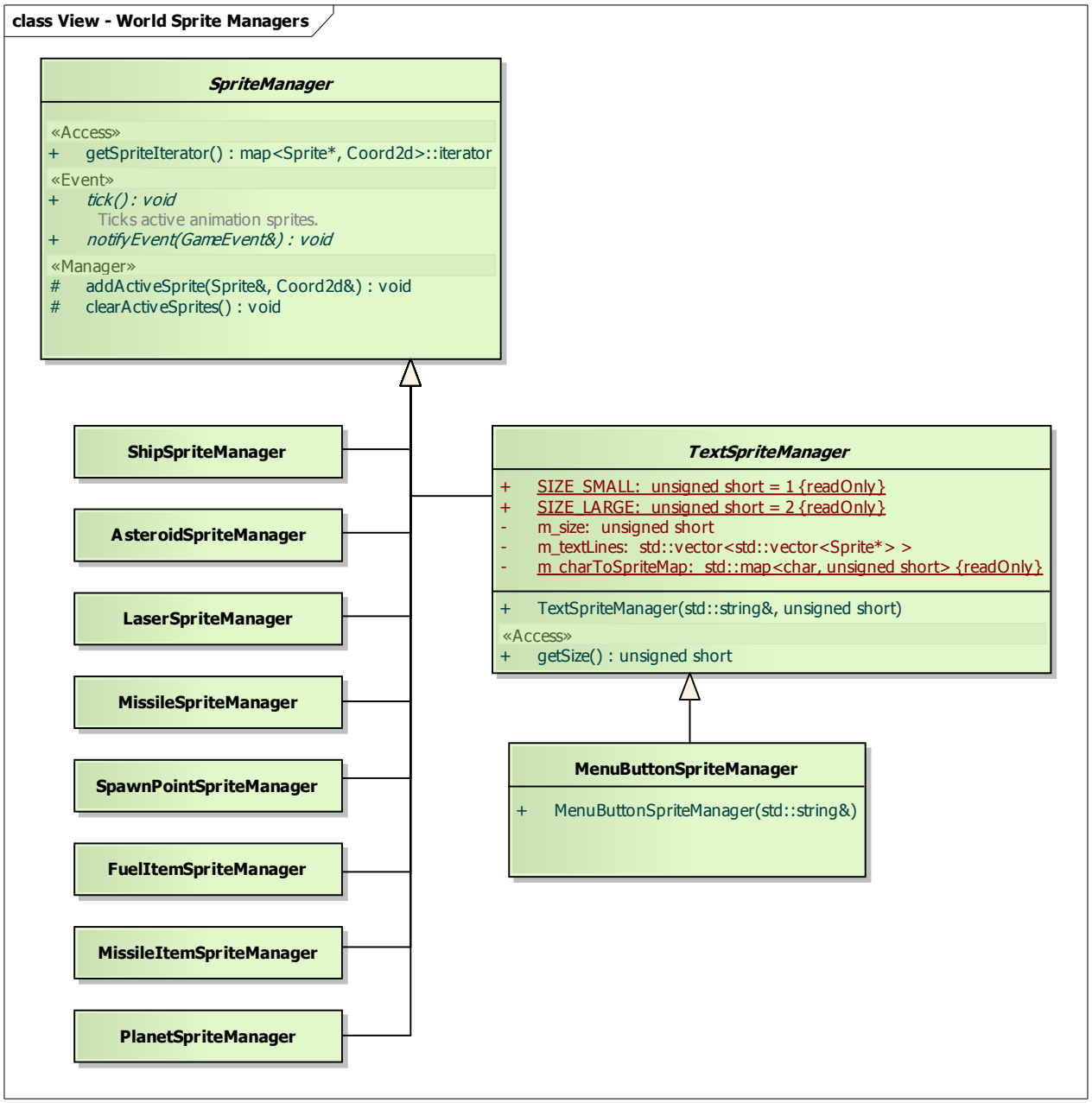


Diagram: ConfigRegistry

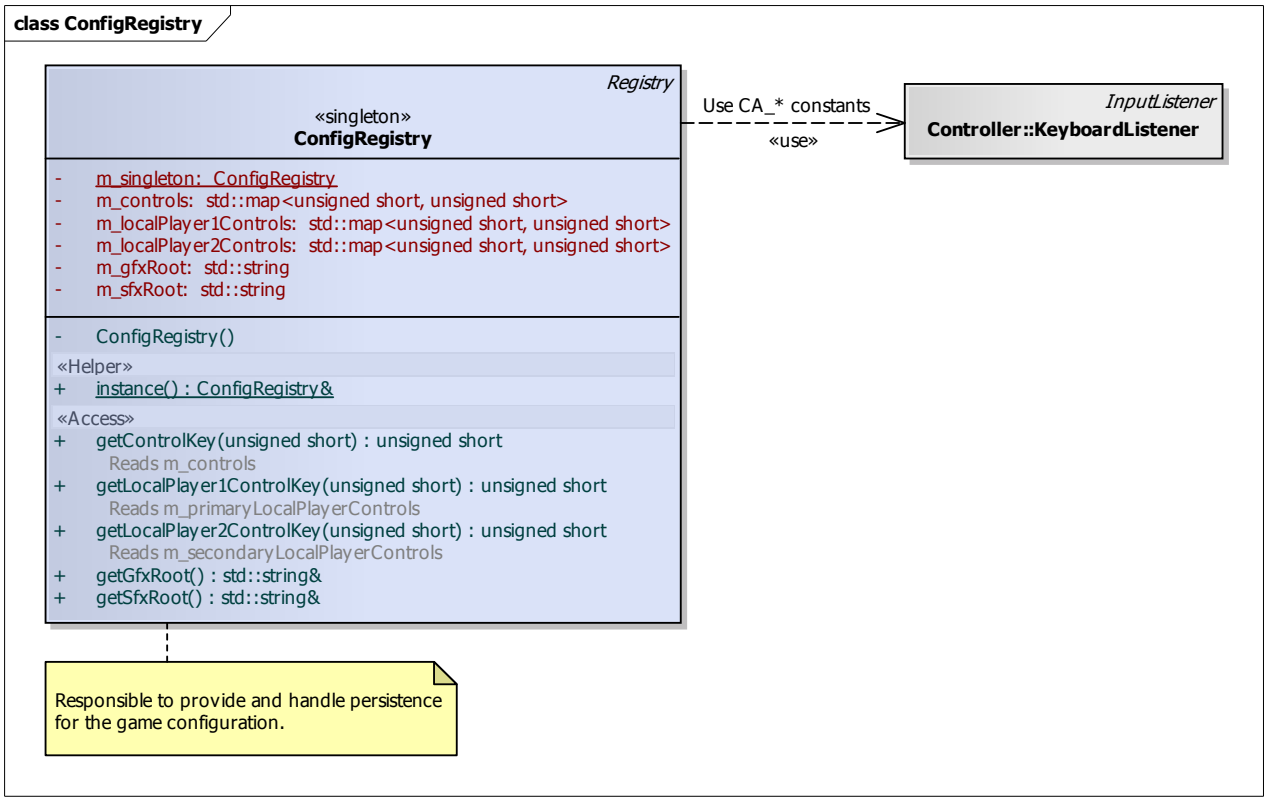


Diagram: HighScoreRegistry

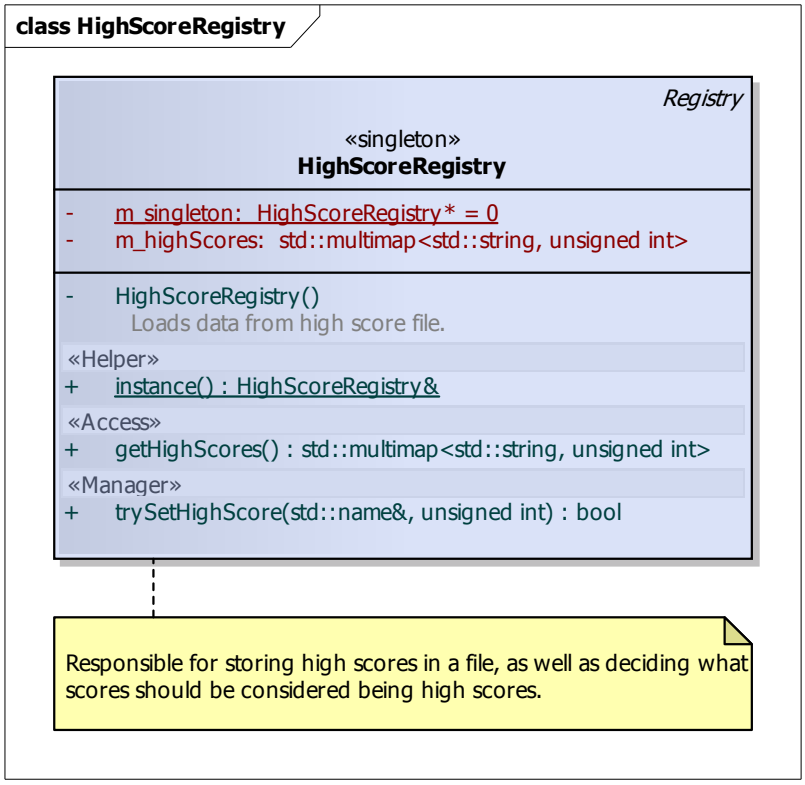


Diagram: Registry

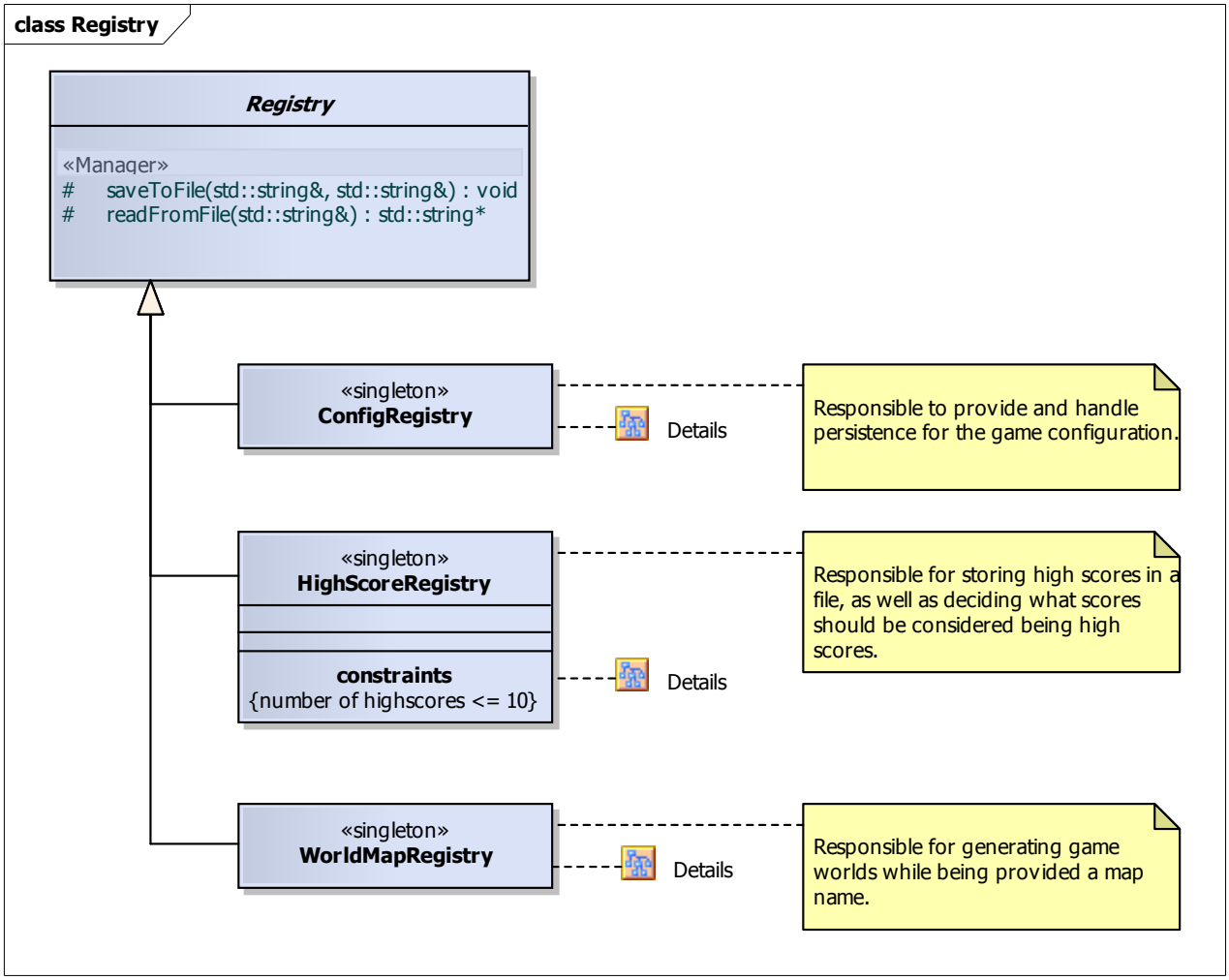


Diagram: WorldMapRegistry

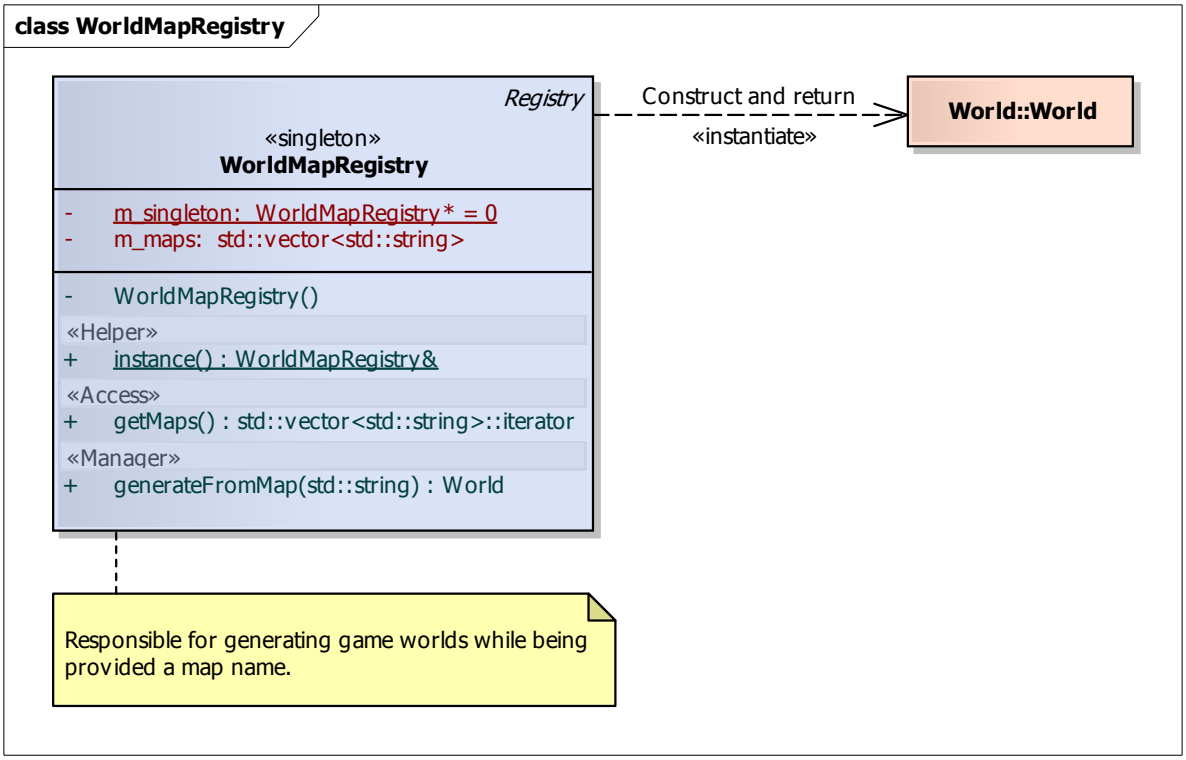
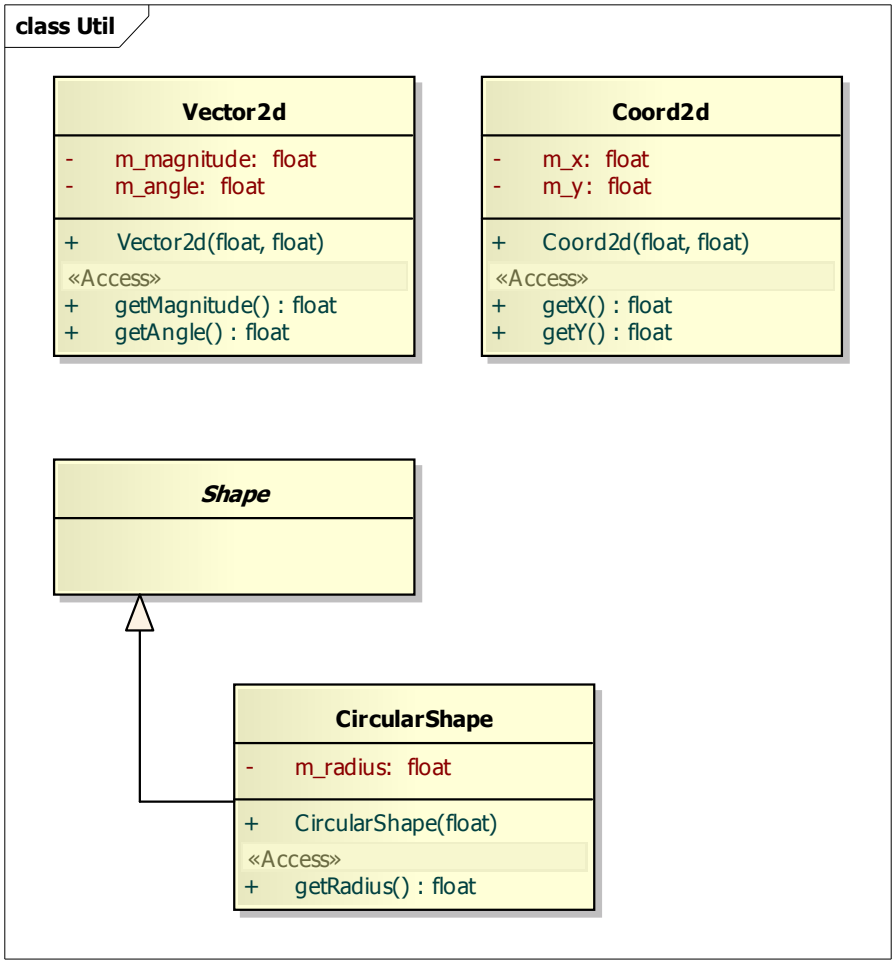


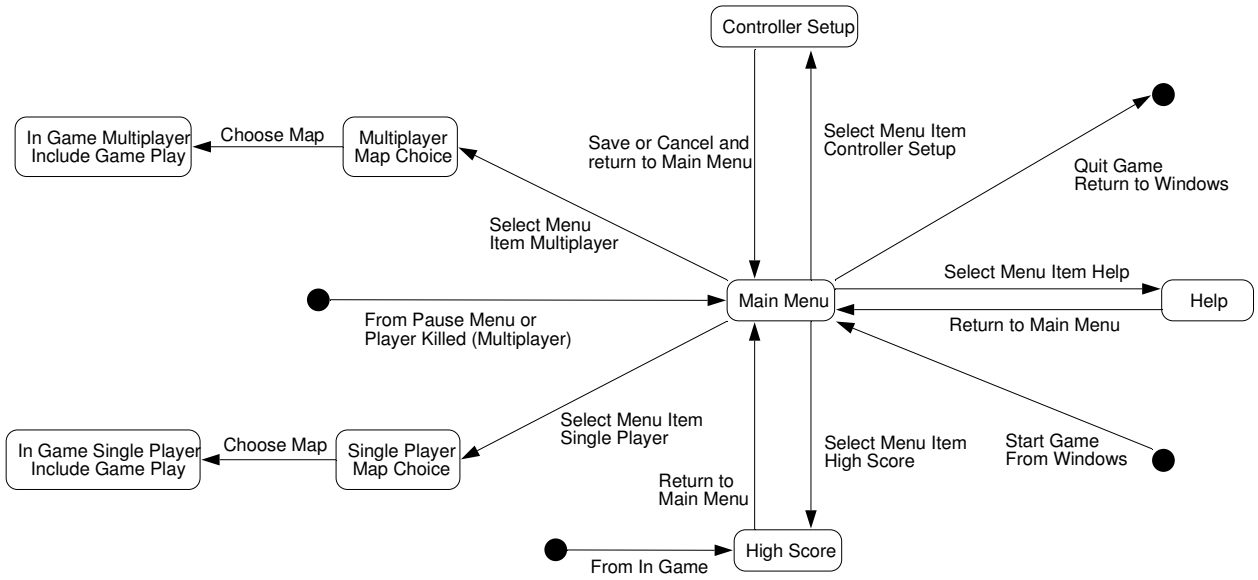
Diagram: Util



5.3 State Charts

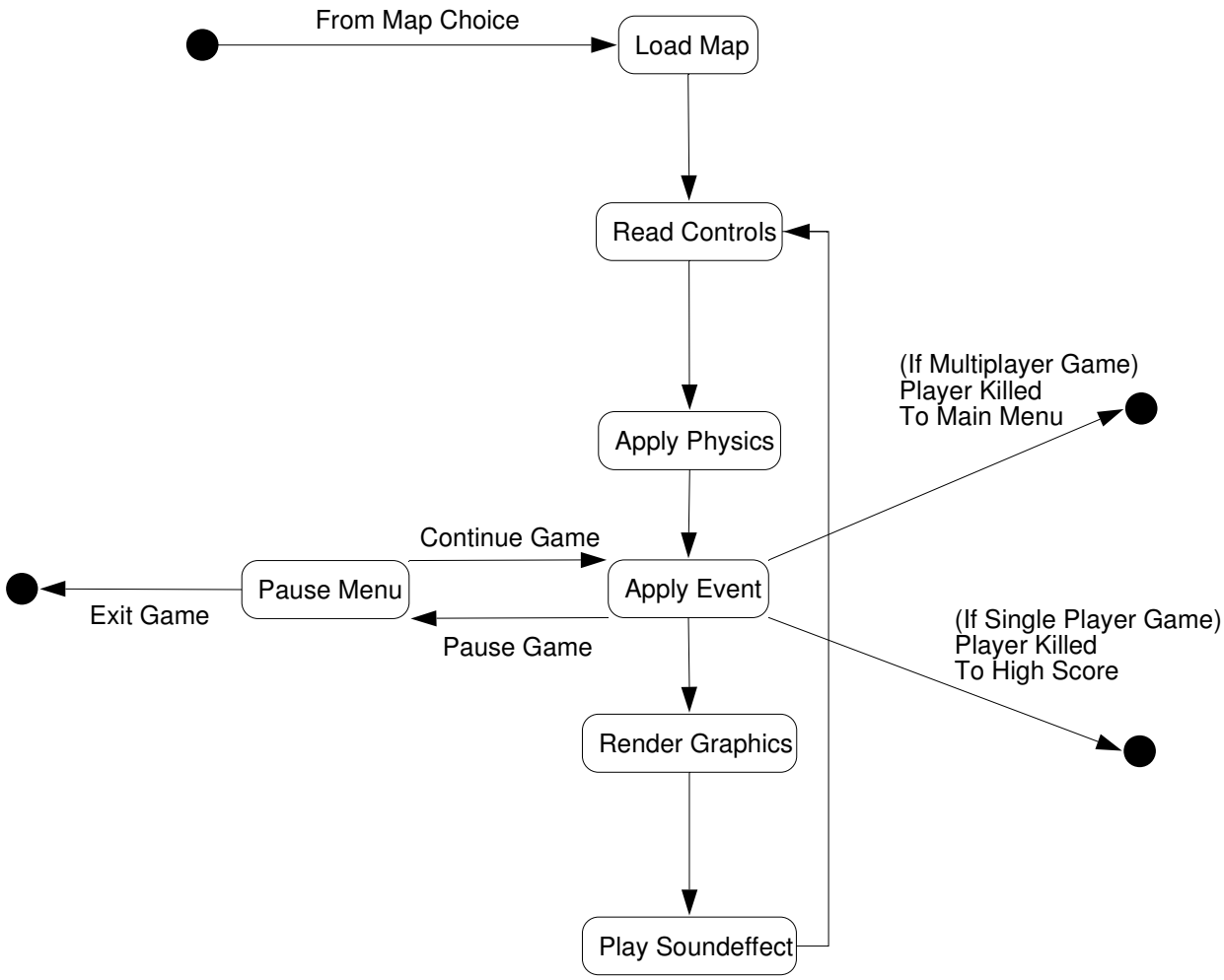
We have two main flow of control in this game. The first part is before starting the actual game and navigating the menus. The second part is for what happens while actually playing the game.

