Bilkent University

CS 319

Object-Oriented Software Engineering

Fall 2014

Dungeon Escape

DESIGN REPORT

December 3rd, 2014

GROUP #10

Ahmet Emre Danışman

Ateş Balcı

Buket Depren

Batuhan Berk Yaşar

Ayşe İrem Güner

Table of Contents

1 Introduction

1.1 Purpose of the System

Dungeon Escape is a 2D side scrolling game with various puzzle elements. The player will try to advance through levels by solving built-in puzzles via usable tools (e.g. boomerangs, hooks and ropes), gates and levers. Interaction with map and how to use those tools will be explained by little tips during the game. Since difficult games get a lot attention nowadays, the game will be challenging enough to preserve player's interest and attention. The real aim of the game is to challenge player with its difficult puzzles.

1.2 Design Goals

Before starting the coding and designing parts of the project, design goals should be decided in order to create a game with optimal qualities. The most significant designing parameters can be seen below.

Portability: Considering performance and speed of the program, Java may not be the best programming language choice to code a game, but we want Dungeon Escape to be a cross-platform game that anyone can play without dealing with system requirement problems so that the game can reach a wide range of users. In this aspect, Java is superior compared to programming languages that promise better performance, like C++. We believe that it is a reasonable tradeoff because our game will not be so complex therefore it will not be needed to code it in the way that it will work fastest.

Usability: If developers want a game to be played, they should make it easy to play and fully understandable. Games that are easy to play should be user-friendly and attractive because players do not have to focus on learning the game, instead they can focus on mastering it. To accomplish usability criteria, players should not have trouble on selecting items, recognizing map etc. and to achieve that, Dungeon Master will have a short tutorial on the gameplay alongside pop-up hints and tips on anything player will come across. Also, main menu and pause menu will be mouse-controlled, friendly themed and plain enough for player to become fond of them as well as having background music. A repetitive and soft music will keep the player's attraction to the game awake and will help keeping the player entertained.

Extensibility: Since Dungeon Escape is an object-oriented programmed game, it is always possible to modify existing functions and add new features (e.g. new guns, levels and characters) to it by simply creating new classes. So that players' interest and excitement about the game can be sustained. These changes will not have a major effect on gameplay, but will give the game some more depth.

Performance: As everyone that has played game at least once knows, lagging and freezing are not fun. These problems may cause the user not to play the game, despite of all other great features that the game has. In order to prevent these problems, the game's performance must be high enough to keep player entertained. Dungeon Escape's responses to the player will be immediate and will run on at least 60 FPS so that the game will be flowing all the time and will not bother the player with performance related problems.

Reliability: Dungeon Master will be bug-free and non-crashing program because there will not be so many inputs to it (only inputs available are control buttons) and it will not use too much space in stack memory. Also, there will be checkpoints in each level to save player’s process in case of any interruption to the program.

1.3 Definitions, acronyms, and abbreviations

2D [1]: 2 Dimensional

FPS [2]: Frames per Second

1.4 References

[1] http://en.wikipedia.org/wiki/2D

[2] http://en.wikipedia.org/wiki/Frame_rate

1.5 Overview

In this ‘Introduction’ chapter, we briefly introduced main goal of our game and the criteria it is based on. We told that we preferred portability over high-performance by choosing Java as our programming language, talked about what we will do to make Dungeon Escape as interesting and entertaining as possible, gave some information about how to extend the game when felt needed or just simply when we want (or even more simply, when we are told by the instructor), mentioned what are our performance goals and what are our strategy to keep the game reliable.

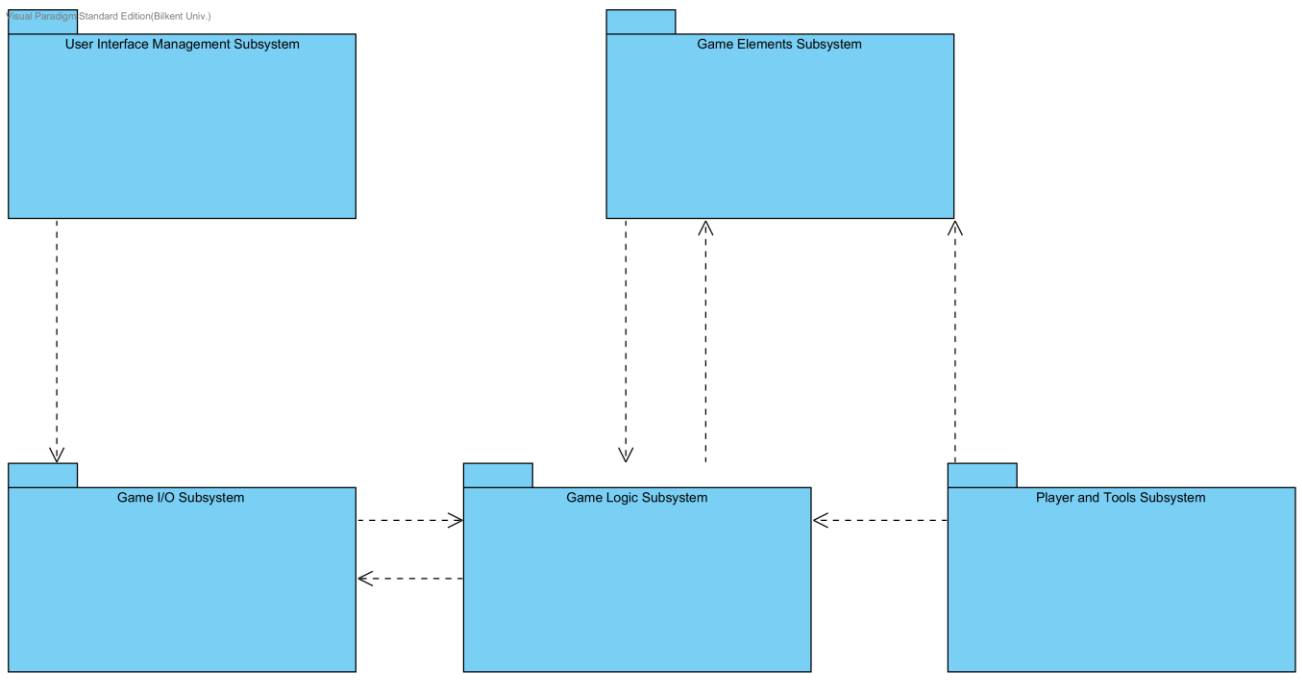
2 Software Architecture

2.1 Overview