5.3.1 Pre-Game Flow



From these menus it's possible to see your high score, get help and setup keyboard for how to control the game. In the menus it's possible to say how you want to play the game, such as what map to play at and also if you want to play multiplayer or single player game.

5.3.2 In-Game Flow



While playing the game there is a main loop that takes care of reading the keyboard, calculating game physics, playing sound and applying game logics.

5.4 Interaction Diagrams

Diagram: Controller - Using an InputManager

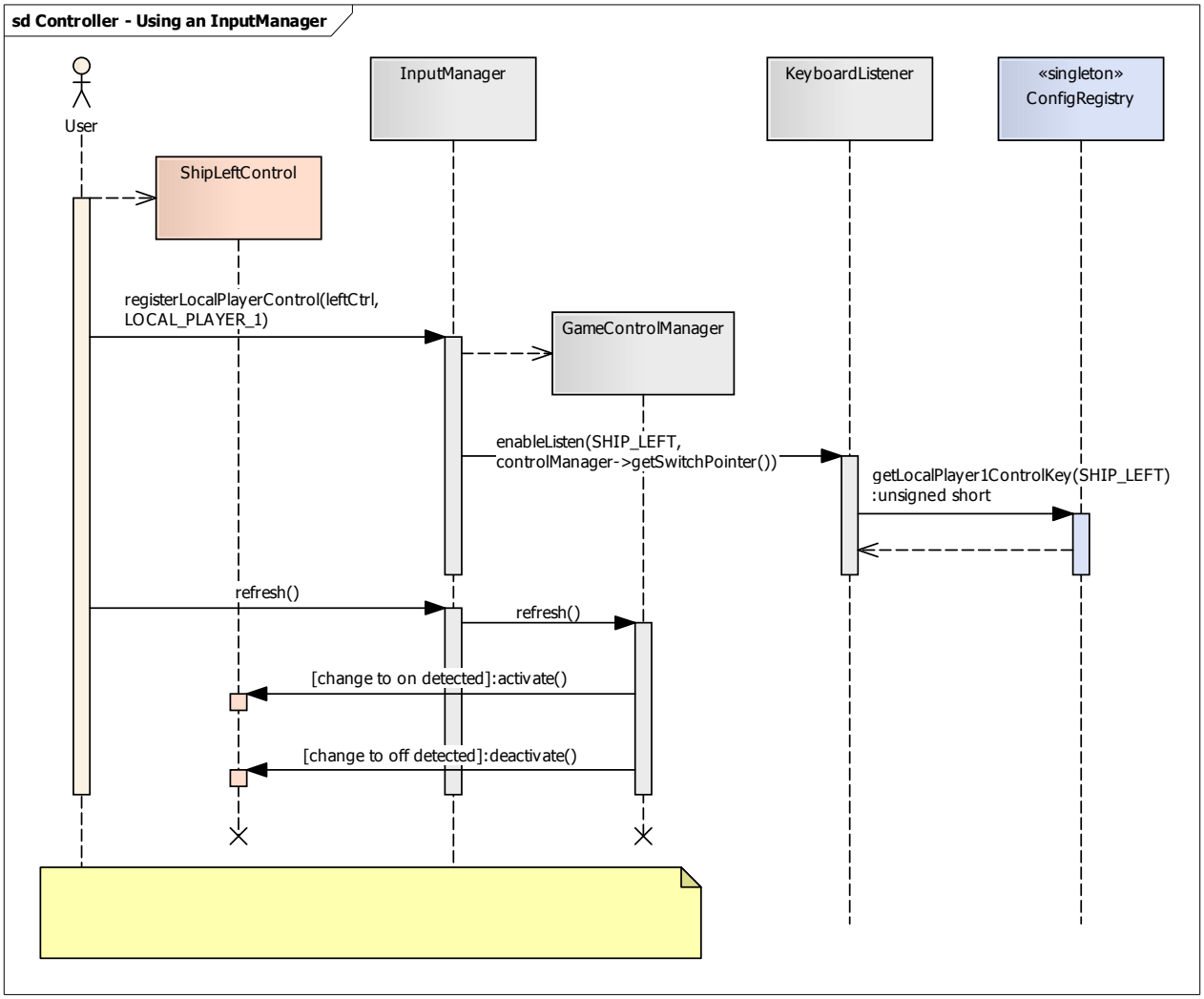


Diagram: StateSwitch

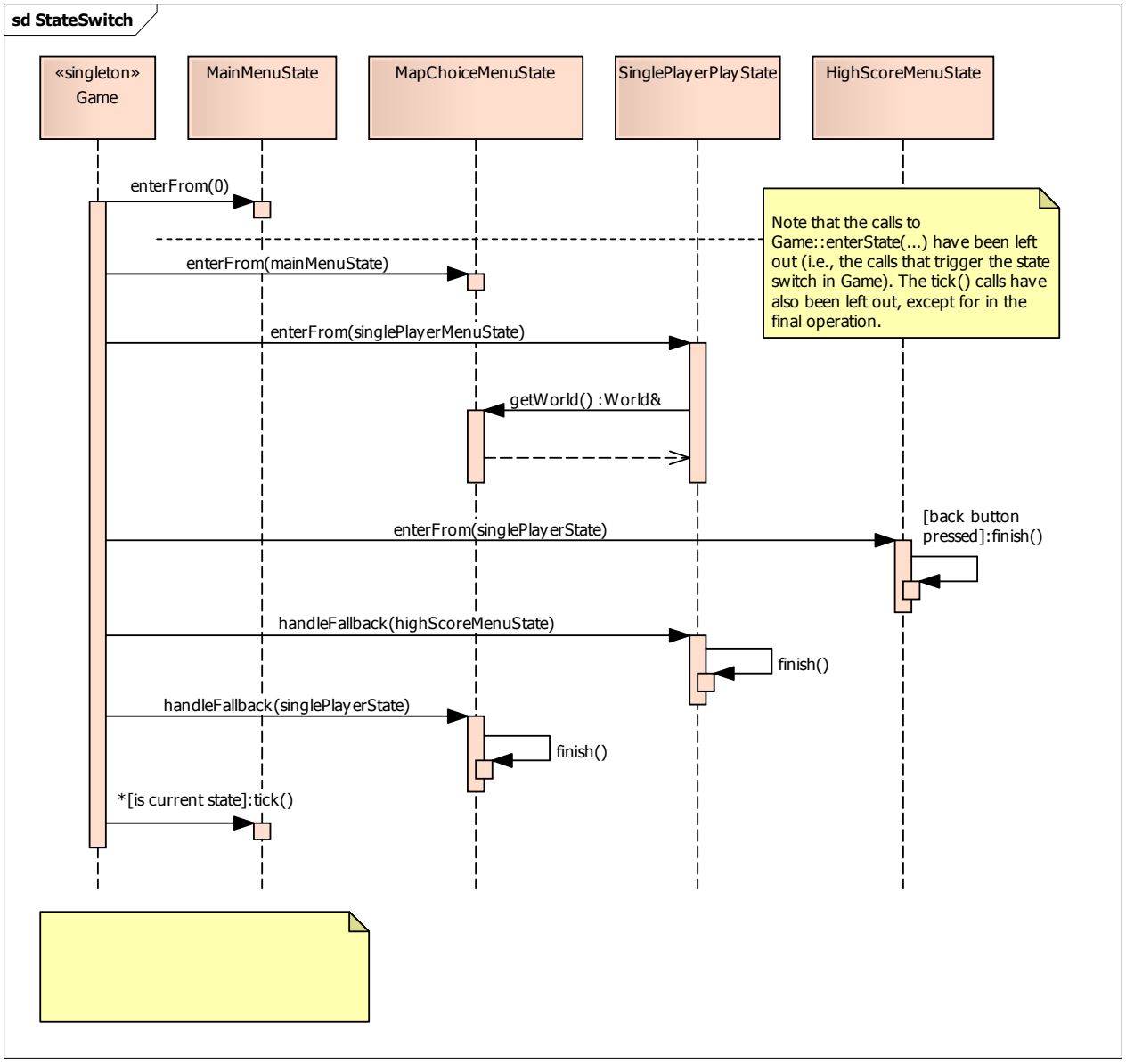


Diagram: Ship Weapon Firing

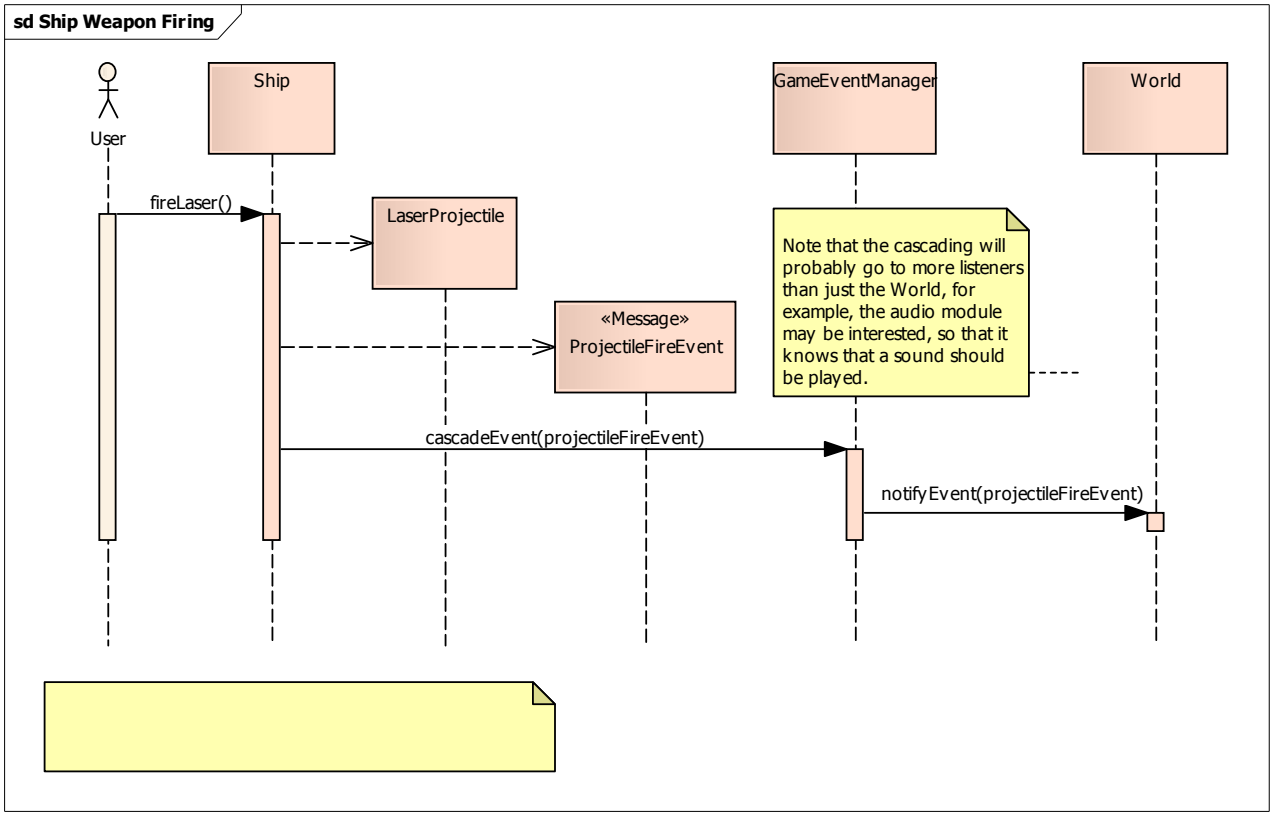


Diagram: Player - Ship Management

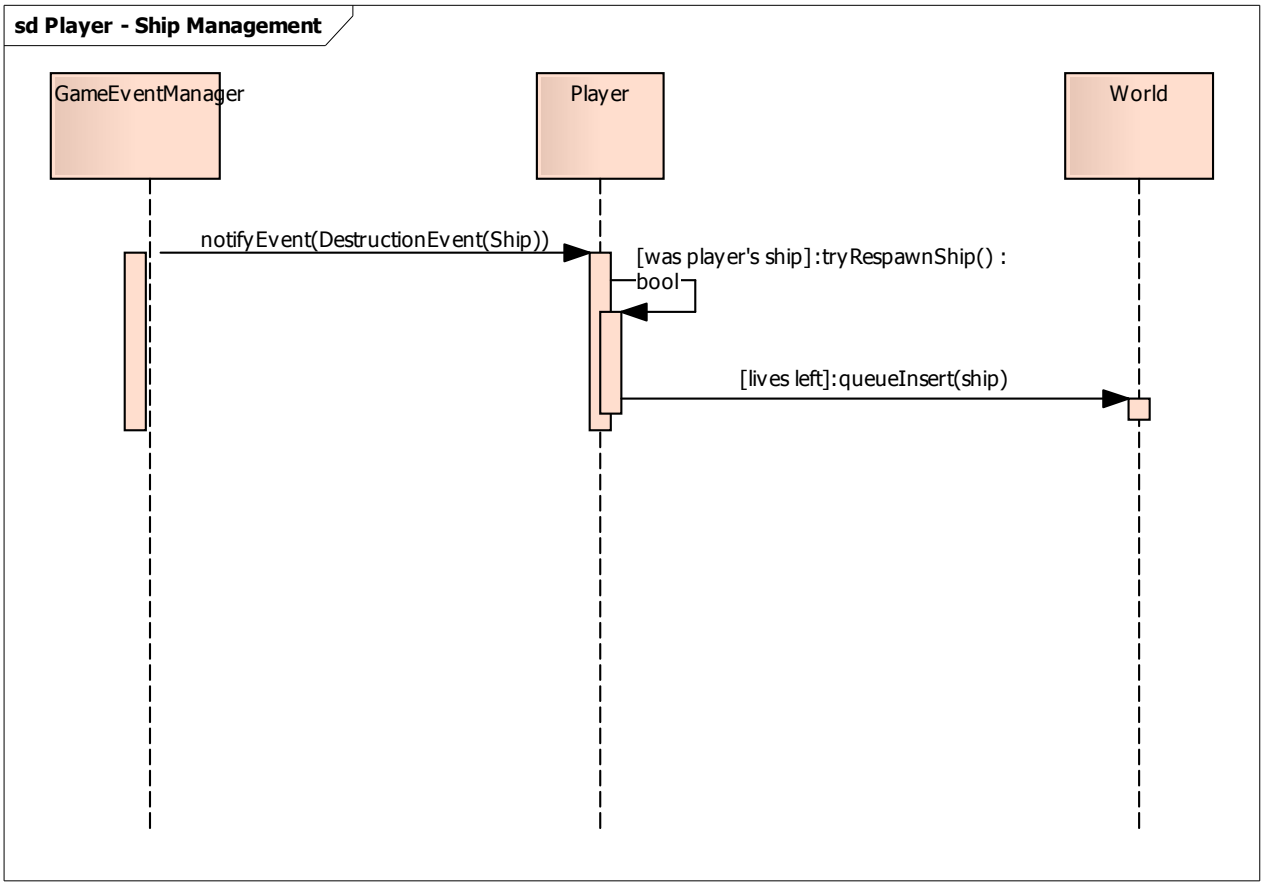
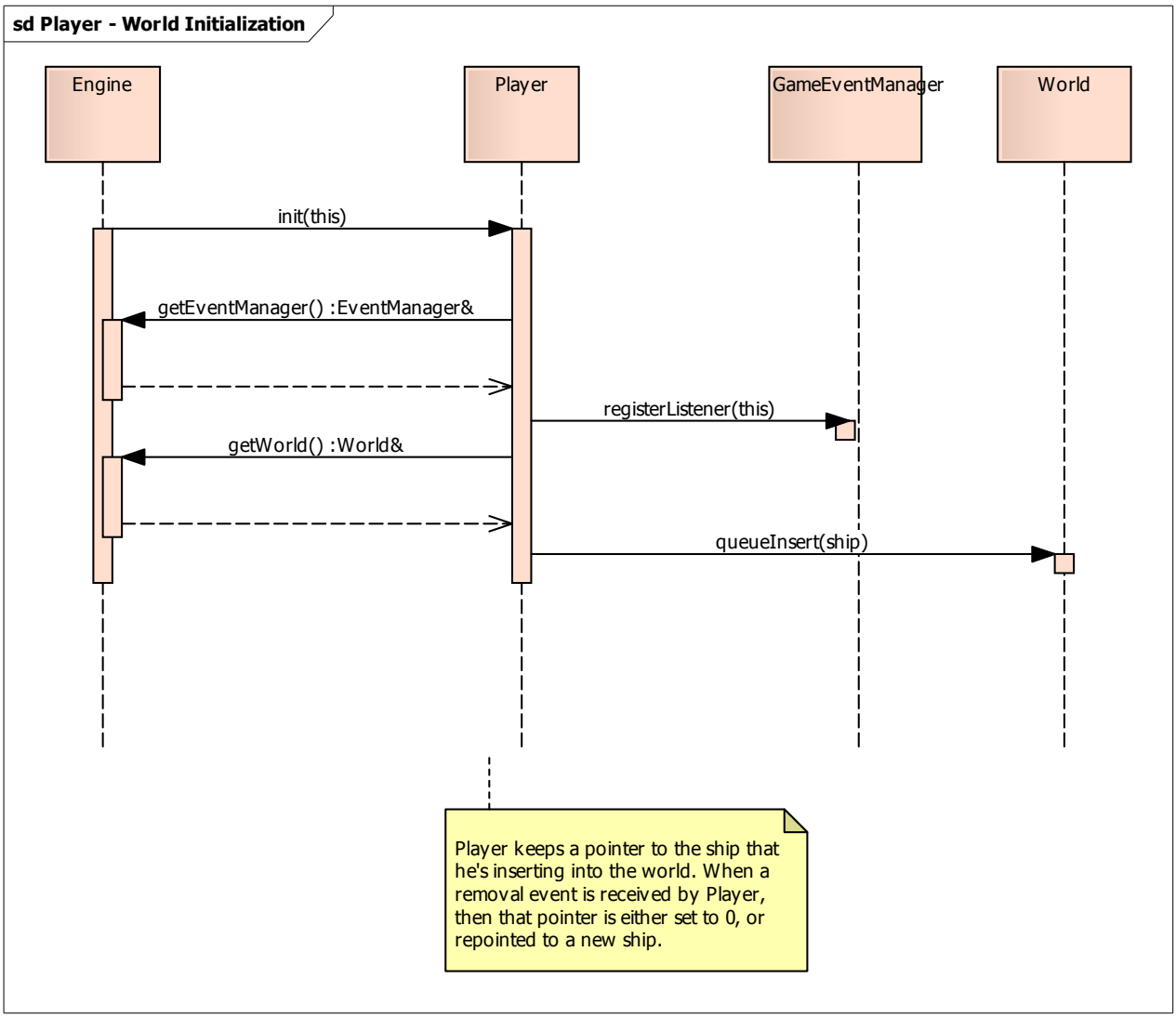


Diagram: Player - World Initialization



5.5 Detailed Design

Gravity

Audio

Responsible for playing sounds for the game. Does so by monitoring the Game module.

Audio::SoundManager

public Class

Implements: GameEventListener, Tickable. : Responsible for monitoring the Game module and playing sounds according to its state.

Audio::SoundManager Methods

Method	Type	Notes
tick ()	«Event» public abstract: *void*	Notifies the object about that the time is being incremented with one time unit. Post-condition: Functional Time-Dependent State Updated - The time-dependent state of the object is updated. Action: Rechecks game module for events/states that require a sound to be played.
notifyEvent (*GameEvent&*)	«Event» public abstract: *void*	param: ev [GameEvent& - in] Checks whether the occured game event should result in a sound being played or not.

init ()	«Manager» public: *void*	Pre-condition: Functional Game singleton initialized
playRepeat (*unsigned short*)	«Manager» private: *void*	param: soundAlias [unsigned short - in] Plays a sound by repeating it.
playOnce (*unsigned short*)	«Manager» private: *void*	param: soundAlias [unsigned short - in] Plays a sound once.

Controller

Responsible for providing the Game module with an easy to use interface for registering different game controls, so that they react on user input.

Controller::GameControlManager

package Class: Responsible for managing a control's activation/deactivation according to changes registered by the input listener(s).

Controller::GameControlManager Attributes

Attribute	Type	Notes
m_controlSwitch	private : *bool*	Tells the status of the most recently registered input for this control, which means that the control may still be activated when this is false. Initial Value: false;

Controller::GameControlManager Methods

Method	Type	Notes
GameControlManager (*GameControl&*)	public:	param: control [GameControl& - in] Constructs a game control manager for a specific game control.
refresh ()	«Manager» public: *void*	Synchronizes the contained game control with the status of the m_switch member. Action: 1. If the control is activated and the input switch is

		false, then the control is deactivated; 2. If the control is deactivated and the input switch is true, then the control is activated; 3. If the control and the input switch have the corresponding status, then nothing happens;
getSwitchPointer ()	«Access» public: *bool**	Returns a pointer to the contained switch. Will be called before making an input listener listen for input, providing it with the switch that it will use to switch according to input status.

Controller::InputListener

package abstract Class: Abstract input listener.

Controller::InputListener Attributes

Attribute	Type	Notes
m_switches	private : *std::map<unsigned short, bool*>*	

Controller::InputListener Methods

Method	Type	Notes
enableListen (*unsigned short, bool**)	«Manager» public: *void*	param: controlAlias [unsigned short - in] One of the CA_* constants, specified by the concrete listener. param: controlSwitch [bool* - in] Pointer to a boolean switch to manipulate according to input detected by the listener.

		Enables listening for input associated with a control, by specifying the alias for that control, and to manipulate the provided boolean pointer depending on whether the input says that the control should be "switched on"=true or "switched off"=false. Pre-condition: Functional Not already listening for the specified control alias Post-condition: Functional The virtual function listen(unsigned short) called
enableListen (*unsigned short*)	«Manager» protected abstract: *void*	param: controlAlias [unsigned short - in] Tells the input listener to listen for input associated with the specified control alias, and to call switchOn or switchOff when the input changes. Pre-condition: Functional Not already listening for the specified control alias Post-condition: Functional Listener listens for input associated with the control alias
switchOn (*unsigned short*)	«Manager» protected: *void*	param: controlAlias [unsigned short - in] Sets the boolean pointer associated with the specified control alias to true. Pre-condition: Functional The specified control alias is being listened for Action: Sets *(m_switches[controlAlias]) to true.
switchOff (*unsigned short*)	«Manager» protected: *void*	param: controlAlias [unsigned short - in]

		Sets the boolean pointer associated with the specified control alias to false. Pre-condition: Functional The specified control alias is being listened for Action: Sets *(m_switches[controlAlias]) to false.

Controller::InputManager

public Class: Responsible for detecting what game controls are requested to be activated or deactivated based on current input, and calling the corresponding function on the contained game control objects.

Controller::InputManager Attributes

Attribute	Type	Notes
LOCAL_PRIMARY_PLAYER	public const static : *unsigned short*	A player type constant used when registering controls for local player 1. Initial Value: 1;
LOCAL_SECONDARY_PLAYER	public const static : *unsigned short*	A player type constant used when registering controls for local player 2. Initial Value: 2;

Controller::InputManager Methods

Method	Type	Notes
registerLocalControl (*NextMenuButtonContr*	«Manager»	param: control [NextMenuButtonControl& - in]

ol&)	public: *void*	<u>Action:</u> Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalControl (*PrevMenuButtonContr ol&*)	«Manager» public: *void*	param: control [PrevMenuButtonControl& - in] <u>Action:</u> Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalControl (*PressMenuButtonContr ol&*)	«Manager» public: *void*	param: control [PressMenuButtonControl& - in] <u>Action:</u> Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalControl (*PauseGameControl&*)	«Manager» public: *void*	param: control [PauseGameControl& - in] <u>Action:</u> Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalControl (*FinishStateControl&*)	«Manager» public: *void*	param: control [FinishStateControl& - in]

		Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalControl (*ConfigKeyboardMapControl&*)	«Manager» public: *void*	param: control [ConfigKeyboardMapControl& - in] Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalPlayerControl (*ShipThrottleControl&, unsigned short*)	«Manager» public: *void*	param: control [ShipThrottleControl& - in] param: playerType [unsigned short - in] Either LOCAL_PRIMARY_PLAYER or LOCAL_SECONDARY_PLAYER. Registers the ship throttle control for a local player, specifying the player type (primary or secondary). Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalPlayerControl (*ShipLeftControl&, unsigned short*)	«Manager» public: *void*	param: control [ShipLeftControl& - in] param: playerType [unsigned short - in] Either LOCAL_PRIMARY_PLAYER or LOCAL_SECONDARY_PLAYER. Registers a ship control for a local player, specifying the player type (primary or secondary).

		Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalPlayerControl (*ShipRightControl&, unsigned short*)	«Manager» public: *void*	param: control [ShipRightControl& - in] param: playerType [unsigned short - in] Either LOCAL_PRIMARY_PLAYER or LOCAL_SECONDARY_PLAYER. Registers a ship control for a local player, specifying the player type (primary or secondary). Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalPlayerControl (*ShipLaserFireControl &, unsigned short*)	«Manager» public: *void*	param: control [ShipLaserFireControl& - in] param: playerType [unsigned short - in] Either LOCAL_PRIMARY_PLAYER or LOCAL_SECONDARY_PLAYER. Registers a ship control for a local player, specifying the player type (primary or secondary). Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
registerLocalPlayerControl (*ShipMissileFireControl*	«Manager» public: *void*	param: control [ShipMissileFireControl& - in] param: playerType [unsigned short - in] Either LOCAL_PRIMARY_PLAYER or

&, unsigned short)		LOCAL_SECONDARY_PLAYER. Registers a ship control for a local player, specifying the player type (primary or secondary). Action: Creates a new instance out of the received game control on the heap, creates a corresponding key listener on the heap, and inserts the resulting pointers in the Control-to-Listener map.
refresh ()	«Manager» public: *void*	Loops through the game control managers vector, refreshing each one of these. Post-condition: Functional Each contained control manager received a refresh() call
reset ()	«Manager» public: *void*	Deactivates all currently active controls. Useful when switching to a child game state and want to avoid having controls left activated when falling back again. Post-condition: Functional All contained active controls deactivated Action: Loops through the game control managers vector, setting their control switch to false and refreshing them.

Controller::KeyboardListener

public Class

Extends: InputListener. : Responsible for monitoring a defined set of keyboard keys and call InputListener::switchOn/switchOff functions when a key's status changes (pressed/unpressed).

Controller::KeyboardListener Attributes

Attribute	Type	Notes
CA_MENU_BTN_NEXT	public const static : *unsigned short*	Control Alias: Navigate to the next button in a menu. Initial Value: 1;
CA_MENU_BTN_PREV	public const static : *unsigned short*	Control Alias: Navigate to the previous button in a menu. Initial Value: 2;
CA_MENU_BTN_PRESS	public const static : *unsigned short*	Control Alias: Press the currently selected button in a menu. Initial Value: 3;
CA_GAMEPLAY_PAUSE	public const static : *unsigned short*	Control Alias: Pause a game play session (state). Initial Value: 4;
CA_GAMEPLAY_P1_SHIP_THROTTLE	public const static : *unsigned short*	Control Alias: Throttle the ship of the first keyboard-controlled player. Initial Value: 5;
CA_GAMEPLAY_P1_SHIP_LEFT	public const static : *unsigned short*	Control Alias: Left-turn the ship of the first keyboard-controlled player. Initial Value: 6;
CA_GAMEPLAY_P1_SHIP_RIGHT	public const static : *unsigned short*	Control Alias: Right-turn the ship of the first keyboard-controlled player. Initial Value: 7;
CA_GAMEPLAY_P1_SHIP_FIRE_LASER	public const static : *unsigned short*	Control Alias: Fire laser guns from the ship of the first keyboard-controlled player. Initial Value: 8;
CA_GAMEPLAY_P1_	public const static :	Control Alias: Fire missiles from the ship of the first keyboard-controlled player.

SHIP_FIRE_MISSILE	*unsigned short*	Initial Value: 9;
CA_GAMEPLAY_P2_SHIP_THROTTLE	public const static : *unsigned short*	Control Alias: Throttle the ship of the second keyboard-controlled player. Initial Value: 10;
CA_GAMEPLAY_P2_SHIP_LEFT	public const static : *unsigned short*	Control Alias: Left-turn the ship of the second keyboard-controlled player. Initial Value: 11;
CA_GAMEPLAY_P2_SHIP_RIGHT	public const static : *unsigned short*	Control Alias: Right-turn the ship of the second keyboard-controlled player. Initial Value: 12;
CA_GAMEPLAY_P2_SHIP_FIRE_LASER	public const static : *unsigned short*	Control Alias: Fire laser guns from the ship of the second keyboard-controlled player. Initial Value: 13;
CA_GAMEPLAY_P2_SHIP_FIRE_MISSILE	public const static : *unsigned short*	Control Alias: Fire missiles from the ship of the second keyboard-controlled player. Initial Value: 14;
CA_FINISH_GAME_STATE	public const static : *unsigned short*	Control Alias: Finish the current game state. Initial Value: 15;
CA_CONFIG_KEYMAP	public const static : *unsigned short*	Control Alias: Configure the mapping between the control and the keyboard. Initial Value: 16;
m_listenedKeysMap	private : *std::map<unsigned short, unsigned short>*	Maps the codes of the listened keyboard keys to control aliases.

Controller::KeyboardListener Methods

Method	Type	Notes
enableListen (*unsigned short*)	«Manager» protected abstract: *void*	param: controlAlias [unsigned short - in] Starts listening for input associated with the specified control alias, resulting in InputListener::switchOn/Off being called when the corresponding key is pressed/unpressed. Pre-condition: Functional Not already listening for the specified control alias Post-condition: Functional Listener listens for input associated with the control alias

Game

Contains the whole game logic. Responsible for handling all different game states, such as menus, game play sessions, etc, as well as transitions between them. Further, the logic for each game state is also found in here, for example, the game world itself, with ships, etc.

Game::Game

public «singleton» Class: Runs the main loop and forwards control to other game states by calling their tick() function on synchronized time intervals. If a game state turns invalid, it will be removed and a fallback to the previous state will be done.

Game::Game Attributes

Attribute	Type	Notes
m_singleton	private static : *Game*	Contains the singleton instance of Game (self). Initial Value: 0;

Game::Game Methods

Method	Type	Notes
instance ()	«Manager» public static: *Game*	Creates a new Game instance if not already created and returns it.
getCurrentState ()	«Access» public: *GameState&*	Returns the current game state.
init ()	«Manager» public: *void*	Runs initialization code and starts the game by starting up the main menu state loop by calling run().

		Pre-condition: Functional Init not called before Post-condition: Functional The game is in the main menu state
run ()	«Manager» public: *void*	Runs the game loop. Action: Runs game loop.
enterState (*GameState&*)	«Manager» public: *void*	param: state [GameState& - in] Enters a game state, pushing the previous one on a stack. Pre-condition: Functional The game state has not been entered already Post-condition: Functional Previous game state pushed on stack Post-condition: Functional The game state was entered - The game state was entered and its enter() function was called, specifying the state that it was entered from.
getStateLevel ()	«Access» public: *unsigned int*	Returns the level of the current state. Example: Main Game State will always be at level 0, states entered from the main state will have level 1, and so on.
registerTickable (*Tickable**)	«Manager» public: *void*	param: tickable [Tickable* - in] Registers a tickable object to tick.
unregisterTickable (*Tickable**)	«Manager» public: *void*	param: tickable [Tickable* - in] Unregisters a tickable object.

Control

Contains controls that represent all actions that a end-user can take during the execution of the game application.

Control::GameControl

public abstract Class {root}: Abstract game control class, whose leaf nodes represent a specific game control, used to manipulate the game state. Examples of possible controls are: "pause game", "steer ship left", etc.

Control::GameControl Attributes

Attribute	Type	Notes
m_isActivated	private : *bool*	Tells whether the control is activated or not.

Control::GameControl Methods

Method	Type	Notes
isActivated ()	«Access» public: *bool*	Tells whether the control is activated or not.
activate ()	«Manager» public abstract: *void*	Activates the control.
deactivate ()	«Manager» public abstract: *void*	Deactivates the control. Pre-condition: Functional Control is activated

Control::MenuControl

public abstract Class

Extends: GameControl. : Controls used in a menu.

Control::MenuControl Methods

Method	Type	Notes
MenuControl (*Menu**)	public:	param: menu [Menu* - in] Constructs a menu control, specifying the menu that it should control.
deactivate ()	«Event» public abstract: *void*	Does nothing, as we, by default, haven't turned anything on with activate() - just performed a one-time action on the menu state's menu. Post-condition: Functional Menu not affected

Control::PrevMenuButtonControl

public Class {leaf}

Extends: MenuControl. : Responsible for setting a menu's button backward iteration on when activated, and off when deactivated.

Control::PrevMenuButtonControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Starts a backward button iteration on the menu. Post-condition: Functional Menu started a backward button iteration

deactivate ()	«Event» public abstract: *void*	Stops a backward button iteration on the menu. Post-condition: Functional Menu stopped backward button iteration

Control::NextMenuButtonControl

public Class {leaf}

Extends: MenuControl. : Responsible for setting a menu's button forward iteration on when activated, and off when deactivated.

Control::NextMenuButtonControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Starts a forward button iteration on the menu. Post-condition: Functional Menu started a forward button iteration
deactivate ()	«Event» public abstract: *void*	Stops a forward button iteration on the menu. Post-condition: Functional Menu stopped forward button iteration

Control::PressMenuButtonControl

public Class {leaf}

Extends: MenuControl. : Responsible for pressing the menu's currently selected button.

Control::PressMenuButtonControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Presses the currently selected button in the current menu state's menu. Pre-condition: Functional Game is in a menu state Post-condition: Functional Current menu state's button pressed

Control::GamePlayControl

public abstract Class

Extends: GameControl. : Abstract class for controls that manipulate the game play state (e.g., ship steering).

Control::PauseGameControl

public Class {leaf}

Extends: GamePlayControl. : Pause a game session.

Control::PauseGameControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Triggers a transition from a game play state to the pause menu state. Pre-condition: Functional Current state is a play state Post-condition: Functional Current state is the pause menu state

deactivate ()	«Event» public abstract: *void*	No function (void).

Control::PlayerControl

public abstract Class

Extends: GamePlayControl. : Abstract class for controls that affect a player.

Control::PlayerControl Methods

Method	Type	Notes
PlayerControl (*Player**)	public:	param: player [Player* - in] The ship to affect. Constructs a ship control for a player.

Control::ShipThrottleControl

public Class {leaf}

Extends: PlayerControl. : Throttle a ship.

Control::ShipThrottleControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Toggles the player's ship throttling on. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship is throttling

deactivate ()	«Event» public abstract: *void*	Toggles the player's ship throttling off. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship is not throttling

Control::ShipLeftControl

public Class {leaf}

Extends: PlayerControl. : Turn a ship left.

Control::ShipLeftControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Turns the player's ship left turning operation on. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship started the left turning operation
deactivate ()	«Event» public abstract: *void*	Turns the player's ship left turning operation off. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship stopped the left turning operation

Control::ShipRightControl

public Class {leaf}

Extends: PlayerControl. : Turn a ship right.

Control::ShipRightControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Turns the player's ship right turning operation on. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship started the right turning operation
deactivate ()	«Event» public abstract: *void*	Turns the player's ship right turning operation off. Pre-condition: Functional Current game state is a play state Post-condition: Functional The player's ship stopped the right turning operation

Control::ShipLaserFireControl

public Class {leaf}

Extends: PlayerControl. : Fire a ship's laser gun.

Control::ShipLaserFireControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Toggles the laser gun on the player's ship on. Pre-condition: Functional Current game state is a play state

		Post-condition: Functional Laser gun on the player's ship toggled on
deactivate ()	«Event» public abstract: *void*	Toggles the laser gun on the player's ship off. Pre-condition: Functional Current game state is a play state Post-condition: Functional Laser gun on the player's ship toggled off

Control::ShipMissileFireControl

public Class {leaf}

Extends: PlayerControl. : Fire a ship's missile launcher.

Control::ShipMissileFireControl Methods

Method	Type	Notes
activate ()	«Event» public abstract: *void*	Toggles the missile launcher on the player's ship on. Pre-condition: Functional Current game state is a play state Post-condition: Functional Missile launcher on the player's ship toggled on
deactivate ()	«Event» public abstract: *void*	Toggles the missile launcher on the player's ship off. Pre-condition: Functional Current game state is a play state Post-condition: Functional Missile launcher on the player's ship toggled off

Control::FinishStateControl

public Class {leaf}

Extends: GameControl. : Responsible for finishing the contained state when activated.

Control::FinishStateControl Methods

Method	Type	Notes
FinishStateControl (*GameState**)	public: *void*	param: state [GameState* - in] Constructs a finish state control for the specified game state.
activate ()	«Event» public abstract: *void*	Finishes the contained state. Post-condition: Functional Contained game state finished
deactivate ()	«Event» public abstract: *void*	Does nothing. Post-condition: Functional Nothing done

Control::ConfigKeyboardMapControl

public Class {leaf}

Extends: GameControl. : Responsible for reconfiguring the mapping between a keyboard key and a game control.

Control::ConfigKeyboardMapControl Attributes

Attribute	Type	Notes

CTRL_P1_SHIP_THROTTLE	public const static : *unsigned int*	Player one ship throttle. Initial Value: 1;
CTRL_P1_SHIP_LEFT	public const static : *unsigned int*	Player one ship left. Initial Value: 2;
CTRL_P1_SHIP_RIGHT	public const static : *unsigned int*	Player one ship right. Initial Value: 3;
CTRL_P1_SHIP_FIRE_LASER	public const static : *unsigned int*	Player one ship laser fire. Initial Value: 4;
CTRL_P1_SHIP_FIRE_MISSILE	public const static : *unsigned int*	Player one ship missile fire. Initial Value: 5;
CTRL_P2_SHIP_THROTTLE	public const static : *unsigned int*	Player two ship throttle. Initial Value: 6;
CTRL_P2_SHIP_LEFT	public const static : *unsigned int*	Player two ship left. Initial Value: 7;
CTRL_P2_SHIP_RIGHT	public const static : *unsigned int*	Player two ship right. Initial Value: 8;
CTRL_P2_SHIP_FIRE_LASER	public const static : *unsigned int*	Player two ship laser fire. Initial Value: 9;
CTRL_P2_SHIP_FIRE_MISSILE	public const static : *unsigned int*	Player two ship missile fire. Initial Value: 10;
m_controlAlias	private : *unsigned int*	The alias of the control to reconfigure.

m_newKey	private : *unsigned int*	The new key to set for the control (will be set by keyboard listener).
m_done	private : *bool*	Tells whether the reconfiguration has been done or not. Initial Value: false;

Control::ConfigKeyboardMapControl Methods

Method	Type	Notes
ConfigKeyboardMapControl (*unsigned int*)	public:	param: controlAlias [unsigned int - in] Must be one of the CTRL_* constants in this class. Constructs a keyboard map reconfig control, specifying the alias for the control that is to be reconfigured.
isDone ()	«Access» public: *bool*	Tells whether the reconfiguration has been done or not.
activate ()	«Event» public abstract: *void*	Saves the new key by overwriting the old setting. Pre-condition: Functional New key has been set - In order to activate this control and save a new key configuration, we have to know the key, i.e., setNewKey has been called.
deactivate ()	«Event» public abstract: *void*	Updates the state of this control to "done". Post-condition: Functional Control status set to done
setNewKey (*unsigned int*)	«Manager» public: *void*	param: key [unsigned int - in] Sets the new keyboard key for the control. Will be called by keyboard when a key press is detected.

		Note, however, that this function will not actually save the new key, it will just update the class' state. To save the key, activate() must be called.

Engine

Contains handlers (strategies) for the world, as well as the engine that is responsible for maintaining the world. Examples of strategies are: Collision Strategy and Gravity Strategy.

Engine::Engine

public Class

Implements: Tickable. : Responsible for maintaining the players and the game world they play within.

Engine::Engine Methods

Method	Type	Notes
Engine ()	public:	Constructs a game engine.
getWorld ()	«Access» public: *World&*	Returns the game world instance.
getPlayer (*unsigned int*)	«Access» public: *Player&*	param: num [unsigned int - in] Returns the player that is associated with the specified number (first=0, second=1, ...).
getPlayers ()	«Access» public: *std::vector<Player>::iterator*	Returns an iterator of all contained players. May be called by the game state to check if a player has lost or not, for example.
tick ()	«Event» public abstract: *void*	Ticks the world instance. Post-condition: Functional World has received a tick call
init (*World&*)	«Manager»	param: world [World& - in]

	public: *void*	Initiates the engine by specifying the game world that it should use. The world will be validated before put into use, resulting in a thrown exception if the world is invalid. Pre-condition: Functional All players are added Pre-condition: Functional World contains no movable objects - No movable objects are allowed to be present in the world initially (as this complicates its validation). Post-condition: Functional All players initialized - All players are initialized, being provided the game world. Post-condition: Functional Event manager created Post-condition: Functional World is registered as a listener in the event manager Post-condition: Functional World is validated
addPlayer (*Player&*)	«Manager» public: *void*	param: player [Player& - in] Adds a player to the engine. For each added player the next player's number increases with 1, starting at 0. Pre-condition: Functional init() not ran Pre-condition: Functional Player not present in engine Post-condition: Functional Player present in engine
validateWorld (*World*)	«Helper» public: *void*	param: world [World - in] Makes sure that no static objects are intersecting and that no movable objects are present.

WorldEvent

Contains event representation classes as well as a world event manager and a related interface for world event listeners.

WorldEvent::CollisionEvent

public «Message» Message

Extends: WorldEvent. : Represents an event of a collision between two objects in the game world and provides these objects. This event should be cascaded ONCE for each collision (i.e., we do not differentiate the objects participating in the collision here). Further, a collision event is expected to be cascaded regardless of whether one or both of the colliders were destroyed.

WorldEvent::CollisionEvent Methods

Method	Type	Notes
CollisionEvent (*WorldObject*, WorldObject**)	public:	param: wo1 [WorldObject* - in] param: wo2 [WorldObject* - in] Constructs a collision event with the specified participating world objects.
getFirst ()	«Access» public const: *WorldObject&*	Returns the first of the two world objects that participated in the collision.
getSecond ()	«Access» public const: *WorldObject&*	Returns the second of the two world objects that participated in the collision.

WorldEvent::DamageEvent

public «Message» Message

Extends: WorldEvent. : Represents a damage event and provides the destroyable object that was damaged (which isn't the same as being destroyed).

WorldEvent::DamageEvent Methods

Method	Type	Notes
DamageEvent (*DestroyableObject&*)	public:	param: wo [DestroyableObject& - in] Constructs a damage event, specifying the damaged object.
getDamaged ()	«Access» public const: *DestroyableObject&*	Returns the object that was damaged.

WorldEvent::DestructionEvent

public «Message» Message

Extends: WorldEvent. : Represents a destruction event and provides the destroyable object that was destroyed as well as the world object that caused its destruction (by, for example, colliding with it). A destruction event must not be treated as a RemovalOrderEvent, and vice versa, as a destruction doesn't necessarily mean that the object will be removed before next tick.

WorldEvent::DestructionEvent Methods

Method	Type	Notes
DestructionEvent (*DestroyableObject&, WorldObject&*)	public:	param: wo [DestroyableObject& - in] param: cause [WorldObject& - in] The world object causing the destruction.

		Constructs a destruction event, specifying the destroyed object as well as the world object that caused its destruction.
getDestroyed ()	«Access» public const: *DestroyableObject&*	Returns the object that was destroyed.

WorldEvent::InsertionEvent

public «Message» Message

Extends: WorldEvent. : Represents the event of a world object being inserted into the game world (i.e., not queued for insertion!).

WorldEvent::InsertionEvent Methods

Method	Type	Notes
InsertionEvent (*WorldObject**)	public:	param: wo [WorldObject* - in] Constructs a world insertion event.
getInserted ()	«Access» public const: *WorldObject&*	Returns the world object that was inserted into the world.

WorldEvent::ItemPickupEvent

public «Message» Message

Extends: WorldEvent. : Represents the event of a ship picking up an item, and provides both the ship and the picked up item.

WorldEvent::ItemPickupEvent Methods

Method	Type	Notes
ItemPickupEvent (*Item*, Ship**)	public:	param: item [Item* - in] param: ship [Ship* - in]
getItem ()	«Access» public const: *Item&*	Returns the item that was picked up by the ship.
getShip ()	«Access» public const: *Ship&*	Returns the ship that picked up the item.

WorldEvent::ProjectileFireEvent

public «Message» Message

Extends: WorldEvent. : Represents the event of firing a projectile and provides the projectile that was fired.

WorldEvent::ProjectileFireEvent Methods

Method	Type	Notes
ProjectileFireEvent (*Projectile**)	public:	param: projectile [Projectile* - in]
getProjectile ()	«Access» public const: *Projectile&*	Returns the projectile that was fired.

WorldEvent::RemovalOrderEvent

public «Message» Message

Extends: WorldEvent. : Represents the event of the world being ordered to remove a world object and provides that object. The removal order will be realized at the end of the world's current tick call. This event should be listened for by all classes that are keeping pointers to world objects, so that they know when to get rid of them.

WorldEvent::RemovalOrderEvent Methods

Method	Type	Notes
RemovalOrderEvent (*WorldObject**)	public:	param: wo [WorldObject* - in] Constructs a removal event of the specified world object. Pre-condition: Functional World object not removed - The world object must not be removed until this event has finished cascading.
getWorldObject ()	«Access» public const: *WorldObject&*	Returns the world object that is to be removed from the world (i.e., is not removed *yet*).

WorldEvent::WorldEvent

public «Message» Message

Extends: GameEvent. :

WorldLife

Contains world strategies for management of the world's life, i.e., inserting and removing world objects according to the rules defined by the strategies.

WorldLife::AsteroidStrategy

public Class

Implements: WorldStrategy. : Responsible for providing the world with asteroids, based on time-based constraints.

WorldLife::AsteroidStrategy Methods

Method	Type	Notes
AsteroidStrategy (*unsigned int*)	public:	param: timeDelta [unsigned int - in] The world time delta between asteroid insertions. Constructs an asteroid insertion world strategy, specifying the world time delta to wait between asteroid insertions.
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Throws in a pre-defined amount (see constructor) of asteroids into the world on each call. If the defined amount is less than 1, it will mean that an asteroid will not be thrown in on each call, but more rarely.

WorldLife::ExpirationStrategy

public Class

Implements: GameEventListener, WorldStrategy. : Responsible for deciding

what world objects should expire and when they should do so, resulting in being removed from the world. Examples are: projectiles, which shouldn't be in the world too long.

WorldLife::ExpirationStrategy Attributes

Attribute	Type	Notes
m_laserExpire	private : *unsigned int*	The number of ticks from a laser's detection to its expiration. Initial Value: 0;
m_missileExpire	private : *unsigned int*	The number of ticks from a missile's detection to its expiration. Initial Value: 0;
m_missileItemExpire	private : *unsigned int*	The number of ticks from a missile item's detection to its expiration. Initial Value: 0;
m_fuelItemExpire	private : *unsigned int*	The number of ticks from a fuel item's detection to its expiration. Initial Value: 0;

WorldLife::ExpirationStrategy Methods

Method	Type	Notes
ExpirationStrategy (*EventManager**)	public: *void*	param: eventManager [EventManager* - in] Constructs an expiration strategy with the specified event manager to register itself in. Post-condition: Functional Expiration strategy registered as a listener in event manager

notifyEvent (*GameEvent&*)	«Manager» public abstract: *void*	param: ev [GameEvent& - in] Notifies the expiration strategy about an occuring event. By default, no action taken (see function overload(s)).
notifyEvent (*RemovalEvent&*)	«Manager» public abstract: *void*	param: ev [RemovalEvent& - in] Notifies the expiration strategy about an occuring world removal event. Post-condition: Functional The removed world object not found in internal pointers - The queues of world object pointers that we store internally must not contain pointers to objects that are being removed from the world. Action: Removes possible internal pointers to the objects that are being removed from the world.
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Applies a world strategy on a world instance. Post-condition: Functional World state changed according to strategy Action: Calls handleExpiration() for each object in the world.
setLaserExpiration (*unsigned int*)	«Manager» public: *void*	param: ticks [unsigned int - in] Sets the number of ticks to receive before a laser is removed from the world. Setting ticks to 0 means that it will last until removed in a natural way.

setMissileExpiration (*unsigned int*)	«Manager» public: *void*	param: ticks [unsigned int - in] Sets the number of ticks to receive before a missile is removed from the world. Setting ticks to 0 means that it will last until removed in a natural way.
setMissileItemExpiration (*unsigned int*)	«Manager» public: *void*	param: ticks [unsigned int - in] Sets the number of ticks to receive before a missile item is removed from the world. Setting ticks to 0 means that it will last until removed in a natural way.
setFuelItemExpiration (*unsigned int*)	«Manager» public: *void*	param: ticks [unsigned int - in] Sets the number of ticks to receive before a fuel item is removed from the world. Setting ticks to 0 means that it will last until removed in a natural way.
handleExpiration (*WorldObject&*)	«Manager» private: *void*	param: wo [WorldObject& - in] Does no handling for a generic world object.
handleExpiration (*LaserProjectile&*)	«Manager» private: *void*	param: laser [LaserProjectile& - in] Handle's a laser's expiration. Post-condition: Functional If no expiration, then not inserted into monitor queue Action: Checks whether the laser already is monitored or not, and if not, then it will be put into the monitoring queue. If, on the other hand, the laser already is monitored, then an expiration check will be done, and, if expired, the laser will be put into the world's remove queue.

handleExpiration (*MissileProjectile&*)	«Manager» private: *void*	param: missile [MissileProjectile& - in] Handle's a missile's expiration. Post-condition: Functional If no expiration, then not inserted into monitor queue Action: Checks whether the missile already is monitored or not, and if not, then it will be put into the monitoring queue. If, on the other hand, the missile already is monitored, then an expiration check will be done, and, if expired, the missile will be put into the world's remove queue.
handleExpiration (*MissileItem&*)	«Manager» private: *void*	param: missileItem [MissileItem& - in] Handle's a missile item's expiration. Post-condition: Functional If no expiration, then not inserted into monitor queue Action: Checks whether the missile item already is monitored or not, and if not, then it will be put into the monitoring queue. If, on the other hand, the item already is monitored, then an expiration check will be done, and, if expired, the item will be put into the world's remove queue.
handleExpiration (*FuelItem&*)	«Manager» private: *void*	param: fuelItem [FuelItem& - in] Handle's a fuel item's expiration. Post-condition: Functional If no expiration, then not inserted into monitor queue

		Action: Checks whether the fuel item already is monitored or not, and if not, then it will be put into the monitoring queue. If, on the other hand, the item already is monitored, then an expiration check will be done, and, if expired, the item will be put into the world's remove queue.

WorldLife::ItemStrategy

public Class

Implements: WorldStrategy. : Responsible for providing the world with items, based on time-based constraints.

WorldLife::ItemStrategy Methods

Method	Type	Notes
registerItem (*Item, unsigned int*)	«Manager» public: *void*	param: item [Item - in] param: interval [unsigned int - in] World time delta between insertions. Registers an item, specifying the time interval between its insertions into the world. Pre-condition: Functional Item not already added
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Checks if there are items that should be inserted into the world, and does insert them if found.

WorldPhysics

Contains managers of the physics in the world.

WorldPhysics::BoundaryStrategy

public abstract Class

Implements: WorldStrategy. : Responsible for making sure that each and every world object is within the world boundaries defined in this object, and if it's not, then it is repositioned according to the reposition() implementation in the concrete class.

WorldPhysics::BoundaryStrategy Methods

Method	Type	Notes
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Makes sure that all world objects are within the defined boundaries. If their not, then they will be repositioned to match that constraint. Post-condition: Functional All world objects are found within defined boundaries - All objects in the world are found within the defined world boundaries. Action: Loops through all world objects, applying reposition() if isBeyond().
reposition (*WorldObject*)	«Manager» protected abstract: *void*	param: wo [WorldObject - in] Repositions a world object without affecting its movement direction or magnitude.

		Pre-condition: Functional World object goes beyond the boundaries - A world object has gone beyond the world boundaries at some point. Post-condition: Functional World object is found within the boundaries - The world object is moved back into the area defined by the world boundary instance. Post-condition: Functional World object's movement direction not changed - The movement direction of the world object must be the same as it was before the repositioning. Post-condition: Functional World object's movement unchanged - The movement direction and magnitude of the world object must be the same as it was before the repositioning.
isBeyond (*WorldObject*)	«Access» protected abstract: *bool*	param: wo [WorldObject - in] Tells whether an object in the world is beyond the world boundaries or not.

WorldPhysics::CollisionStrategy

public Class

Implements: WorldStrategy. : Responsible for detecting and handling collisions between objects in the game world, provided a game world instance.

WorldPhysics::CollisionStrategy Methods

Method	Type	Notes
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Searches for and handles collisions in the world.

		Post-condition: Functional All collisions in the world are handled Post-condition: Functional All non-handled spawn points available - All spawn points that weren't intersected by anything have their availability set to true.
handleCollision (*WorldObject&, WorldObject&*)	«Manager» private abstract: *void*	param: active [WorldObject& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [WorldObject& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Leaves active object unaffected. This function catches all cases where no collision handling is required. Post-condition: Functional Active world object unaffected
handleCollision (*Item&, Ship&*)	«Manager» private abstract: *void*	param: active [Item& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Ship& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Removes the item as a result of it being picked up by the ship. Post-condition: Functional Item is put into the world's remove queue
handleCollision	«Manager»	param: active [Ship& - in]

(*Ship&, FuelItem&*)	private abstract: *void*	The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [FuelItem& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Increases the ship's fuel level. Post-condition: Functional Ship's fuel level increased
handleCollision (*Ship&, MissileItem&*)	«Manager» private abstract: *void*	param: active [Ship& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [MissileItem& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Increases the ship's missile count. Post-condition: Functional Ship's missile count increased
handleCollision (*SpawnPoint&, MovableObject&*)	«Manager» private abstract: *void*	param: active [SpawnPoint& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [MovableObject& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Sets the spawn point to not being free (see SpawnPoint::toggleFree()). Note that: if an item appears on a spawn point, the spawn point shall still

		be considered free. Post-condition: Functional Spawn point not free
handleCollision (*Projectile&, DestroyableObject&*)	«Manager» private abstract: *void*	param: active [Projectile& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [DestroyableObject& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Puts the projectile in the world's remove queue, as it is considered consumed. Post-condition: Functional Projectile put into the world's remove queue
handleCollision (*DestroyableObject&, Projectile&*)	«Manager» private abstract: *void*	param: active [DestroyableObject& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Projectile& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Causes damage on the destroyable object, according to the projectile's "max damage strength". Post-condition: Functional Damage caused to the destroyable object
handleCollision (*DestroyableObject&, Planet&*)	«Manager» private abstract: *void*	param: active [DestroyableObject& - in] The world object that actively participates in the intersection, i.e., may be affected by it.

		param: passive [Planet& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Destroys the destroyable object. Post-condition: Functional The destroyable is destroyed
handleCollision (*Asteroid&, Asteroid&*)	«Manager» private abstract: *void*	param: active [Asteroid& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Asteroid& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Causes damage on the active asteroid according to the collision power (i.e., the movement towards the other asteroid). Post-condition: Functional Active asteroid damaged
handleCollision (*Ship&, Ship&*)	«Manager» private abstract: *void*	param: active [Ship& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Ship& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Causes damage on the active ship according to the collision power (i.e., the movement towards the other ship).

		Post-condition: Functional Active ship damaged
handleCollision (*Ship&, Asteroid&*)	«Manager» private abstract: *void*	param: active [Ship& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Asteroid& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Causes damage to the ship according to the collision power (i.e., the movement towards the asteroid). Post-condition: Functional Ship damaged
handleCollision (*Asteroid&, Ship&*)	«Manager» private abstract: *void*	param: active [Asteroid& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Ship& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function). Causes damage to the asteroid according to the collision power (i.e., the movement towards the ship). Post-condition: Functional Asteroid damaged
handleCollision (*LaserProjectile&, Planet&*)	«Manager» private abstract: *void*	param: active [LaserProjectile& - in] The world object that actively participates in the intersection, i.e., may be affected by it. param: passive [Planet& - in] The world object that passively participates in the intersection, i.e., will not be affected by it (in this function).

		Bounces the laser projectile on the planet. Post-condition: Functional Laser projectile set off in a direction from the planet
collidesWith (*WorldObject&, WorldObject&*)	«Helper» private: *bool*	param: wo1 [WorldObject& - in] param: wo2 [WorldObject& - in] Tells whether two objects collide or not.

WorldPhysics::GravityStrategy

public Class

Implements: WorldStrategy. : Responsible for calculating and applying gravity affections for each gravity-affectable game world object.

WorldPhysics::GravityStrategy Methods

Method	Type	Notes
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Affects all movable world objects found in the world according to their gravity fields. Post-condition: Functional Movable objects' movement affected - The world objects found in the range of one or more gravity fields are affected by the corresponding gravity vectors.
affect (*WorldObject&, WorldObject&*)	«Manager» private: *void*	param: wo1 [WorldObject& - in] param: wo2 [WorldObject& - in]

		Does no affection. This function only exists to catch all cases which the other affect function(s) can't handle.
affect (*MovableObject&, Planet&*)	«Manager» private: *void*	param: mwo [MovableObject& - in] param: swo [Planet& - in] Affects a movable object with a planet's gravity, in proportion to the object's mass.

WorldPhysics::RectangularBoundaryStrategy

public Class

Extends: BoundaryStrategy. : Represents the world boundaries, i.e., the area that world objects are allowed to appear on. Here, the world boundaries have a rectangular shape.

WorldPhysics::RectangularBoundaryStrategy Methods

Method	Type	Notes
RectangularBoundaryStrategy (*Coord2d, Coord2d*)	public:	param: upperLeft [Coord2d - in] param: lowerRight [Coord2d - in] Constructs a rectangular boundary with specified upper left and lower right coordinates, this way defining the boundaries.
reposition (*WorldObject*)	«Manager» protected abstract: *void*	param: wo [WorldObject - in] Repositions a world object without affecting its movement direction or magnitude. See pre and post conditions for algorithm details. Pre-condition: Functional World object goes beyond

		the boundaries - A world object has gone beyond the world boundaries at some point. Post-condition: Functional World object is found within the boundaries - The world object is moved back into the area defined by the world boundary instance. Post-condition: Functional World object's movement direction not changed - The movement direction of the world object must be the same as it was before the repositioning. Post-condition: Functional World object's movement unchanged - The movement direction and magnitude of the world object must be the same as it was before the repositioning. Post-condition: Functional World object's position mirrored twice - The world object's new position is a result of mirroring its previous position on both the vertical and the horizontal axis.
isBeyond (*WorldObject*)	«Access» protected abstract: *bool*	param: wo [WorldObject - in] Tells whether an object in the world is beyond the rectangular world boundaries or not.

Event

Contains game event manager, game event listener interface and game event interface, this way providing a way to detect any occuring game event from anywhere.

Event::GameEvent

public «Message» Message: Abstract world event class, bringing all events together under a common type.

Event::GameEventManager

public Class: Responsible for cascading every event that is received to all registered observers (for example, the view and audio module).

Event::GameEventManager Methods

Method	Type	Notes
cascadeEvent (*GameEvent&*)	«Manager» public: *void*	param: ev [GameEvent& - in] Cascades an event to all registered listeners. Pre-condition: Functional At least one event listener present Post-condition: Functional All registered listeners received the event Post-condition: Functional No listener has stored the event - It must not happen that a listener stores the events he receives, as the contained pointers may become invalid before next tick.
registerListener (*GameEventListener**)	«Manager» public: *void*	param: listener [GameEventListener* - in]

		Registers an event listener. Post-condition: Functional Listener inserted into m_listeners
unregisterListener (*GameEventListener**)	«Manager» public: *void*	param: listener [GameEventListener* - in] Unregisters an event listener. Pre-condition: Functional The listener is present in event manager Post-condition: Functional The listener is removed from event manager

Event::GameEventListener

public abstract «interface» Interface: Defines the interface of a class that needs to listen for game events.

Event::GameEventListener Interfaces

Method	Type	Notes
notifyEvent (*GameEvent&*)	«Event» public abstract: *void*	param: ev [GameEvent& - in] Notifies the listener about an occuring game event.

Menu

Contains a representation of a menu and its items.

Menu::EnterStateAction

public Class

Implements: MenuAction. : Responsible for entering a new state from the current one.

Menu::EnterStateAction Methods

Method	Type	Notes
EnterStateAction (*GameState*)	public:	param: state [GameState - in] Constructs an enter game state action, specifying the game state instance to enter.
trigger ()	«Event» public abstract: *void*	Triggers a transition from current game state to a new game state. Post-condition: Functional The contained game state is entered Action: Uses the Game singleton's enterState function.

Menu::LeaveStateAction

public Class

Implements: MenuAction. : Responsible for leaving the current state to the previous one, or if none is available, to quit the game.

Menu::LeaveStateAction Methods

Method	Type	Notes
LeaveStateAction ()	public: *void*	Constructs a leave game state action.
trigger ()	«Event» public abstract: *void*	Finishes the current game state. Post-condition: Functional Current game state is set to finished Action: Accesses the current state through the Game singleton and calls its finish() function.

Menu::Menu

public Class

Implements: Tickable. : Responsible for holding a set of buttons and providing functionality to "press" them as well as navigate in the button list.

Menu::Menu Attributes

Attribute	Type	Notes
ITERATE_NONE	private const static : *unsigned short*	Initial Value: 1;
ITERATE_BACKWARD	private const static : *unsigned short*	Initial Value: 2;
ITERATE_FORWARD	private const static : *unsigned short*	Initial Value: 3;

ITERATION_INIT_DELAY	private const static : *unsigned short*	The number of ticks to wait before continuing the iteration after the first selection. Example: If we're on button "FOO" and then start an iteration, where the next button is "BAR", then this delay defines the number of ticks that we should stay on button "BAR" before entering a faster iteration over the rest of the buttons. Initial Value: 50;
ITERATION_DELAY	private const static : *unsigned short*	The number of ticks to wait on each button during an ongoing iteration. Initial Value: 25;
m_buttonIteration	private : *unsigned short*	The button iteration mode. Initial Value: ITERATE_NONE;
m_iterationCountdown	private : *unsigned int*	Number of ticks left before continuing an ongoing iteration.

Menu::Menu Methods

Method	**Type**	**Notes**
Menu (*std::string*)	public:	param: title [std::string - in] Constructs a menu, specifying its title.
tick ()	«Event» public abstract: *void*	Does button iteration when on. Post-condition: Functional Time-Dependent State Updated - The time-dependent state of the object is updated.
addButton (*MenuButton*)	«Manager» public: *void*	param: item [MenuButton - in] Adds a button to the menu. Insert order defines

		appearance order.
toggleBackwardButtonIteration (*bool*)	«Manager» public: *void*	param: switchTo [bool - in] Toggles the backward iteration over the buttons in the menu. Pre-condition: Functional Button list is not empty - Should be catched and handled internally, i.e., ignore navigation.
toggleForwardButtonIteration (*bool*)	«Manager» public: *void*	param: switchTo [bool - in] Toggles the forward iteration over the buttons in the menu. Pre-condition: Functional Button list is not empty - Should be catched and handled internally, i.e., ignore navigation.
pressSelectedButton ()	«Manager» public: *void*	Presses the currently selected button. Post-condition: Functional Currently selected button is pressed - The button's press() function is called.
selectPrevButton ()	«Manager» private: *void*	Goes one step backward in the button list, selecting the previous button. Pre-condition: Functional Button is not the first one in the list - Should be catched and handled internally, i.e., ignore navigation. Pre-condition: Functional Button list is not empty - Should be catched and handled internally, i.e., ignore navigation. Post-condition: Functional Previous button selected

selectNextButton ()	«Manager» private: *void*	Goes one step forward in the button list, selecting the next button. Pre-condition: Functional Button is not the last one in the list - Should be catched and handled internally, i.e., ignore navigation. Pre-condition: Functional The button list is not empty - Should be catched and handled internally, i.e., ignore navigation. Post-condition: Functional Next button selected

Menu::MenuButton

public Class: Responsible for holding one or more actions to take when pressed.

Menu::MenuButton Attributes

Attribute	Type	Notes
m_name	private : *std::string*	The button's name.

Menu::MenuButton Methods

Method	Type	Notes
MenuButton (*std::string&*)	public:	param: name [std::string& - in] Constructs a button, specifying its name.
getName ()	«Access» public const: *std::string&*	Returns the button's name.

press ()	«Event» public: *void*	Triggers the contained action(s). Action: Triggers contained action(s).
addAction (*MenuAction&*)	«Manager» public: *void*	param: action [MenuAction& - in] Adds an action to trigger when the menu button is pressed. First in first out. Post-condition: Functional The added action is pushed on the actions stack

Menu::MenuAction

public abstract «interface» Interface: Defines the interface of an action that can be executed by a menu button.

Menu::MenuAction Interfaces

Method	Type	Notes
trigger ()	«Event» public abstract: *void*	Triggers the action.

Player

Contains player-related classes, such as the player itself.

Player::Player

public Class

Implements: GameEventListener. : Responsible for maintaining generic statistics about a player's activity in the game world, as well as for managing the player's ships.

Player::Player Attributes

Attribute	Type	Notes
m_name	private : *std::string*	The name of the player.
m_shipsLeft	private : *unsigned short*	Number of ships left to use. Initial Value: 3;
m_killedShips	private : *unsigned int*	Number of killed ships. Initial Value: 0;
m_destroyedAsteroids	private : *unsigned int*	Number of destroyed asteroids. Initial Value: 0;

Player::Player Methods

Method	Type	Notes
Player (*std::string&, Ship&*)	public:	param: name [std::string& - in] param: shipTemplate [Ship& - in]

		Constructs a player, specifying his name and the ship template to use when inserting new ships into the world.
getName ()	«Access» public const: *std::string&*	Returns the name of the player.
getLives ()	«Access» public: *unsigned int*	Gets the number of lives/ships left.
getCurrentShip ()	«Access» public const: *Ship&*	Returns the player's current ship.
getDestroyedAsteroidsCount ()	«Access» public: *unsigned int*	Returns the total number of asteroids destroyed by this player.
getDestroyedShipsCount ()	«Access» public: *unsigned int*	Returns the total number of ships destroyed by this player.
notifyEvent (*GameEvent&*)	«Event» public abstract: *void*	param: ev [GameEvent& - in] Catches generic events - does nothing with them.
notifyEvent (*WorldDestructionEvent&*)	«Event» public abstract: *void*	param: ev [WorldDestructionEvent& - in] Checks if the destroyed object was a ship and if it was this player's ship, if yes, then, if there are ships left, a new ship will be inserted into the world.
init (*Engine&*)	«Manager» public: *void*	param: engine [Engine& - in] Initializes the player in the current environment. Post-condition: Functional Registered in the engine's event manager - The player is registered in the

		provided engine's world event manager. Post-condition: Functional The player is in control of a ship in the world
tryRespawnShip ()	«Manager» private: *bool*	Tries to respawn a new ship for the player. Returns true if respawn succeeds (i.e., the player have lives left) or false otherwise. Post-condition: Functional Current ship pointer updated - Either set to a new ship or to 0.
handleDetectedDestruction (*WorldObject&, WorldObject&*)	«Manager» private: *void*	param: wo1 [WorldObject& - in] param: wo2 [WorldObject& - in] Catches all irrelevant destruction events.
handleDetectedDestruction (*Ship&, Projectile&*)	«Manager» private: *void*	param: ship [Ship& - in] param: projectile [Projectile& - in] Handles the "ship destroyed by projectile" case. If the ship is this player's ship, then we try to respawn a new ship in the world. If the projectile is originating from this ship and the ship isn't this player's, then we increment the "killed ships" counter.

State

Contains all the game states that the game may found itself in. For example: Main Menu or Single Player Game Session.

State::GameState

public abstract Class {root}

Implements: Tickable. : Represents an abstract state that the game may find itself in.

State::GameState Attributes

Attribute	Type	Notes
m_valid	private : *bool*	Tells whether the game state is supposed to be run (true) or to be finished (false). Initial Value: true;

State::GameState Methods

Method	Type	Notes
isValid ()	«Access» public: *bool*	Tells whether the game state is supposed to be run (true) or to be finished (false).
getEventManager ()	«Access» public const: *GameEventManager&*	Provides the event manager for the game state. Action: Creates a new GameEventManager if not already set, and returns it.
enterFrom	«Manager» public abstract:	param: fromState [GameState* - in] Takes a pointer because we need to be able to

(*GameState**)	*void*	accept 0 as value. Enters the game state, and, when possible, lets it know what state it was entered from (i.e., its parent state). Takes a pointer because we need to be able to accept 0 as value (when entering main menu, for example). Pre-condition: Functional State not entered before Post-condition: Functional Game state is properly initialized Post-condition: Functional Parent state NOT stored
handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Notifies this state about a fallback to it from a child state, providing the child state instance. The child game state will be checked, and, when required, proper action will be taken (for example, if the child game state was a pause menu, it may have ordered a "quit", and we should realize it). Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Orders from child state are realized - Any orders from the child state shall be realized. For example, if the child state was a pause menu, and it concluded that the game session should quit, then we must realize that order. Action: does nothing by default
finish ()	«Manager» public: *void*	Sets m_valid to false, causing the game state to finish when isValid() is checked by Game.

State::MenuState

public abstract Class

Extends: GameState. : The state to be in when navigating through menus.

State::MenuState Methods

Method	Type	Notes
getMenu ()	«Access» public: *Menu&*	Returns the menu that is used in the menu state.
tick ()	«Event» public abstract: *void*	Makes the menu state to self-update according to the status of the controller module. Action: Polls controller and updates contained menu if necessary.
initMenu (*Menu*)	«Manager» protected: *void*	param: menu [Menu - in] Sets the menu and registers menu controls for it in input manager. This function must not be called more than once. Pre-condition: Functional Menu not already set Post-condition: Functional Menu controls registered in input manager Post-condition: Functional Menu set

State::MainMenuState

public Class {leaf}

Extends: MenuState. :

State::MainMenuState Methods

Method	Type	Notes
enterFrom (*GameState**)	«Manager» public abstract: *void*	param: fromState [GameState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the main menu state, only accepting fromState == 0. If fromState isn't zero, then an exception will be thrown. Pre-condition: Functional Argument fromState == 0 Pre-condition: Functional State not entered before Post-condition: Functional Main menu state properly initialized

State::MapChoiceMenuState

public Class {leaf}

Extends: MenuState. : The map choice menu, containing one button for each available map.

State::MapChoiceMenuState Methods

Method	Type	Notes
getWorld ()	«Access» public const: *World&*	Returns the world instance that has been setup according to the currently selected map. Pre-condition: Functional A map has been selected Post-condition: Functional A world is generated from the map

enterFrom (*MainMenuState**)	«Manager» public abstract: *void*	param: fromState [MainMenuState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the map choice menu state, specifying the instance of the main menu state that it is being entered from. Pre-condition: Functional State not entered before Post-condition: Functional Map choice menu state properly initialized Post-condition: Functional Parent state NOT stored
handleFallback (*PlayState&*)	«Manager» public abstract: *void*	param: childState [PlayState& - in] Notifies the state about being "fallbacked" into, from a playing state. Takes no action. (Could possibly forward fallback to the main menu state.) Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way.
generateWorld (*std::string&*)	«Manager» private: *void*	param: mapName [std::string& - in] Generates a world instance from the map with the specified name and stores it in the state. Should be called by EnterSinglePlayerStateAction only. Action: Uses WorldMapRegistry...
MapChoiceMenuState (*PlayState&*)	public:	param: playingState [PlayState& - in] Constructs a map choice menu state, specifying what

		play state to enter when a map is selected.

State::MapChoiceMenuState::EnterPlayingStateAction

public Class

Extends: EnterStateAction. :

State::MapChoiceMenuState::EnterPlayingStateAction Methods

Method	Type	Notes
EnterPlayingStateAction (*PlayState&, std::string&*)	public:	param: playState [PlayState& - in] param: mapName [std::string& - in] Constructs a new action for entering a single player game state on the specified map.
trigger ()	«Event» public abstract: *void*	Causes the map choice state to generate a world from the chosen map and enters the specified game playing state. Post-condition: Functional A playing game state is entered Post-condition: Functional World generated in map choice menu state Action: Uses the Game singleton's enterState function.

State::ControlsMenuState

public Class {leaf}

Extends: MenuState. :

State::ControlsMenuState Methods

Method	Type	Notes
enterFrom (*GameState**)	«Manager» public abstract: *void*	param: fromState [GameState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the game controls configuration state from any other state (except for this one, of course). Pre-condition: Functional State not entered before Post-condition: Functional Game controls menu state properly initialized Post-condition: Functional Parent state NOT stored
handleFallback (*ControlsConfigMenuState&*)	«Manager» public abstract: *void*	param: childState [ControlsConfigMenuState& - in] Handles fallback from game controls config menu state. Other states won't be supported to fallback from. Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Action: Does nothing, except for restricting states to accept fallback from.

State::ControlsConfigMenuState

public Class {leaf}

Extends: MenuState. : The state of the controls configuration loop, i.e., where each control for a single player is reconfigured on a step-by-step basis.

State::ControlsConfigMenuState Methods

Method	Type	Notes
ControlsConfigMenuState ()	public:	
enterFrom (*GameState**)	«Manager» public abstract: *void*	param: fromState [GameState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the game state, and, when possible, lets it know what state it was entered from (i.e., its parent state). Takes a pointer because we need to be able to accept 0 as value (when entering main menu, for example). Pre-condition: Functional State not entered before Post-condition: Functional Game state is properly initialized Post-condition: Functional Parent state NOT stored
handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Notifies this state about a fallback to it from a child state, providing the child state instance. The child game state will be checked, and, when required, proper action will be taken (for example, if the child game state was a pause menu, it may have ordered a "quit", and we should realize it). Pre-condition: Functional This game state has been

		entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Orders from child state are realized - Any orders from the child state shall be realized. For example, if the child state was a pause menu, and it concluded that the game session should quit, then we must realize that order. Action: does nothing by default
tick ()	«Event» public abstract: *void*	Makes the menu state to self-update according to the status of the controller module. Action: Polls controller and updates contained menu if necessary.

State::HighScoreMenuState

public Class {leaf}
Extends: MenuState. :

State::HighScoreMenuState Methods

Method	Type	Notes
enterFrom (*MainMenuState**)	«Manager» public abstract: *void*	param: fromState [MainMenuState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the high score menu state, specifying the main menu state from which it was entered. Pre-condition: Functional State not entered before

		Post-condition: Functional Back button name is "Back" - Since we're going back to the main menu. Post-condition: Functional High score menu state is initialized properly - The game state was initialized. Post-condition: Functional Parent state NOT stored
enterFrom (*SinglePlayerPlayState**)	«Manager» public abstract: *void*	param: fromState [SinglePlayerPlayState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the high score menu state, specifying the single player play state from which it was entered. Pre-condition: Functional State not entered before Post-condition: Functional Back button name is "Continue" - Since we're continuing, after game over (without game over we wouldn't be here). Post-condition: Functional High score menu state is initialized properly - The game state was initialized. Post-condition: Functional Parent state NOT stored
handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Throws illegal fallback exception, as this state is a leaf state. Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Illegal fallback exception thrown Action: Throws illegal fallback exception.

State::HelpMenuState

public Class {leaf}
Extends: MenuState. :

State::HelpMenuState Methods

Method	Type	Notes
enterFrom (*GameState**)	«Manager» public abstract: *void*	param: fromState [GameState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the help menu, specifying the game state from which it was entered. Pre-condition: Functional State not entered before Post-condition: Functional Help menu state is properly initialized Post-condition: Functional Parent state NOT stored
handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Throws an exception, as there is no valid fallback to the help menu state. Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Illegal fallback exception thrown Action: Throws illegal falback exception.

State::PauseMenuState

public Class {leaf}

Extends: MenuState. :

State::PauseMenuState Attributes

Attribute	Type	Notes
m_quitGame	private : *bool*	Tells whether the pause menu button "quit game" was pressed or not. Initial Value: false;

State::PauseMenuState Methods

Method	Type	Notes
doQuitGame ()	«Access» public: *bool*	Tells whether the pause menu orders its parent play state quit or not.
enterFrom (*PlayState**)	«Manager» public abstract: *void*	param: fromState [PlayState* - in] Takes a pointer because we need to be able to accept 0 as value. Enters the pause menu state, specifying the play state from which it is being entered. Does nothing, except for setting up restrictions for what states to accept enter from. Pre-condition: Functional State not entered before Post-condition: Functional Parent state NOT stored Post-condition: Functional Pause menu state is properly initialized

handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Throws illegal fallback, as this function overload catches all unsupported child states. To support fallback from a state, a corresponding function overload can be added later. Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Illegal fallback exception thrown

State::PauseMenuState::LeavePauseMenuStateAction

public Class

Extends: LeaveStateAction. : Responsible for handling a press on the pause menu's "Quit Game" button.

State::PauseMenuState::LeavePauseMenuStateAction Methods

Method	Type	Notes
trigger ()	«Event» public abstract: *void*	Finishes the current game state. Post-condition: Functional Current game state is set to finished Post-condition: Functional Pause menu state updated about quit request Action: Updates pause menu state, setting m_quitGame=true

		and calls LeaveStateAction::trigger().
LeavePauseMenuStateAction (*PauseMenuState**)	public:	param: pauseState [PauseMenuState* - in] Constructs a quit action for the pause menu state's menu "quit" button.

State::PlayState

public abstract Class

Extends: GameState. : The game state to be in when a game session is active. Responsible for setting up and managing the game engine.

State::PlayState Attributes

Attribute	Type	Notes
m_awaitsFinish	protected : *bool*	Tells whether the play state is awaiting finish (i.e., "press any key to continue" when game was over) or not. Initial Value: false;

State::PlayState Methods

Method	Type	Notes
getEngine ()	«Access» public: *Engine&*	Returns the play state's game engine.
isAwaitingFinish ()	«Access» public: *bool*	Tells whether the play state is awaiting finish on game over, i.e., the "press any key to continue" stuff, or not.
enterFrom	«Manager» public abstract:	param: fromState [GameState* - in] Takes a pointer because we need to be able to

(*GameState**)	*void*	accept 0 as value. Handles unsupported parent game states by throwing an exception when called. The reason is that we have to read a specific parent type in order to get the world to play on, for example. Pre-condition: Functional State not entered before Post-condition: Functional Invalid parent state exception thrown Action: Throws invalid parent state exception.
handleFallback (*GameState&*)	«Manager» public abstract: *void*	param: childState [GameState& - in] Notifies this state about a fallback to it from a child state, providing the child state instance. The child game state will be checked, and, when required, proper action will be taken (for example, if the child game state was a pause menu, it may have ordered a "quit", and we should realize it). Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Illegal fallback exception thrown Action: Throws illegal fallback exception.

State::SinglePlayerPlayState

public Class {leaf}

Extends: PlayState. : Responsible for starting a single player game and keeping it going until game over rules are met, resulting in entering the high score menu.

State::SinglePlayerPlayState Methods

Method	Type	Notes
tick ()	«Event» public abstract: *void*	Checks whether the game is over or not (by checking the player and his ship) and forwards tick to engine. If finish is awaited, then nothing happens here. Pre-condition: Functional The state was entered Post-condition: Functional Engine received tick call Post-condition: Functional Game play continuation validated - Checked whether the game is over or not.
enterFrom (*MapChoiceMenuState**)	«Manager» public abstract: *void*	param: fromState [MapChoiceMenuState* - in] Takes a pointer because we need to be able to accept 0 as value. Initializes the single player play state, fetching its world instance from the provided map choice menu state. Pre-condition: Functional Parent state provides a world instance Pre-condition: Functional State not entered before Post-condition: Functional Engine instance created Post-condition: Functional Invalid parent state exception thrown Post-condition: Functional Parent state NOT stored Post-condition: Functional World read and sent to engine Action:

		Reads world and self-initializes.
handleFallback (*PauseMenuState&*)	«Manager» public abstract: *void*	param: childState [PauseMenuState& - in] Handles fallback from the pause menu state, checking whether it ordered a game quit or not. Pre-condition: Functional This game state has been entered Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Input manager reset - The input manager must be reset when falling back from a child state. The reason for this is that otherwise all controls that were active before entering the child state would continue to be active when falling back to this state, regardless of whether they *are* active (i.e., keys are pressed) or not. Example: If pressing pause and at the same time a ship throttle control is active, then in the pause menu we release the key that activated the throttle control, and then finally, when we go back to the playing state, the ship throttles without the throttle key being pressed. Post-condition: Functional Orders from pause menu realized - If the pause menu ordered a game quit, then it should be realized here, if not, then the game should continue. Action: Handles fallback from the pause menu state, checking whether it ordered a game quit or not.
handleFallback (*HighScoreMenuState&*)	«Manager» public abstract: *void*	param: childState [HighScoreMenuState& - in] Handles fallback from a high score menu. The handling will always result in this state being

		finished, as a preceding enter into the high score menu state from *this* state, was a result of a game over. Pre-condition: Functional This game state has been entered Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Single player state finishes Action: Finishes this state.
triggerGameOver ()	«Manager» private: *void*	Manages a game over by saving a high score (if any) and entering the high score menu state. Called by tick when game over rules are met. Post-condition: Functional High score menu state entered Post-condition: Functional High score saved in registry if high enough Action: Saves high score and entes high score menu.

State::TwoPlayersPlayState

public Class {leaf}

Extends: PlayState. : Responsible for starting a two player game and keeping it going until game over rules are met.

State::TwoPlayersPlayState Methods

Method	Type	Notes
tick ()	«Event» public abstract: *void*	Checks whether the game is over or not (by checking the players and their ships) and forwards tick to engine. Pre-condition: Functional The state was entered Post-condition: Functional Engine received tick call Post-condition: Functional Game play continuation validated - Checked whether the game is over or not.
enterFrom (*MapChoiceMenuState**)	«Manager» public abstract: *void*	param: fromState [MapChoiceMenuState* - in] Takes a pointer because we need to be able to accept 0 as value. Initializes the two players play state, fetching its world instance from the provided map choice menu state. Pre-condition: Functional State not entered before Post-condition: Functional Parent state NOT stored Action: Reads world and self-initializes.
handleFallback (*PauseMenuState&*)	«Manager» public abstract: *void*	param: pauseState [PauseMenuState& - in] Handles fallback from the pause menu state, checking whether it ordered a game quit or not. Pre-condition: Functional This game state has been entered before Post-condition: Functional Child state NOT stored - The child state must not be stored in any way. Post-condition: Functional Input manager reset - The input manager must be reset when falling back from a child state. The reason for this is that otherwise all controls that were active before entering the child

		state would continue to be active when falling back to this state, regardless of whether they *are* active (i.e., keys are pressed) or not. Post-condition: Functional Orders from child state are realized - Any orders from the child state shall be realized. For example, if the child state was a pause menu, and it concluded that the game session should quit, then we must realize that order. Action: Handles fallback from the pause menu state, checking whether it ordered a game quit or not.
triggerGameOver ()	«Manager» private: *void*	Requests user input to continue to the main menu. Called by tick when game over rules are met.

World

Contains the class structure representing the game world and its objects, such as ships and planets, for example.

World::Asteroid

public Class

Extends: DestroyableObject, MovableObject. : Representation of an asteroid flying around randomly in the game world.

World::Asteroid Attributes

Attribute	Type	Notes
INIT_STRENGTH	private const static : *unsigned int*	Initial strength of an asteroid.
m_rotationSpeed	private : *float*	The speed at which the asteroid rotates. If positive, it rotates clockwise, if negative, it rotates counter-clockwise.

World::Asteroid Methods

Method	Type	Notes
Asteroid (*float*)	public:	param: rotationSpeed [float - in] Constructs an asteroid, specifying the rotation direction as well as its speed.
tick ()	«Event» public abstract:	Updates position and orientation (i.e., rotation).

	void	Post-condition: Functional Time-Dependent State Updated - The time-dependent state of the object is updated. Action: Updates position and orientation (i.e., rotation).

World::DestroyableObject

public abstract Class: World objects implementing this interface are considered destroyable, i.e., it is possible to destroy them by causing enough damage to them. Note that destruction has nothing to do with removal from the world.

World::DestroyableObject Attributes

Attribute	Type	Notes
m_strength	private : *unsigned int*	The remaining strength of the object.

World::DestroyableObject Methods

Method	Type	Notes
isDestroyed ()	«Access» public abstract: *bool*	Tells whether the world object is destroyed or not.
reduceStrength (*unsigned int*)	«Manager» public abstract: *void*	param: amount [unsigned int - in] Reduces the world object's strength by the specified amount.

destroy ()	«Manager» public abstract: *void*	Destroys the world object, i.e., causes maximal damage, resulting in the object being destroyed.
DestroyableObject (*unsigned int*)	public:	param: strength [unsigned int - in] Constructs a destroyable world object with the specified strength.

World::FuelItem

public Class

Extends: Item. : Representation of an item that contains a fuel powerup.

World::FuelItem Attributes

Attribute	Type	Notes
m_fuelAmount	private : *unsigned int*	The fuel amount that the fuel item contains.

World::FuelItem Methods

Method	Type	Notes
FuelItem (*unsigned int*)	public:	param: amount [unsigned int - in] Constructs a new fuel item, specifying the fuel amount it should contain.
getFuelAmount ()	«Access» public: *unsigned int*	Returns the amount of fuel in this item.

World::Item

public abstract Class

Extends: StaticObject. : Contains common behavior and properties of items occuring in the game world.

World::Item Methods

Method	Type	Notes
Item ()	public:	Constructs an item with a pre-defined mass (i.e. 0, as it shouldn't be affected by gravities) and shape (all items have the same size). Post-condition: Functional Parents setup properly - Parent constructors called with correct values.

World::LaserProjectile

public Class

Extends: Projectile. : Representation of a laser projectile, which is not affected by gravities.

World::LaserProjectile Attributes

Attribute	Type	Notes
SET_OFF_SPEED	private const static : *unsigned int*	The movement speed (i.e., the magnitude of the movement vector) at which the laser is set off, relative to the ship firing it.

World::LaserProjectile Methods

Method	Type	Notes
LaserProjectile (*Ship*)	public:	param: ship [Ship - in] Constructs a laser projectile, specifying the ship from which it originates. Action: Laser movement is set relative to the speed and direction of the ship.
getMaxDamageStrength ()	«Access» public abstract: *unsigned int*	Returns the maximal damage strength of the laser.

World::MissileItem

public Class

Extends: Item. : Representation of an item that contains ship missiles.

World::MissileItem Attributes

Attribute	Type	Notes
m_numMissiles	private : *unsigned int*	Numer of missiles in this weapon item.

World::MissileItem Methods

Method	Type	Notes
MissileItem (*unsigned int*)	public: *void*	param: numMissiles [unsigned int - in] Constructs a new missile item, specifying how many

		missiles it should contain.
getMissileCount ()	«Access» public: *unsigned int*	Returns the number of missiles in this item.

World::MissileProjectile

public Class

Extends: DestroyableObject, Projectile. : Representation of a gravity-affectable missile projectile.

World::MissileProjectile Attributes

Attribute	Type	Notes
INIT_STRENGTH	private const static : *unsigned int*	Initial strength of a missile.
SET_OFF_SPEED	private const static : *unsigned int*	The movement speed (i.e., the magnitude of the movement vector) at which the missile is set off, relative to the ship firing it.

World::MissileProjectile Methods

Method	Type	Notes
MissileProjectile (*Ship*)	public:	param: ship [Ship - in] Constructs a missile, specifying the ship from which it originates. Action: Missile movement is set relative to the speed and

		direction of the ship.
getMaxDamageStrength ()	«Access» public abstract: *unsigned int*	Returns the maximal damage strength of the projectile.
tick ()	«Event» public abstract: *void*	Updates the position and maintains orientation so that it follows the missile's movement. Post-condition: Functional Position Updated - If the ship is moving, then its position has been updated according to the magnitude of the movement vector. Action: Updates the position and maintains orientation so that it follows the missile's movement.

World::MovableObject

public abstract Class

Extends: WorldObject. : Contains common behavior and properties of movable game world objects.

World::MovableObject Methods

Method	Type	Notes
tick ()	«Event» public abstract: *void*	Updates the position of the movable object. Post-condition: Functional Position Updated - If the ship is moving, then its position has been updated according to the magnitude of the movement vector. Action: updates position

setMovement (*Vector2d*)	«Manager» public abstract: *void*	param: vector [Vector2d - in] Changes the movement of the movable world object. A change here may result in a change in the object's orientation, depending on the object type. Post-condition: Functional Movement Changed - The object's movement vector has been replaced with a new one.
getMovement ()	«Access» public: *Vector2d*	Returns the movement vector of the movable object. The magnitude represents the speed and the angle represents the direction of movement.

World::Planet

public Class

Extends: StaticObject. : Representation of a planet.

World::Planet Attributes

Attribute	Type	Notes
TYPE_RED	public const static : *unsigned short*	A red planet. Initial Value: 1;
TYPE_BLUE	public const static : *unsigned short*	A blue planet. Initial Value: 2;
TYPE_GREEN	public const static : *unsigned short*	A green planet. Initial Value: 3;
m_type	private :	The planet's type.

	unsigned short	

World::Planet Methods

Method	Type	Notes
Planet (*double, float, unsigned short*)	public:	param: mass [double - in] param: radius [float - in] param: type [unsigned short - in] See TYPE_* constants. Constructs a new planet specifying its mass and radius. Post-condition: Functional Circular shape created - A circular shape of the planet is created and stored in the instance. Action: Creates and forwards a circular shape to the parent constructor, out of the received arguments during initialization.
getType ()	«Access» public: *unsigned short*	Returns the planet's type, which will be one of the public TYPE_* constants.

World::Projectile

public abstract Class

Extends: MovableObject. : Contains common behavior and properties of weapon projectiles fired by a ship.

World::Projectile Methods

Method	Type	Notes
Projectile (*Ship*)	public:	param: ship [Ship - in] Constructs a projectile, specifying the ship from which it originates.
getFiringShip ()	«Access» public: *Ship*	Returns the ship that fired the projectile.
getMaxDamageStrength ()	«Access» public abstract: *unsigned int*	Returns the maximal damage strength of the projectile.

World::Ship

public Class

Extends: DestroyableObject, MovableObject. : Representation of the ship that a game player will control.

World::Ship Attributes

Attribute	Type	Notes
INIT_STRENGTH	private const static : *unsigned int*	Initial strength of a ship.
MAX_SPEED	private const static : *unsigned int*	The maximum movement speed of a ship, i.e., maximum magnitude of the movement vector.
m_limitFuel	private : *int*	Tells whether the ship's fuel is limited (true) or unlimited (false).

		Initial Value: true;
m_limitLasers	private : *bool*	Tells whether the ship's lasers are limited (true) or unlimited (false). Initial Value: true;
m_limitMissiles	private : *bool*	Tells whether the ship's missiles are limited (true) or unlimited (false). Initial Value: true;
m_fuelAmount	private : *float*	The ship's current fuel amount percentage. This is irrelevant if the fuel is unlimited.
m_lasersCount	private : *unsigned int*	The ship's current number of lasers. This is irrelevant if the lasers are unlimited.
m_missilesCount	private : *unsigned int*	The ship's current number of missiles. This is irrelevant if the missiles are unlimited.
m_laserGunChargeLevel	private : *float*	Defines the laser gun charge level. When 1.0, the laser gun is considered charged. Further, the values must not go below 0 or over 1.0. Initial Value: 1.0;
m_missileLauncherChargeLevel	private : *float*	Defines the missile launcher charge level. When 1.0, the missile launcher is considered charged. Further, the values must not go below 0 or over 1.0. Initial Value: 1.0;
m_isTurningLeft	private : *bool*	Tells whether the ship is turning left or not. Initial Value: false;
m_isTurningRight	private : *bool*	Tells whether the ship is turning right or not. Initial Value: false;
m_isThrottling	private : *bool*	Tells whether the ship is throttling or not. Initial Value: false;
m_laserGunOn	private :	Tells whether the ship's laser gun is on or off.

	bool	Initial Value: false;
m_missileLauncherOn	private : *bool*	Tells whether the ship's missile launcher is on or off. Initial Value: false;

World::Ship Methods

Method	Type	Notes
getFuelLevel ()	«Access» public: *unsigned int*	Returns the ship's fuel level.
getMissilesCount ()	«Access» public: *unsigned int*	Returns the ship's amount of missiles.
getLasersCount ()	«Access» public: *unsigned int*	Returns the ship's amount of lasers.
hasLimitedFuel ()	«Access» public: *bool*	Tells whether the ship's fuel supply is limited.
hasLimitedLasers ()	«Access» public: *bool*	Tells whether the ship's laser supply is limited.
hasLimitedMissiles ()	«Access» public: *bool*	Tells whether the ship's missile supply is limited.
tick ()	«Event» public abstract: *void*	Updates the position of the ship as well as charges its weapons (so that there is a fire delay). Post-condition: Functional Position Updated - If the ship is moving, then its position has been updated according to the magnitude of the movement vector. Action:

		updates position and charges weapons
setEventManager (*EventManager**)	«Manager» public: *void*	param: eventManager [EventManager* - in] Sets the ship's event manager.
setLimitLasers (*bool*)	«Manager» public: *void*	param: doLimit [bool - in] Sets the ship's laser limit on/off.
toggleThrottle (*bool*)	«Manager» public: *void*	param: doThrottle [bool - in] Toggles ship's throttle on or off.
setLimitFuel (*bool*)	«Manager» public: *void*	param: doLimit [bool - in] Sets the ship's fuel limit on/off.
setLimitMissiles (*bool*)	«Manager» public: *void*	param: doLimit [bool - in] Sets the ship's missile limit on/off.
toggleTurnLeft (*bool*)	«Manager» public: *void*	param: doTurn [bool - in] Toggles ship's left turning on or off. Post-condition: Functional If toggled on, right turning toggled off - If the ship is set to turn left, then it can't turn right at the same time.
toggleTurnRight (*bool*)	«Manager» public: *void*	param: doTurn [bool - in] Toggles ship's right turning on or off. Post-condition: Functional If toggled on, left turning toggled off - If the ship is set to turn right, then it can't turn left at the same time.

toggleLaserGun (*bool*)	«Manager» public: *void*	param: doFire [bool - in] Toggles the ship's laser gun on/off. If on, then a laser is fired as soon as the laser gun has finished charging. If no lasers are left, the none will be fired.
toggleMissileLauncher (*bool*)	«Manager» public: *void*	param: doFire [bool - in] Toggles the ship's missile launcher on/off. If on, then a missile is fired as soon as the missile launcher has finished charging. If no missiles are left, the none will be fired.
fireLaser ()	«Manager» private: *void*	Fires a missile if finished charging. Pre-condition: Functional Event manager is present Pre-condition: Functional Laser charging complete Post-condition: Functional Laser discharged - Once fired, the laser gun must be charged again, therefore we have to discharge it in order to enforce that. Post-condition: Functional Laser fire event sent to event manager - A laser with the same orientation as the ship and with a movemet relative to the ship's movement used as an argument to the event constructor.
chargeLaserGun (*float*)	«Manager» private: *void*	param: progress [float - in] Charges the laser gun according to the specified progress. When the charge level reaches 1.0, the laser gun is considered charged. Post-condition: Functional 0 <= Charge level <= 1.0 - Charge level must not go over 1.0, which can be seen as 100%, nor below 0.

fireMissile ()	«Manager» private: *void*	Fires a missile if finished charging. Pre-condition: Functional Event manager is present Pre-condition: Functional Missile charging complete Post-condition: Functional Missile discharged - Once fired, the missile launcher must be charged again, therefore we have to discharge it in order to enforce that. Post-condition: Functional Missile fire event sent to event manager - A missile with the same orientation as the ship and with a movemet relative to the ship's movement used as an argument to the event constructor.
chargeMissileLauncher (*float*)	«Manager» private: *void*	param: progress [float - in] Charges the missile launcher according to the specified progress. When the charge level reaches 1.0, the missile launcher is considered charged. Post-condition: Functional 0 <= Charge level <= 1.0 - Charge level must not go over 1.0, which can be seen as 100%, nor below 0.

World::SpawnPoint

public Class

Extends: StaticObject. : Defines a point in which a world object may appear, telling whether the area is free of obstacles or not.

World::SpawnPoint Attributes

Attribute	Type	Notes

m_isFree	private : *bool*	Whether the spawn point is free or not.

World::SpawnPoint Methods

Method	Type	Notes
SpawnPoint ()	public:	Constructs a spawn point with a circular shape and mass == 0. Post-condition: Functional Spawn point has a circular shape - A circular shape of the spawn point is created and stored in the instance.
isFree ()	«Access» public: *bool*	Returns true if the spawn point is free to use, i.e., that no other world object spawned in it for a certain amount of time, otherways false is returned.
toggleAvailability (*bool*)	«Manager» public: *void*	param: setFree [bool - in] Toggles the availability of the spawn point by setting free to false or true.

World::StaticObject

public abstract Class

Extends: WorldObject. : Contains common behavior and properties of static game world objects.

World::World

public Class

Implements: GameEventListener, Tickable. : Representation of the whole game

world, containing all world objects that are supposed to exist at a certain moment during a game session.

World::World Attributes

Attribute	Type	Notes
m_time	private : *unsigned int*	The number of ticks received by the world - the world's time. Initial Value: 0;

World::World Methods

Method	Type	Notes
World (*BoundaryStrategy&*)	public:	param: boundaryStrategy [BoundaryStrategy& - in] Constructs a world with the specified boundary strategy and an event manager to use for events triggered in or by the world. Post-condition: Functional Boundary strategy inserted into m_strategies Post-condition: Functional Boundary strategy pointer stored in m_boundaryStrategy
getTime ()	«Access» public: *unsigned int*	Returns the world's current time, i.e., amount of received ticks.
getBoundaryStrategy ()	«Access» public const: *BoundaryStrategy&*	Returns the boundary strategy used by the world.
	«Access»	Returns an iterator of the world objects vector.

getWorldObjectsIterator ()	public: *std::vector<WorldObject>::iterator*	
getSpawnPointsIterator ()	«Access» public: *std::vector<SpawnPoint*>::iterator*	Returns an iterator of the spawn points vector.
tick ()	«Event» public abstract: *void*	Calls tick() on all available world objects and applies available strategies on self. Post-condition: Functional m_time incremented with 1 Post-condition: Functional World objects have had their tick() called - Each of the contained world objects have received a call to their tick() function. Post-condition: Functional World Strategies Applied on World - Available strategies have been applied on the world. Action: runs tick() on all contained world objects and applies available world strategies
notifyEvent (*WorldEvent&*)	«Event» public abstract: *void*	param: ev [WorldEvent& - in] Notifies the world about an occuring event. Currently does nothing for generic events.
notifyEvent (*ProjectileFireEvent&*)	«Event» public abstract: *void*	param: ev [ProjectileFireEvent& - in] Handles a projectile fire event by inserting a copy of the fired projectile into the world. Action:

		Inserts projectile into the world.
addStrategy (*WorldStrategy&*)	«Manager» public: *void*	param: strategy [WorldStrategy& - in] Adds a world strategy to apply on the world on each tick. Post-condition: Functional Strategy inserted into m_strategies
addStrategy (*BoundaryStrategy&*)	«Manager» public: *void*	param: strategy [BoundaryStrategy& - in] Adds a world boundary strategy to apply on the world on each tick. Post-condition: Functional Strategy inserted into m_strategies Post-condition: Functional The boundary strategy's reference stored separately - Boundary strategy stored in a "shortcut pointer", so that it can be quickly retrieved.
getMovableObjectsIterator ()	«Access» public: *std::vector<MovableObject*>::iterator*	Returns an iterator of the movable world objects vector.
queueInsert (*Ship**)	«Manager» public: *void*	param: ship [Ship* - in] Having a pointer, we let the caller have a way to access the ship. However, the caller is then responsible for listening for ship removal events. Queues an insert of a ship into the world, waiting for an available spawn point. Pre-condition: Functional The ship is not present in the world

queueInsert (*Asteroid&*)	«Manager» public: *void*	param: asteroid [Asteroid& - in] Queues an insert of an asteroid into the world, waiting for an available spawn point. Pre-condition: Functional The asteroid is not present in the world
tryInsertShips ()	«Manager» private: *void*	Searches for free spawn points and, if found, inserts as many of the the queued ship(s) as there are available spawn points. Each ship will have its event manager (the same as the world uses) set just before being inserted. Post-condition: Functional Insertion event(s) cascaded in the event manager - For each successful insertion, an insertion event shall be cascaded through the event manager. Post-condition: Functional Ship's event manager is set to the world's
tryInsertAsteroids ()	«Manager» private: *void*	Searches for free spawn points and, if found, inserts as many of the the queued asteroid(s) as there are available spawn points. Post-condition: Functional Insertion event(s) cascaded in the event manager - For each successful insertion, an insertion event shall be cascaded through the event manager.
insert (*WorldObject&, Coord2d&*)	«Manager» private: *void*	param: wo [WorldObject& - in] param: position [Coord2d& - in] Inserts a world object into the world, specifying the position that it should initially appear at.

		Pre-condition: Functional Specified position is within boundaries - The specified coordinate must be within the boundaries of the world. Pre-condition: Functional The object is not present in the world - The object must not be present in the world already. Post-condition: Functional The object is placed at the specified coordinate - The object is inserted into the world at the specified position and the world object's internal position is updated. Post-condition: Functional World object inserted into m_worldObjects
insert (*MovableObject&, Coord2d&*)	«Manager» private: *void*	param: wo [MovableObject& - in] param: position [Coord2d& - in] Inserts a movable world object into the world, specifying the position that it should initially appear at. Pre-condition: Functional Specified position is within boundaries - The specified coordinate must be within the boundaries of the world. Pre-condition: Functional The object is not present in the world - The object must not be present in the world already. Post-condition: Functional Movable object inserted into m_movableObjects Post-condition: Functional Movable object inserted into m_worldObjects Post-condition: Functional The object is placed at the specified coordinate - The object is inserted into the world at the specified position and the world object's internal position is updated. Action:

		Inserts a pointer to the movable object into m_movableObjects.
insert (*SpawnPoint&, Coord2d&*)	«Manager» private: *void*	param: wo [SpawnPoint& - in] param: position [Coord2d& - in] Inserts a spawn point object into the world, specifying the position that it should appear at. Pre-condition: Functional Specified position is within boundaries - The specified coordinate must be within the boundaries of the world. Pre-condition: Functional The object is not present in the world - The object must not be present in the world already. Post-condition: Functional Spawn point inserted into m_spawnPoints Post-condition: Functional Spawn point inserted into m_worldObjects Post-condition: Functional The object is placed at the specified coordinate - The object is inserted into the world at the specified position and the world object's internal position is updated. Action: Inserts a pointer to the spawn point object into m_spawnPoints.
insert (*Item&*)	«Manager» private: *void*	param: item [Item& - in] Inserts an item on a random spawn point in the world, not caring whether the spawn point is free of obstacles or not. Post-condition: Functional Insertion event cascaded - Insertion event cascaded through the world's event manager.

queueRemoval (*WorldObject&*)	«Manager» public: *void*	param: wo [WorldObject& - in] Queues removal of a world object, deferring the actual removal to the final step of the currently ran tick(). Further, it sends out a RemovalEvent to the event manager. Pre-condition: Functional The object is present in the world - The object must be present in the world in order to be removed. Post-condition: Functional Removal event sent out to event manager Post-condition: Functional World object pointer inserted into m_removeQueue Action: Cascades a removal event through the event manager.
performRemovals ()	«Manager» private: *void*	Performs the queued removals, if any.
remove (*WorldObject&*)	«Manager» private: *void*	param: wo [WorldObject& - in] Removes a world object from the world. Pre-condition: Functional The object is present in the world - The object must be present in the world in order to be removed. Post-condition: Functional The object is removed from m_worldObjects
remove (*SpawnPoint&*)	«Manager» private: *void*	param: wo [SpawnPoint& - in] Removes the spawn point object from the world,

		together with its shortcut in the m_spawnPoints vector. Pre-condition: Functional The object is present in the world - The object must be present in the world in order to be removed. Post-condition: Functional The spawn point is removed from m_spawnPoints Post-condition: Functional The spawn point is removed from m_worldObjects Action: Removes pointer from m_spawnPoints.
remove (*MovableObject&*)	«Manager» private: *void*	param: wo [MovableObject& - in] Removes the movable object from the world, together with its shortcut in the m_movableObjects vector. Pre-condition: Functional The object is present in the world - The object must be present in the world in order to be removed. Post-condition: Functional The movable object is removed from m_movableObjects Post-condition: Functional The movable object is removed from m_worldObjects Action: Removes pointer from m_movableObjects.

World::WorldObject

public abstract Class

Implements: Tickable. : Contains common behavior and properties of game world objects.

World::WorldObject Attributes

Attribute	Type	Notes
m_mass	protected : *double*	The mass of the world object. This may be used for gravity calculations. In order to have an object that is not affected by gravities, it's mass should be zero.

World::WorldObject Methods

Method	Type	Notes
WorldObject (*double, Shape*)	public:	param: mass [double - in] param: shape [Shape - in] Constructs a world object, specifying its shape.
tick ()	«Event» public abstract: *void*	Notifies the object about that the time is being incremented with one time unit. Pre-condition: Functional Registered as Tickable - The object must be registered as a tickable object in the game engine in order to have its tick() function called. Post-condition: Functional Time-Dependent State Updated - The time-dependent state of the object is updated. Action: do nothing by default
setPosition (*Coord2d&*)	«Manager» public abstract: *void*	param: coord [Coord2d& - in] Set the position of the object in the world, according

		to the object's centre.
getPosition ()	«Access» public const: *Coord2d&*	Returns the position of the object in the world, according to the object's centre.
getOrientation ()	«Access» public const: *Vector2d&*	Returns the vector of the object's orientation, i.e., the direction in which it's "pointing", as well as its boundary "circle". The vector's magnitude represents the radius of the boundary circle, and the angle represents the direction in which the object is "pointing".
getShape ()	«Access» public const: *Shape&*	Returns the shape of the object, i.e., a representation of its spacial form and size, for example a circular shape with a radius.
getMass ()	«Access» public: *double*	Returns the mass of the world object.

World::WorldStrategy

public abstract «interface» Interface: Interface implemented by classes that need to manage the world by applying self-defined strategies on it.

World::WorldStrategy Interfaces

Method	Type	Notes
applyWorldStrategy (*World*)	«Manager» public abstract: *void*	param: world [World - in] Applies a world strategy on a world instance. Post-condition: Functional World state changed according to strategy

Registry

Responsible for holding, managing persistence for and providing global data to other modules.

Registry::ConfigRegistry

public «singleton» Class

Extends: Registry. : Responsible to provide and handle persistence for the game configuration.

Registry::ConfigRegistry Attributes

Attribute	Type	Notes
m_singleton	private static : *ConfigRegistry*	The singleton instance.
m_controls	private : *std::map<unsigned short, unsigned short>*	Maps static control aliases to keyboard key codes.
m_localPlayer1Controls	private : *std::map<unsigned short, unsigned short>*	Maps configurable control aliases for the first local player to keyboard key codes.
m_localPlayer2Controls	private : *std::map<unsigned short, unsigned short>*	Maps configurable control aliases for the second local player to keyboard key codes.
m_gfxRoot	private : *std::string*	Path to the directory that contains all graphics.
m_sfxRoot	private : *std::string*	Path to the directory that contains all sounds.

Registry::ConfigRegistry Methods

Method	Type	Notes
ConfigRegistry ()	private:	Constructs a config registry instance.
instance ()	«Helper» public static: *ConfigRegistry &*	Returns the singleton instance of the config registry.
getControlKey (*unsigned short*)	«Access» public: *unsigned short*	param: controlAlias [unsigned short - in] One of the CTRL_* constants defined in Controller::InputListener. Returns the corresponding keyboard key for the provided control alias. Action: Reads m_controls
getLocalPlayer1ControlKey (*unsigned short*)	«Access» public: *unsigned short*	param: controlAlias [unsigned short - in] One of the CTRL_* constants defined in Controller::InputListener. Returns the corresponding keyboard key for the provided control alias. Action: Reads m_primaryLocalPlayerControls
getLocalPlayer2ControlKey (*unsigned short*)	«Access» public: *unsigned short*	param: controlAlias [unsigned short - in] One of the CTRL_* constants defined in Controller::InputListener. Returns the corresponding keyboard key for the provided control alias.

		Action: Reads m_secondaryLocalPlayerControls
getGfxRoot ()	«Access» public const: *std::string&*	Returns the path to the directory where all graphics reside.
getSfxRoot ()	«Access» public const: *std::string&*	Returns the path to the directory where all sounds reside.

Registry::HighScoreRegistry

public «singleton» Class

Extends: Registry. : Responsible for storing high scores in a file, as well as deciding what scores should be considered being high scores.

Registry::HighScoreRegistry Attributes

Attribute	Type	Notes
m_singleton	private static : *HighScoreRegistry*	The singleton instance. Initial Value: 0;
m_highScores	private : *std::multimap<std::string, unsigned int>*	Contains the names of the players achieving a high score together with the high score they achieved.

Registry::HighScoreRegistry Methods

Method	Type	Notes
instance ()	«Helper» public static:	Returns the singleton instance of the high score

	HighScoreRegistry&	registry.
getHighScores ()	«Access» public: *std::multimap<std::string, unsigned int>*	Returns a multimap containing the top ten high scores and the players that achieved them.
trySetHighScore (*std::name&, unsigned int*)	«Manager» public: *bool*	param: playerName [std::name& - in] param: score [unsigned int - in] Tries to set the specified score as a high score and will do so if the score is high enough (top 10), returning true. If the high score is not high enough, false will be returned.
HighScoreRegistry ()	private:	Constructs a high score registry. Action: Loads data from high score file.

Registry::Registry

public abstract Class: Abstract registry class, holding file management functions.

Registry::Registry Methods

Method	Type	Notes
saveToFile (*std::string&, std::string&*)	«Manager» protected: *void*	param: text [std::string& - in] param: fileName [std::string& - in] Saves a text string to a file with the specified file name. If the file doesn't exist, then it will be created.

readFromFile (*std::string&*)	«Manager» protected: *std::string**	param: fileName [std::string& - in] Reads the content of a file and returns every row of it in a string array.

Registry::WorldMapRegistry

public «singleton» Class

Extends: Registry. : Responsible for generating game worlds while being provided a map name.

Registry::WorldMapRegistry Attributes

Attribute	Type	Notes
m_singleton	private static : *WorldMapRegistry*	The singleton instance. Initial Value: 0;
m_maps	private : *std::vector<std::string>*	Contains the names of all available maps.

Registry::WorldMapRegistry Methods

Method	Type	Notes
instance ()	«Helper» public static: *WorldMapRegistry&*	Returns the singleton instance of the world map registry.
getMaps ()	«Access» public: *std::vector<std:*	Returns an iterator of strings containing the names of all available maps.

	:string>::iterato r	
generateFromMap (*std::string*)	«Manager» public: *World*	param: mapName [std::string - in] Generates a game world instance from a given map name. Pre-condition: Functional Map exists Post-condition: Functional World created according to specified map
WorldMapRegistry ()	private:	Constructs a world map registry instance.

Util

Contains common utilities, such as coordinate representations etc.

CircularShape

public Class

Extends: Shape. : A circular shape.

CircularShape Attributes

Attribute	Type	Notes
m_radius	private : *float*	The radius of the circular shape.

CircularShape Methods

Method	Type	Notes
CircularShape (*float*)	public:	param: radius [float - in] Constructs a circular shape, defining its radius.
getRadius ()	«Access» public: *float*	Returns the radius of the circular shape.

Coord2d

public Class: Represents a coordinate in the absolute 2D-space.

Coord2d Attributes

Attribute	Type	Notes
m_x	private : *float*	The x coordinate.
m_y	private : *float*	The y coordinate.

Coord2d Methods

Method	Type	Notes
Coord2d (*float, float*)	public:	param: x [float - in] param: y [float - in] Constructs a new coordinate in 2D-space.
getX ()	«Access» public: *float*	Returns the X coordinate.
getY ()	«Access» public: *float*	Returns the Y coordinate.

Shape

public abstract Class: An abstrac geometric shape.

Vector2d

public Class: Defines a vector in 2D space by combining a coordinate and a length.

Vector2d Attributes

Attribute	Type	Notes
m_magnitude	private : *float*	The magnitude/length of the vector.
m_angle	private : *float*	The angle of the vector.

Vector2d Methods

Method	Type	Notes
Vector2d (*float, float*)	public:	param: magnitude [float - in] param: angle [float - in] Constructs a 2D vector of a certain magnitude and a certain angle.
getMagnitude ()	«Access» public: *float*	Returns the magnitude of the vector. i.e., its "length".
getAngle ()	«Access» public: *float*	Returns the angle of the vector.

Tickable

public abstract «interface» Interface: When a class object need to receive a tick, it has to implement this interface.

Tickable Interfaces

Method	Type	Notes
tick ()	«Event» public abstract:	Notifies the object about that the time is being incremented with one time unit.

	void	Post-condition: Functional Time-Dependent State Updated - The time-dependent state of the object is updated.

View

Responsible for drawing graphics for the game. Does so by monitoring the Game module and associating elements in it with own graphical objects, which then will be painted.

View::OpenGLRenderer

public Class

Implements: Renderer, Tickable. : Responsible for rendering graphics by using OpenGL.

View::OpenGLRenderer Methods

Method	Type	Notes
tick ()	«Event» public abstract: *void*	Ticks all contained *and current* sprite managers and triggers graphics rendering for the current game state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen Post-condition: Functional Sprite managers have received a tick
init ()	«Manager» public abstract: *void*	Initializes OpenGL. Pre-condition: Functional Game singleton initialized
render ()	«Manager» public abstract: *void*	Renders graphics for the current game state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen

renderState (*ControlsMenuState&*)	«Manager» private abstract: *void*	param: state [ControlsMenuState& - in] Renders graphics for the controls menu state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Current controls are displayed on screen Post-condition: Functional Graphics for current state displayed on screen
renderState (*MenuState&*)	«Manager» private abstract: *void*	param: state [MenuState& - in] Renders graphics for a generic menu state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen
renderState (*ControlsConfigMenuState&*)	«Manager» private abstract: *void*	param: state [ControlsConfigMenuState& - in] Renders graphics for the controls config menu state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Configuration instructions are displayed on screen Post-condition: Functional Graphics for current state displayed on screen
renderState (*HighScoreMenuState&*)	«Manager» private abstract: *void*	param: state [HighScoreMenuState& - in] Renders graphics for the high score menu state.

		Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen Post-condition: Functional High scores displayed on screen
renderState (*HelpMenuState&*)	«Manager» private abstract: *void*	param: state [HelpMenuState& - in] Renders graphics for the help menu state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen Post-condition: Functional Help text is displayed on screen
renderState (*SinglePlayerPlayState &*)	«Manager» private abstract: *void*	param: state [SinglePlayerPlayState& - in] Renders graphics for the single player play state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen
renderState (*TwoPlayerPlayState&*)	«Manager» private abstract: *void*	param: state [TwoPlayerPlayState& - in] Renders graphics for the two players play state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen

renderSpriteManager (*TextSpriteManager&*)	«Manager» private abstract: *void*	param: manager [TextSpriteManager& - in] Renders graphics for a single text sprite manager. Pre-condition: Functional Coordinate system moved to the desired position Pre-condition: Functional Modelview matrix pushed

View::Renderer

public abstract «interface» Interface: Interface for a graphics renderer.

View::Renderer Interfaces

Method	Type	Notes
render ()	«Manager» public abstract: *void*	Renders graphics for the current game state. Pre-condition: Functional A valid game state is entered in Game Post-condition: Functional Graphics for current state displayed on screen
init ()	«Manager» public abstract: *void*	Initializes the renderer. Pre-condition: Functional Game singleton initialized

Sprite

Sprite::AnimationSprite

public Class

Extends: Sprite. Implements: Tickable. :

Sprite::AnimationSprite Attributes

Attribute	Type	Notes
ANIM_SHIP_DESTROY	public const static : *unsigned short*	Ship destruction animation. Initial Value: 1;
ANIM_ASTEROID_DESTROY	public const static : *unsigned short*	Asteroid destruction animation. Initial Value: 2;
ANIM_SHIP_THROTTLE	public const static : *unsigned short*	Animates a throttling ship. Initial Value: 5;
m_animMap	private static : *std::map<unsigned short, std::vector<unsigned short> >*	Maps animation aliases to a vector of image aliases (see ANIM_* and IMG_* constants).
m_frames	private : *std::vector<unsigned short>*	Pointer to a vector of image frames. The vector is one of those residing in the m_animMap.
m_tickInterval	private : *unsigned short*	The number of ticks to wait between two image frames.
m_tickCountdown	private : *unsigned short*	Ticks left to next frame swap.

m_animsTotal	private : *unsigned short*	The total number of times that the animation is requested to be played. If 0, the animation will continue forever (i.e., until its host removes it).
m_animsLeft	private : *unsigned short*	The remaining number of times that the animation can be played. Initial Value: 0;

Sprite::AnimationSprite Methods

Method	Type	Notes
AnimationSprite (*unsigned short, unsigned short*)	public:	param: animation [unsigned short - in] The animation to represent. param: frameInterval [unsigned short - in] Ticks between two frames. Constructs a always-repeating animation sprite, specifying the animation it should represent and the desired time interval (ticks) between two frames. Action: Calls parent constructor with the first frame in the animation.
AnimationSprite (*unsigned short, unsigned short, unsigned short*)	public:	param: animation [unsigned short - in] The animation to represent. param: frameInterval [unsigned short - in] Ticks between two frames. param: repeat [unsigned short - in] Number of times to repeat the animation. Constructs an animation sprite, specifying the animation it should represent, the desired time interval (ticks) between two frames, and the number

		of times it should be repeated (0 means played once). Action: Calls parent constructor with the first frame in the animation.
isFinished ()	«Access» public: *bool*	Tells whether the animation has finished playing as well as repeating or not. If the animation is set to always repeat, then this function will always return false.
tick ()	«Event» public abstract: *void*	Counts down to next frame change. Post-condition: Functional Countdown decremented - m_tickCountdown should be decremented with 1. Action: Decrements m_tickCountdown and if it reaches 0, advances to next image frame, changing m_image, or, if repeat is requested and the last element was reached, falls back to the first frame.

Sprite::Sprite

public Class:

Sprite::Sprite Attributes

Attribute	Type	Notes
IMG_SHIP	public const static : *unsigned short*	Ship image ID. Initial Value: 1;
IMG_SHIP_DESTROY	public const static :	Image ID of the first frame in a ship's destruction animation.

_F1	*unsigned short*	Initial Value: 2;
IMG_SHIP_DESTROY_F2	public const static : *unsigned short*	Image ID of the second frame in a ship's destruction animation. Initial Value: 3;
IMG_SHIP_DESTROY_F3	public const static : *unsigned short*	Image ID of the third frame in a ship's destruction animation. Initial Value: 4;
IMG_ASTEROID_DESTROY_F1	public const static : *unsigned short*	Image ID of the first frame in an asteroid's destruction animation. Initial Value: 5;
IMG_ASTEROID_DESTROY_F2	public const static : *unsigned short*	Image ID of the second frame in an asteroid's destruction animation. Initial Value: 6;
IMG_ASTEROID_DESTROY_F3	public const static : *unsigned short*	Image ID of the third frame in an asteroid's destruction animation. Initial Value: 7;
IMG_SHIP_THROTTLE_F1	public const static : *unsigned short*	Image ID of the first frame in a ship's throttling animation. Initial Value: 8;
IMG_SHIP_THROTTLE_F2	public const static : *unsigned short*	Image ID of the second frame in a ship's throttling animation. Initial Value: 9;
IMG_ASTEROID	public const static : *unsigned short*	Asteroid image ID. Initial Value: 10;
IMG_SHIP_THROTTLE_F3	public const static : *unsigned short*	Image ID of the third frame in a ship's throttling animation. Initial Value: 11;

IMG_FUEL_ITEM	public const static : *unsigned short*	Fuel item image ID. Initial Value: 12;
IMG_MISSILE_ITEM	public const static : *unsigned short*	Missile item image ID. Initial Value: 13;
IMG_MISSILE	public const static : *unsigned short*	Missile image ID. Initial Value: 14;
IMG_LASER	public const static : *unsigned short*	Laser image ID. Initial Value: 15;
IMG_SPAWN_POINT	public const static : *unsigned short*	Spawn point image ID. Initial Value: 16;
IMG_BACKGROUND_1	public const static : *unsigned short*	World background 1 tile image ID. Initial Value: 17;
IMG_PLANET_1	public const static : *unsigned short*	Planet 1 image ID. Initial Value: 18;
IMG_PLANET_2	public const static : *unsigned short*	Planet 2 image ID. Initial Value: 19;
IMG_PLANET_3	public const static : *unsigned short*	Planet 3 image ID. Initial Value: 20;
IMG_CHAR_UC_A	public const static : *unsigned short*	Image ID of the capital letter A. Initial Value: 21;
IMG_CHAR_UC_B	public const static : *unsigned short*	Image ID of the capital letter B. Initial Value: 22;

IMG_CHAR_UC_C	public const static : *unsigned short*	Image ID of the capital letter C. Initial Value: 23;
IMG_CHAR_UC_D	public const static : *unsigned short*	Image ID of the capital letter D. Initial Value: 24;
IMG_CHAR_UC_E	public const static : *unsigned short*	Image ID of the capital letter E. Initial Value: 25;
IMG_CHAR_UC_F	public const static : *unsigned short*	Image ID of the capital letter F. Initial Value: 26;
IMG_CHAR_UC_G	public const static : *unsigned short*	Image ID of the capital letter G. Initial Value: 27;
IMG_CHAR_UC_H	public const static : *unsigned short*	Image ID of the capital letter H. Initial Value: 28;
IMG_CHAR_UC_I	public const static : *unsigned short*	Image ID of the capital letter I. Initial Value: 29;
IMG_CHAR_UC_J	public const static : *unsigned short*	Image ID of the capital letter J. Initial Value: 30;
IMG_CHAR_UC_K	public const static : *unsigned short*	Image ID of the capital letter K. Initial Value: 31;
IMG_CHAR_UC_L	public const static : *unsigned short*	Image ID of the capital letter L. Initial Value: 32;
IMG_CHAR_UC_M	public const static : *unsigned short*	Image ID of the capital letter M. Initial Value: 33;

IMG_CHAR_UC_N	public const static : *unsigned short*	Image ID of the capital letter N. Initial Value: 34;
IMG_CHAR_UC_O	public const static : *unsigned short*	Image ID of the capital letter O. Initial Value: 35;
IMG_CHAR_UC_P	public const static : *unsigned short*	Image ID of the capital letter P. Initial Value: 36;
IMG_CHAR_UC_Q	public const static : *unsigned short*	Image ID of the capital letter Q. Initial Value: 37;
IMG_CHAR_UC_R	public const static : *unsigned short*	Image ID of the capital letter R. Initial Value: 38;
IMG_CHAR_UC_S	public const static : *unsigned short*	Image ID of the capital letter S. Initial Value: 39;
IMG_CHAR_UC_T	public const static : *unsigned short*	Image ID of the capital letter T. Initial Value: 40;
IMG_CHAR_UC_U	public const static : *unsigned short*	Image ID of the capital letter U. Initial Value: 41;
IMG_CHAR_UC_V	public const static : *unsigned short*	Image ID of the capital letter V. Initial Value: 42;
IMG_CHAR_UC_W	public const static : *unsigned short*	Image ID of the capital letter W. Initial Value: 43;
IMG_CHAR_UC_X	public const static : *unsigned short*	Image ID of the capital letter X. Initial Value: 44;

IMG_CHAR_UC_Y	public const static : *unsigned short*	Image ID of the capital letter Y. Initial Value: 45;
IMG_CHAR_UC_Z	public const static : *unsigned short*	Image ID of the capital letter Z. Initial Value: 46;
IMG_CHAR_SPACE	public const static : *unsigned short*	Image ID of the space character. Initial Value: 47;
IMG_CHAR_COMMA	public const static : *unsigned short*	Image ID of the comma character. Initial Value: 48;
IMG_CHAR_COLON	public const static : *unsigned short*	Image ID of the colon character. Initial Value: 481;
IMG_CHAR_POINT	public const static : *unsigned short*	Image ID of the point character. Initial Value: 49;
IMG_CHAR_QUESTION	public const static : *unsigned short*	Image ID of the question mark character. Initial Value: 50;
IMG_CHAR_EXCLAMATION	public const static : *unsigned short*	Image ID of the exclamation mark character. Initial Value: 51;
IMG_CHAR_0	public const static : *unsigned short*	Image ID of the 0 character. Initial Value: 52;
IMG_CHAR_1	public const static : *unsigned short*	Image ID of the 1 character. Initial Value: 53;
IMG_CHAR_2	public const static : *unsigned short*	Image ID of the 2 character. Initial Value: 54;

IMG_CHAR_3	public const static : *unsigned short*	Image ID of the 3 character. Initial Value: 55;
IMG_CHAR_4	public const static : *unsigned short*	Image ID of the 4 character. Initial Value: 56;
IMG_CHAR_5	public const static : *unsigned short*	Image ID of the 5 character. Initial Value: 57;
IMG_CHAR_6	public const static : *unsigned short*	Image ID of the 6 character. Initial Value: 58;
IMG_CHAR_7	public const static : *unsigned short*	Image ID of the 7 character. Initial Value: 59;
IMG_CHAR_8	public const static : *unsigned short*	Image ID of the 8 character. Initial Value: 60;
IMG_CHAR_9	public const static : *unsigned short*	Image ID of the 9 character. Initial Value: 61;
IMG_HUD_SHIP_ENERGY_0	public const static : *unsigned short*	Image ID of the ship's energy percentage bar in the heads-up display (0%). Initial Value: 62;
IMG_HUD_SHIP_ENERGY_20	public const static : *unsigned short*	Image ID of the ship's energy percentage bar in the heads-up display (20%). Initial Value: 63;
IMG_HUD_SHIP_ENERGY_40	public const static : *unsigned short*	Image ID of the ship's energy percentage bar in the heads-up display (40%). Initial Value: 64;
IMG_HUD_SHIP_ENE	public const static :	Image ID of the ship's energy percentage bar in the

RGY_60	*unsigned short*	heads-up display (60%). Initial Value: 65;
IMG_HUD_SHIP_ENERGY_80	public const static : *unsigned short*	Image ID of the ship's energy percentage bar in the heads-up display (80%). Initial Value: 66;
IMG_HUD_SHIP_ENERGY_100	public const static : *unsigned short*	Image ID of the ship's energy percentage bar in the heads-up display (100%). Initial Value: 67;
IMG_HUD_LIVES_LEFT	public const static : *unsigned short*	Image ID of the "lives left" container in the heads-up display. Initial Value: 68;
IMG_HUD_SHIP_FUEL_0	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (0%). Initial Value: 69;
IMG_HUD_SHIP_FUEL_10	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (10%). Initial Value: 70;
IMG_HUD_SHIP_FUEL_20	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (20%). Initial Value: 71;
IMG_HUD_SHIP_FUEL_30	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (30%). Initial Value: 72;
IMG_HUD_SHIP_FUEL_40	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (40%). Initial Value: 73;
IMG_HUD_SHIP_FUE	public const static :	Image ID of the ship's fuel percentage bar in the heads-up display (50%).

L_50	*unsigned short*	Initial Value: 74;
IMG_HUD_SHIP_FUEL_60	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (60%). Initial Value: 75;
IMG_HUD_SHIP_FUEL_70	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (70%). Initial Value: 76;
IMG_HUD_SHIP_FUEL_80	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (80%). Initial Value: 77;
IMG_HUD_SHIP_FUEL_90	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (90%). Initial Value: 78;
IMG_HUD_SHIP_FUEL_100	public const static : *unsigned short*	Image ID of the ship's fuel percentage bar in the heads-up display (100%). Initial Value: 79;
IMG_MENU_SELECTED_PREFIX	public const static : *unsigned short*	Selected menu item prefix image ID. Initial Value: 80;
m_image	protected : *unsigned short*	The sprite alias (see IMG_* constants).

Sprite::Sprite Methods

Method	**Type**	**Notes**
Sprite (*unsigned short*)	public:	param: image [unsigned short - in] Constructs a sprite, specifying the image it should represent.

		Pre-condition: Functional Image is a constant defined in Sprite
getImage ()	«Access» public const: *unsigned short*	Returns the sprite's image constant value.

SpriteManager

SpriteManager::AsteroidSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for an asteroid.

SpriteManager::FuelItemSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a fuel item.

SpriteManager::LaserSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a laser.

SpriteManager::MenuButtonSpriteManager

public Class

Extends: TextSpriteManager. :

SpriteManager::MenuButtonSpriteManager Methods

Method	Type	Notes
MenuButtonSpriteManager (*std::string&*)	public:	param: name [std::string& - in] Constructs a menu button sprite manager, being provided its name. Action: Calls parent constructor with SIZE_LARGE, for example.

SpriteManager::MissileItemSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a missile item.

SpriteManager::MissileSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a missile.

SpriteManager::PlanetSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a planet.

SpriteManager::PlanetSpriteManager Methods

Method	Type	Notes
PlanetSpriteManager (*unsigned short*)	public:	param: planetType [unsigned short - in] The planet type according to what Planet::getType() returns. Constructs a sprite manager for a planet, specifying its type and therefore its sprite.

SpriteManager::ShipSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a ship.

SpriteManager::SpawnPointSpriteManager

public Class

Extends: SpriteManager. : Responsible for managing sprites for a spawn point.

SpriteManager::SpriteManager

public abstract Class

Implements: GameEventListener, Tickable. : Abstract sprite manager.

SpriteManager::SpriteManager Methods

Method	Type	Notes
getSpriteIterator ()	«Access» public const: *map<Sprite*, Coord2d>::iterator*	Returns an iterator of active sprites (those that are meant to be displayed) mapped into their offset coordinate to the previous sprite. Example: if we have a text, then the first sprite will have coordinate (0,0), while the other will have, say, (20,0), and so on.
tick ()	«Event» public abstract: *void*	Ticks the contained animation sprites. Post-condition: Functional Contained animation sprites received a tick Action: Ticks active animation sprites.
notifyEvent (*GameEvent&*)	«Event» public abstract: *void*	param: ev [GameEvent& - in] Notifies the listener about an occuring game event.
addActiveSprite (*Sprite&, Coord2d&*)	«Manager» protected: *void*	param: sprite [Sprite& - in] param: offsetCoord [Coord2d& - in] Adds an active sprite, specifying its offset coordinate to the previously added sprite.

		Post-condition: Functional If tickable then stored in tickable sprites Action: Adds a sprite to m_avtiveSprites and, if tickable, to m_tickableActiveSprites.
clearActiveSprites ()	«Manager» protected: *void*	Clears all currently active sprites. Post-condition: Functional All active sprites are cleared - Both the active and tickable active sprites are cleared.

SpriteManager::TextSpriteManager

public abstract Class

Extends: SpriteManager. : Abstract sprite manager that will manage texts.

SpriteManager::TextSpriteManager Attributes

Attribute	Type	Notes
SIZE_SMALL	public const static : *unsigned short*	Small text size. Initial Value: 1;
SIZE_LARGE	public const static : *unsigned short*	Large text size. Initial Value: 2;
m_size	private : *unsigned short*	The size that the contained text should have.
m_textLines	private : *std::vector<std::vector<Sprite**	Vector of text lines, each of which is a vector consisting of sprite pointers.

	> >	
m_charToSpriteMap	private const static : *std::map<char, unsigned short>*	Maps single characters to the IMG_CHAR_* constants in Sprite.

SpriteManager::TextSpriteManager Methods

Method	Type	Notes
TextSpriteManager (*std::string&, unsigned short*)	public:	param: text [std::string& - in] param: size [unsigned short - in] See SIZE_* constants. Constructs a text sprite specifying the text to represent as well as the size it should have. Action: Translates string into line-by-line sprite vectors. See m_textLines. Uses m_charToSpriteMap for mapping.
getSize ()	«Access» public: *unsigned short*	Returns the desired size of the text. See SIZE_* constants.

5.6 Package Diagram

Diagram: Controller - Package Diagram

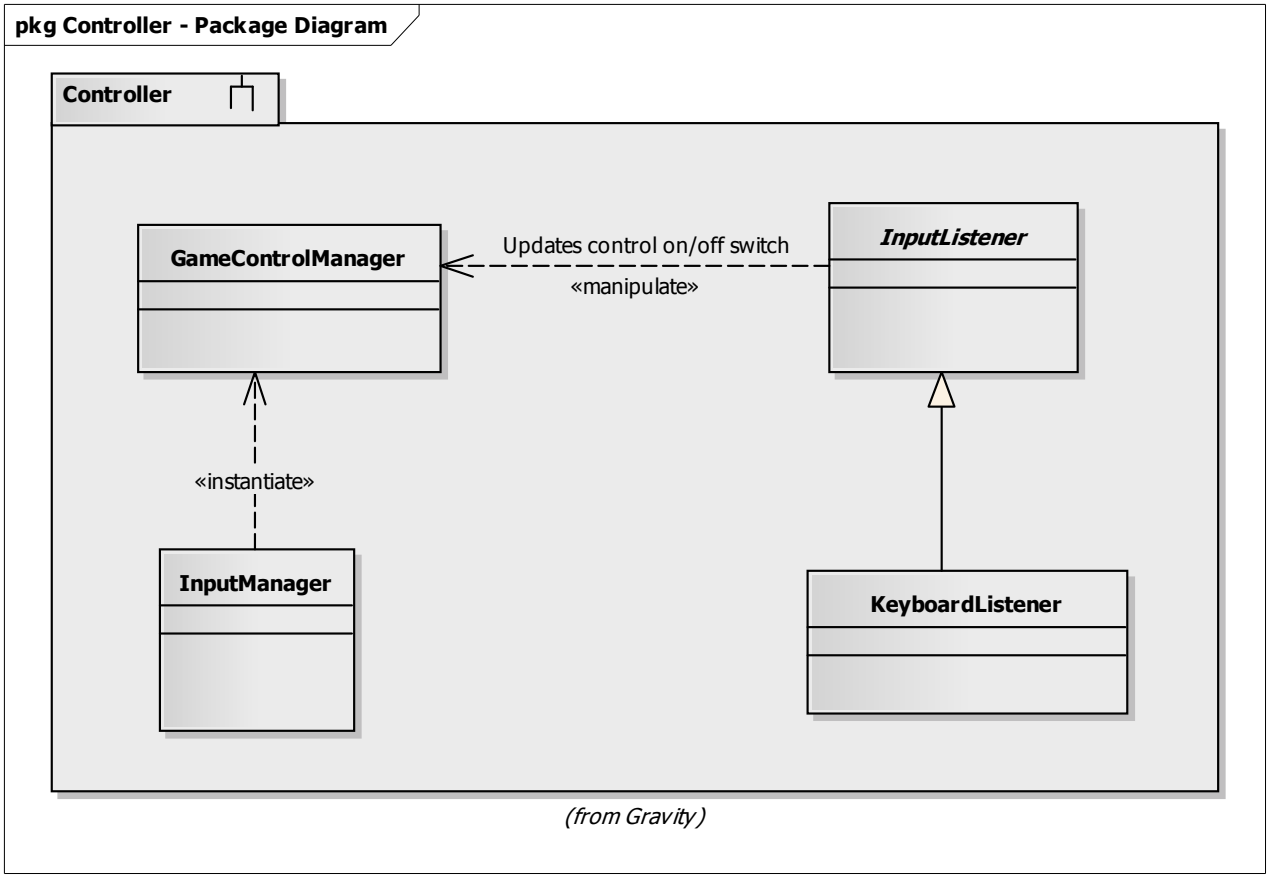


Diagram: Game - Package Diagram

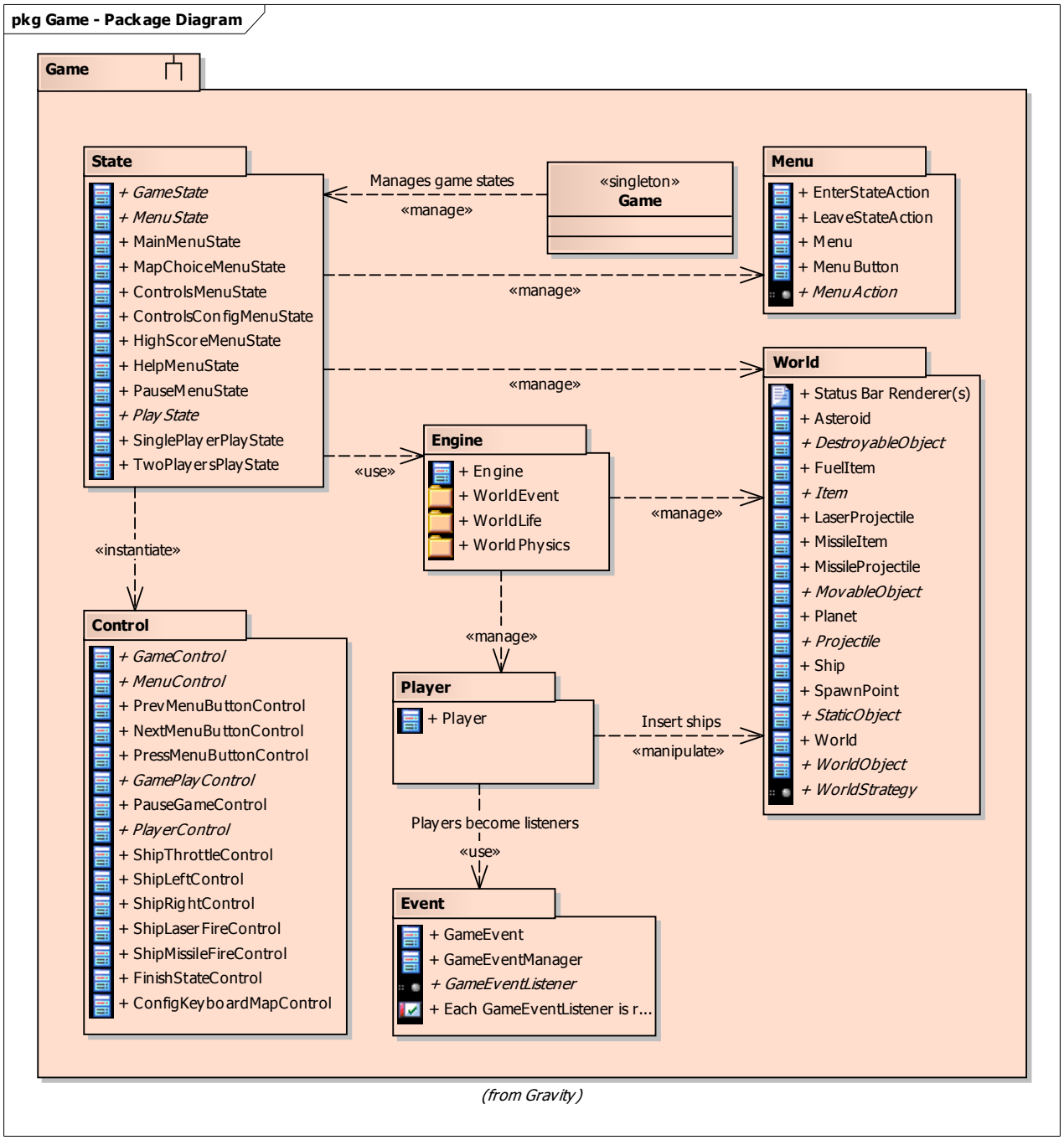


Diagram: Game Engine - Package Diagram

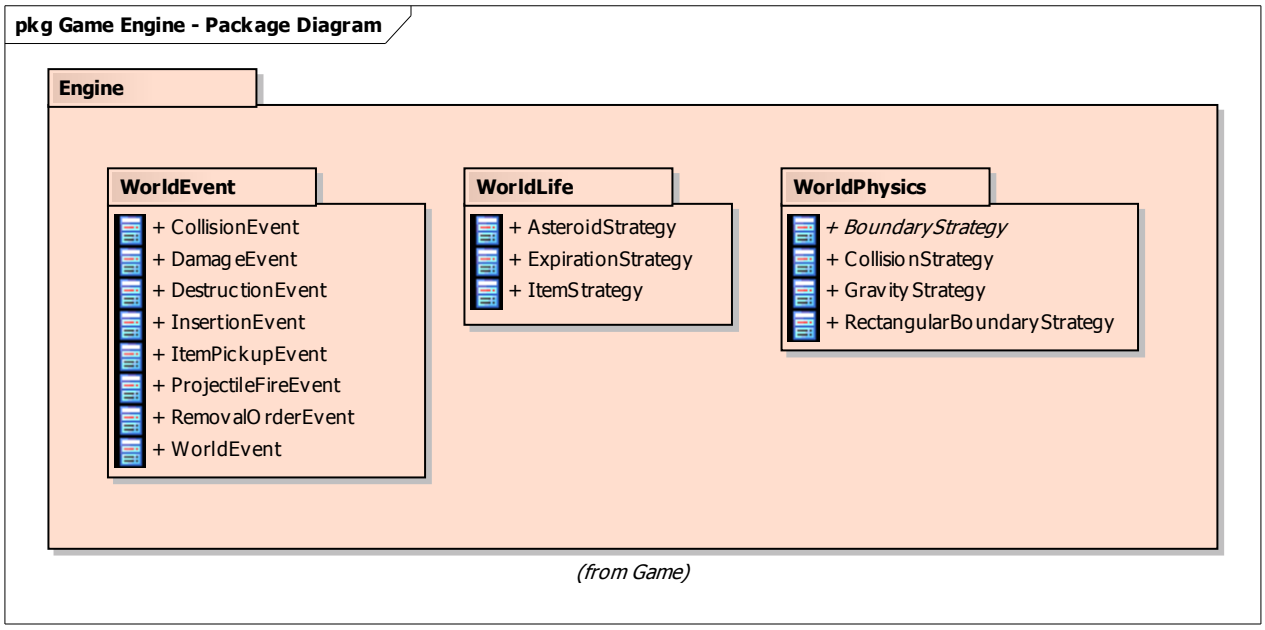


Diagram: Audio - Package Diagram

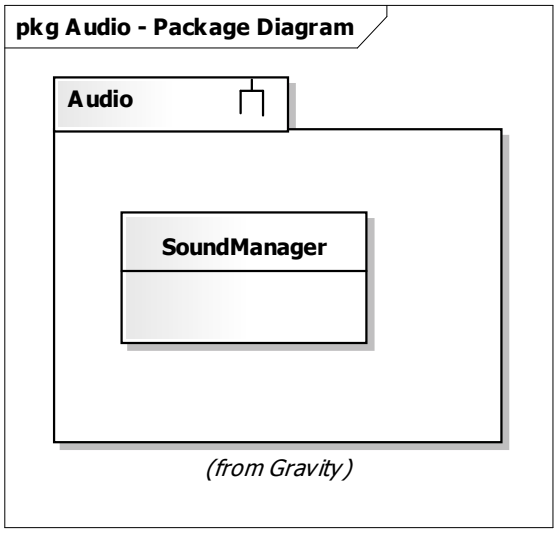


Diagram: View - Package Diagram

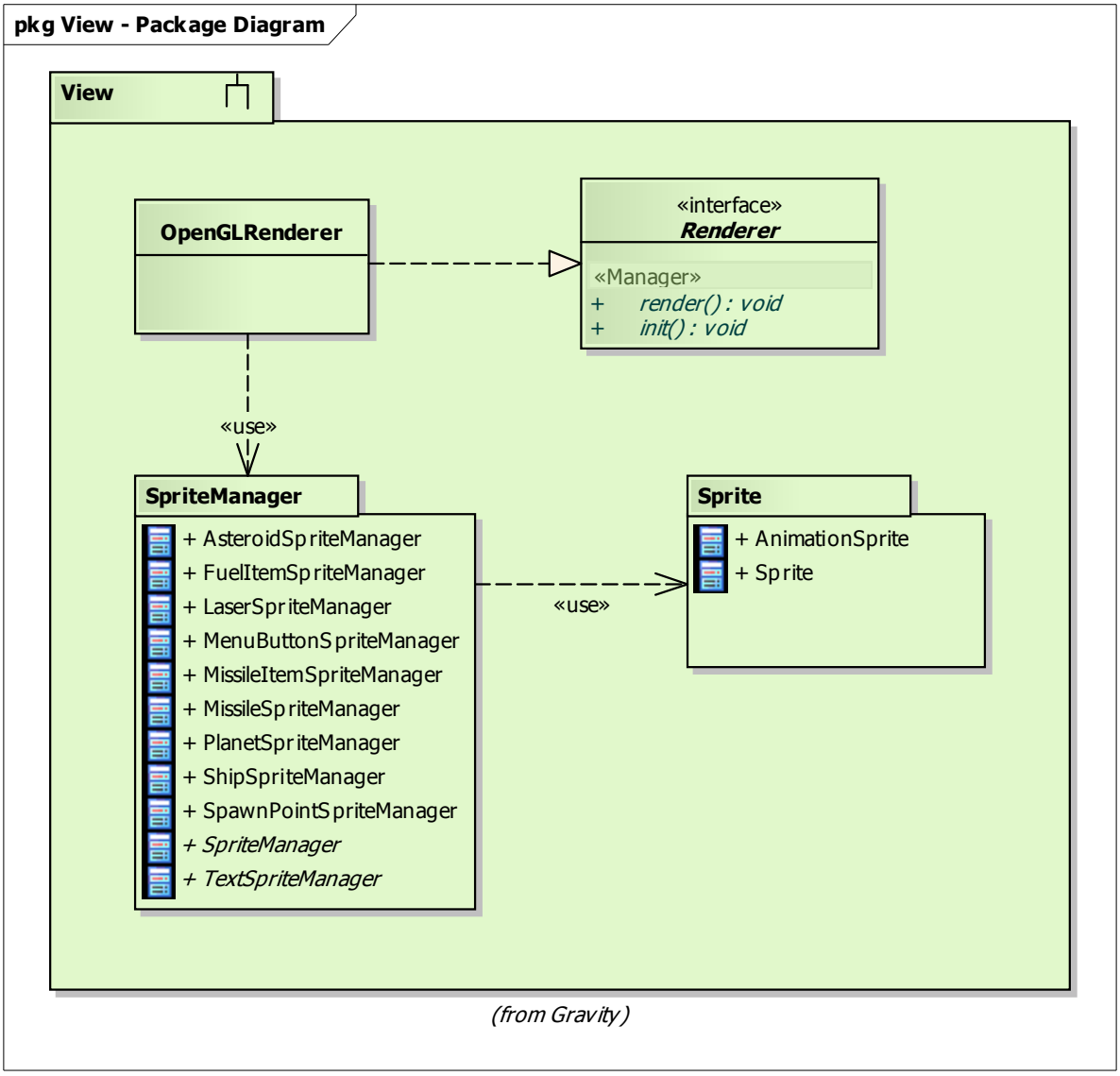


Diagram: Registry - Package Diagram

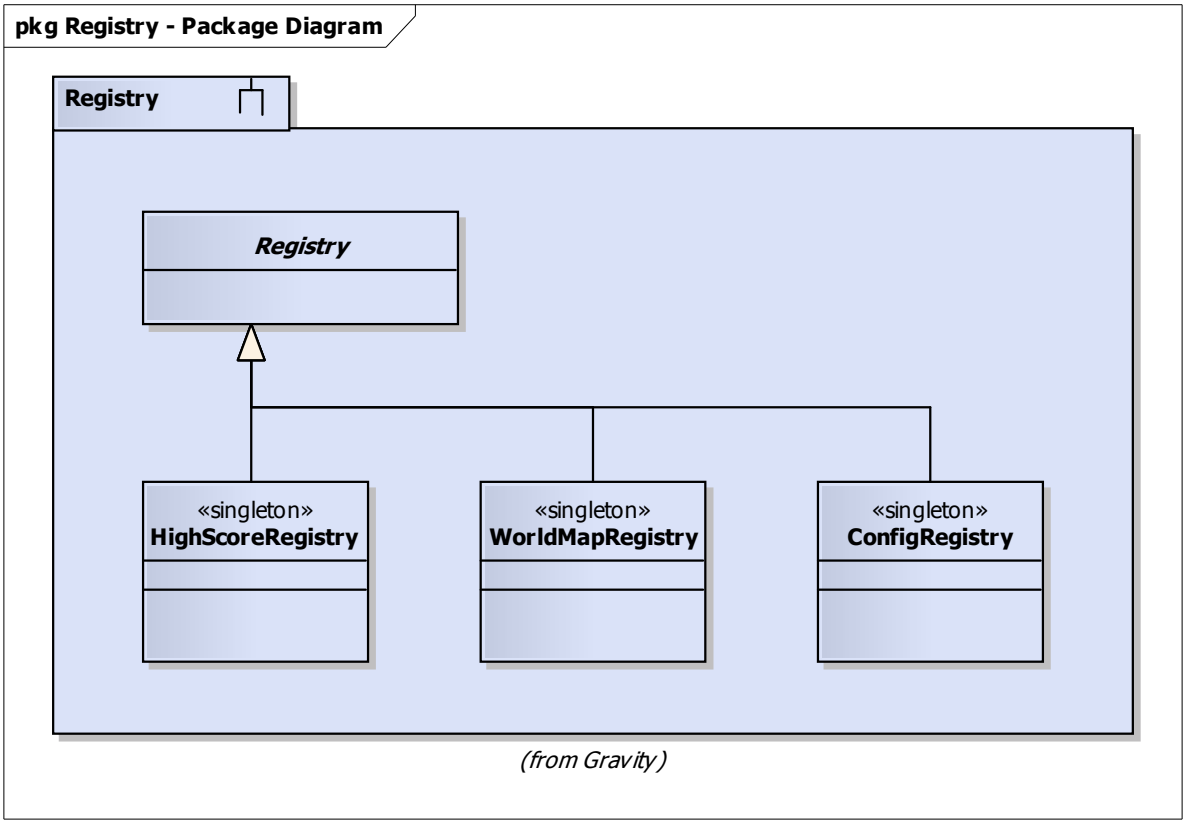
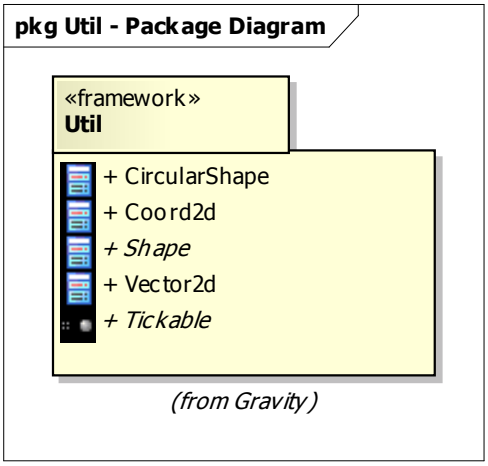


Diagram: Util - Package Diagram



6. Functional Test Cases

6.1 Quick Start Help

Description of functionality being tested

A user who doesn't know what the game is about and/or what the controls are, shall

be able to get a short summary on these points before starting a game session. The

summary shall contain the goals of the game play, together with the currently set controls.

Reference to requirement document

Section 4.1.1.

Expected behavior

A help screen is shown with information about how to play the game and the controls of the game are explained.

Steps to reproduce test

1. Start the game.
2. Select the quick start help in the main menu.
3. Read the quick start help and verify content.

6.2 Map Choice

Description of functionality being tested

Before starting a game session, the user shall be able to choose the map that that session should be played on. If no active map choice is made, the game system shall choose one of the available maps. Each map shall be a definition of the world/environment the player finds herself in while playing a game session.

Reference to requirement document

Section 4.1.2.

Expected behavior

When a game is started after choosing a new map the new map is shown.

Steps to reproduce test

1. Start the game.

2. Select the “start the single player game” menu option.
3. Choose a map.

6.3 Controls Configuration

Description of functionality being tested

Before starting a game session, the user(s) shall be able (but not required) to configure the game controls, i.e., what keyboard keys to use for what action in the game. The game shall provide default controls, allowing the user(s) to change them at any time.

Reference to requirement document

Section 4.1.3.

Expected behavior

If you can choose keys for the controls and after that the chosen keys is used to control the ship(s) in the game.

Steps to reproduce test

1. Start the game.
2. Choose “setup controls” in the menu.
3. Setup keys to control the ships.
4. Return to main menu.
5. Choose “start multiplayer game”.
6. Choose “start game” in the multiplayer game rule choice menu.
7. Test that you can control the ship with the keys chosen in step 3.

6.4 Single Player or Two Players Choice

Description of functionality being tested

Before starting a game session, the user shall be requested to choose whether she wants to play in single player mode or multiplayer mode.

Reference to requirement document

Section 4.1.4.

Expected behavior

When choosing single player mode you get to play by yourself. When selecting multiplayer mode you get to play against one human opponent both using the keyboard to control their ships.

Steps to reproduce test

1. Start the game.
2. Select “start single player game”.
3. See if you're playing in single player mode.
4. End the game.
5. From the main menu select “start multiplayer game”.

6.5 Two Players Game Rule Choice

Description of functionality being tested

Before starting a game session in two player mode, the users shall be able to set a

points limit, defining when the game is going to end.

Reference to requirement document

Section 4.1.5.

Expected behavior

You should be able to choose the number of lives before starting a two player game. The game shall end when the set point limit is reached by any of the two players.

Steps to reproduce test

1. Start the game.
2. Choose “Start multiplayer game”.
3. Choose number of lives.
4. Play the game until out of lives.

6.6 Single Player High Score List

Description of functionality being tested

Exiting the game, either as a result of losing all ships (game over) or by exiting the

game deliberately, shall let the player know about the current high score list and, if

she had qualified for a placement on it, request her name. Only exit options provided by the game shall follow this requirement.

Reference to requirement document

Section 4.1.6.

Expected behavior

If the player has reached a number of points enough for the high score list, the player should be prompted for her name. Also the high score list should be shown after a game ends.

Steps to reproduce test

1. Start the game.
2. Choose “Start single player game”.
3. Achieve more points than the last entry in the high score list.
4. Lose all ships.
5. Enter name into high score list.

6.7 Exiting The Game

Description of functionality being tested

It shall be possible to exit the game at any stage. While not playing the game, the user shall be able to quit to the operating system. The user shall be prompted if she is sure she wants to quit, when any exit function is chosen. While playing, the user shall be able to choose whether to exit to setup or exit to the operating system.

Reference to requirement document

Section 4.1.7.

Expected behavior

If it's possible to exit to setup and to operating system while playing the game and also to exit to operating system while in setup this requirement is met.

Steps to reproduce test

1. Start the game.
2. Test exit button while in main menu and also when a game is started.

6.8 World Boundary Wrapping

Description of functionality being tested

When a player makes her ship to go beyond one of the world boundaries, it shall be "teleported" to the opposite side of the world. For example, going out on the left side shall result in appearing on the right side.

Reference to requirement document

Section 4.1.8.

Expected behavior

When a ship is at the border of the game world it is teleported to the opposite vertical and horizontal border, maintaining its movement.

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Move the ship to a border of the game world.

6.9 Player's World View

Description of functionality being tested

The player shall see her ship from above, at a distance that depends on the ship's movement speed. When the speed is increasing, the viewing distance shall increase too. Conversely, when the speed is decreasing, the viewing distance shall decrease too.

Reference to requirement document

Section 4.1.9.

Expected behavior

The the zoom-level is proportional to the speed of the ship (more zoomed out the faster the ship moves).

Steps to reproduce test

1. Start the game.
2. Choose “Start single player game”.
3. Accelerate the ship.

6.10 Game Play Information

Description of functionality being tested

The player shall be able to see information about her ship's health and fuel amount during the game play. If the game rules say that there should not be any fuel restrictions (i.e., unlimited fuel supply), then information about the fuel amount should be left out.

Reference to requirement document

Section 4.1.10.

Expected behavior

The ship information is visible while playing the game.

Steps to reproduce test

1. Start the game.

2. Choose “Start single player game”.

6.11 Scoring in Single Player Mode

Description of functionality being tested

Scores shall be gained partly by shooting at asteroids and partly by the time the player managed to stay alive, having three ships (and therefore chances) at her disposal.

Reference to requirement document

Section 4.1.11.

Expected behavior

The points are gained slowly while traveling around, and faster when shooting an asteroid.

Steps to reproduce test

1. Start the game.
2. Choose “Start single player game”.
3. Shoot down an asteroid.

6.12 Scoring in Two Players Mode

Description of functionality being tested

A player shall get rewarded when destroying his opponent's ship with any weapon. A player shall be punished when his ship is destroyed by crashing into some obstacle in the world (including the opponent's ship).

Reference to requirement document

Section 4.1.12.

Expected behavior

The player get points when destroying an opponent, and gets lower score when being killed.

Steps to reproduce test

1. Start the game.
2. Choose “Start Multi player game”.
3. Shoot down the opponent.

6.13 Single Player Ship Disposal (Lives)

Description of functionality being tested

The player shall start having three ships at her disposal. Gaining a certain amount of points shall give another ship. The ships shall not be used simultaneously, but once one is crashed it shall be replaced with a new one if available, otherwise the game ends and the achieved points shall be displayed together with a high score list.

Reference to requirement document

Section 4.1.13.

Expected behavior

If the game information displays three ships when the game starts, and if one ship is removed when the players ship is destroyed, and if one ship is added when a preset amount of points are collected this requirement is met.

Steps to reproduce test

1. Start the game.
2. Choose “Start single player game”.
3. Run the ship into a planet.

6.14 Ship Speed Restriction

Description of functionality being tested

A ship's movement shall be restricted in speed. When the speed reaches a set limit, throttling shall not be able to increase it.

Reference to requirement document

Section 4.1.14.

Expected behavior

The player accelerates and reaches this speed no further thrust will accelerate the ship.

Steps to reproduce test

1. Start the game.
2. Choose “Start single player game”.
3. Accelerate the ship.

6.15 Ship Fuel Restriction

Description of functionality being tested

A ship shall either have a fuel restriction (when in single player mode) or have an infinite fuel supply (when in two players mode). When there is a fuel restriction, it shall also be a restriction on how much fuel the ship can have at once.

Reference to requirement document

Section 4.1.15.

Expected behavior

If the fuel amount reaches zero no further thrust will be possible. In two player mode no amount of thrust will decrease the amount of fuel available.

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Accelerate the ship until fuel meter is empty.

6.16 Ship Damage

Description of functionality being tested

A ship shall be completely damaged when colliding with other ships, a planet or an asteroid. A ship's damage resulting from a weapon projectile hit shall be defined by the destructive power of that projectile. Complete damage results in ship destruction.

Reference to requirement document

Section 4.1.16.

Expected behavior

The ship shall be destroyed after colliding with a ship, planet or asteroid and the ship shall also resist one shot without being completely damaged.

Steps to reproduce test

1. Start the game.
2. Choose “Start multiplayer game”.
3. Move the ship into another ship, a planet or an asteroid.

6.17 Operating a Ship

Description of functionality being tested

A ship shall be controllable by throttling (i.e., gaining speed in the direction of the ship) and steering right and left respectively. Once a ship's movement and speed is achieved it shall remain constant until its destruction, unless affected by a gravity or its throttling.

Reference to requirement document

Section 4.1.17.

Expected behavior

The ship shall be maneuverable.

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Use the configured controls to maneuver.

6.18 Ship Laser Gun

Description of functionality being tested

A ship shall be able to fire laser projectiles. The laser projectiles shall not be affected by planetary gravities. The damage caused by a laser shall be partial. Lasers shall travel fast (in relation to missiles).

Reference to requirement document

Section 4.1.18.

Expected behavior

The ship shall fire a laser.

Steps to reproduce test

1. Start the game.

2. Choose “start single player game”.
3. Press the control associated with firing laser weapon.

6.19 Ship Missile Launcher

Description of functionality being tested

A ship shall be able to fire missile projectiles. The missile projectiles shall be affected by planetary gravities. The damage caused by a missile shall be complete. Missiles shall be slower than a laser projectile, but faster than a ship.

Reference to requirement document

Section 4.1.19.

Expected behavior

The ship shall fire a missile

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Press the control associated with firing missile weapon.

6.20 Operating a Ship's Weapons

Description of functionality being tested

The player shall be able to choose which weapon to use before firing it off. The projectile resulting from firing a ship's weapon shall be set off from the ship's current position and in the current direction of the ship (i.e., not ship movement, but where the ship will strive to go when/if throttling).

Reference to requirement document

Section 4.1.20.

Expected behavior

The player shall be able to choose a weapon before firing it. Upon firing the chosen weapon shall be ired.

Steps to reproduce test

1. Start the game.
2. Choose the “start singe player mode”.
3. Press the button associated with changing weapons.
4. Fire the weapon.

6.21 Planets

Description of functionality being tested

A planet shall have a gravity which shall affect ships, missiles and asteroids exclusively. A planet shall not move in any way. An object being affected by a planet's gravity shall be pulled towards that planet with a certain strength. Asteroids hitting a planet shall, if that is the rule of the game mode or the map, increase the planet's gravitation.

Reference to requirement document

Section 4.1.21.

Expected behavior

The planet shall attract the players, asteroids and missiles.

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Move the ship around the planets.
4. Fire missile.

6.22 Asteroids

Description of functionality being tested

In single player mode, asteroids shall be sent in to the world at an adequate frequency, making the game play challenging enough. In two players mode, asteroids may be sent in at a deliberate frequency. Asteroids shall be destructible, both partially and completely. Partial destruction means that an asteroid is split into two smaller asteroids, while complete destruction means that the whole asteroid is destroyed.

Reference to requirement document

Section 4.1.22.

Expected behavior

Asteroids shall appear in single player mode.

Steps to reproduce test

1. Start the game.
2. Choose the “start single player game”.
3. Move around the game world.

6.23 Items

Description of functionality being tested

In single player mode, items containing fuel shall occur randomly in both place and time in the world, and frequently enough to guarantee that the player has a fair chance to pick them up before running out of fuel. In two players mode, items shall not contain fuel, but instead they shall contain weaponry and ship health upgrades. Items shall appear in free space in the world, allowing a player to pick them up.

Reference to requirement document

Section 4.1.23.

Expected behavior

Items shall appear.

Steps to reproduce test

1. Start the game.
2. Choose “start single player game”.
3. Move around the world.

6.24 Sound Effects

Description of functionality being tested

Sound effects shall be played for each of the following events: collisions, fired weapons, ship throttle and item pickups.

Reference to requirement document

Section 4.1.24.

Expected behavior

When the event occurs the sound shall be heard.

Steps to reproduce test

1. Start the game.

2. Choose “start single player game”.
3. Fire a laser.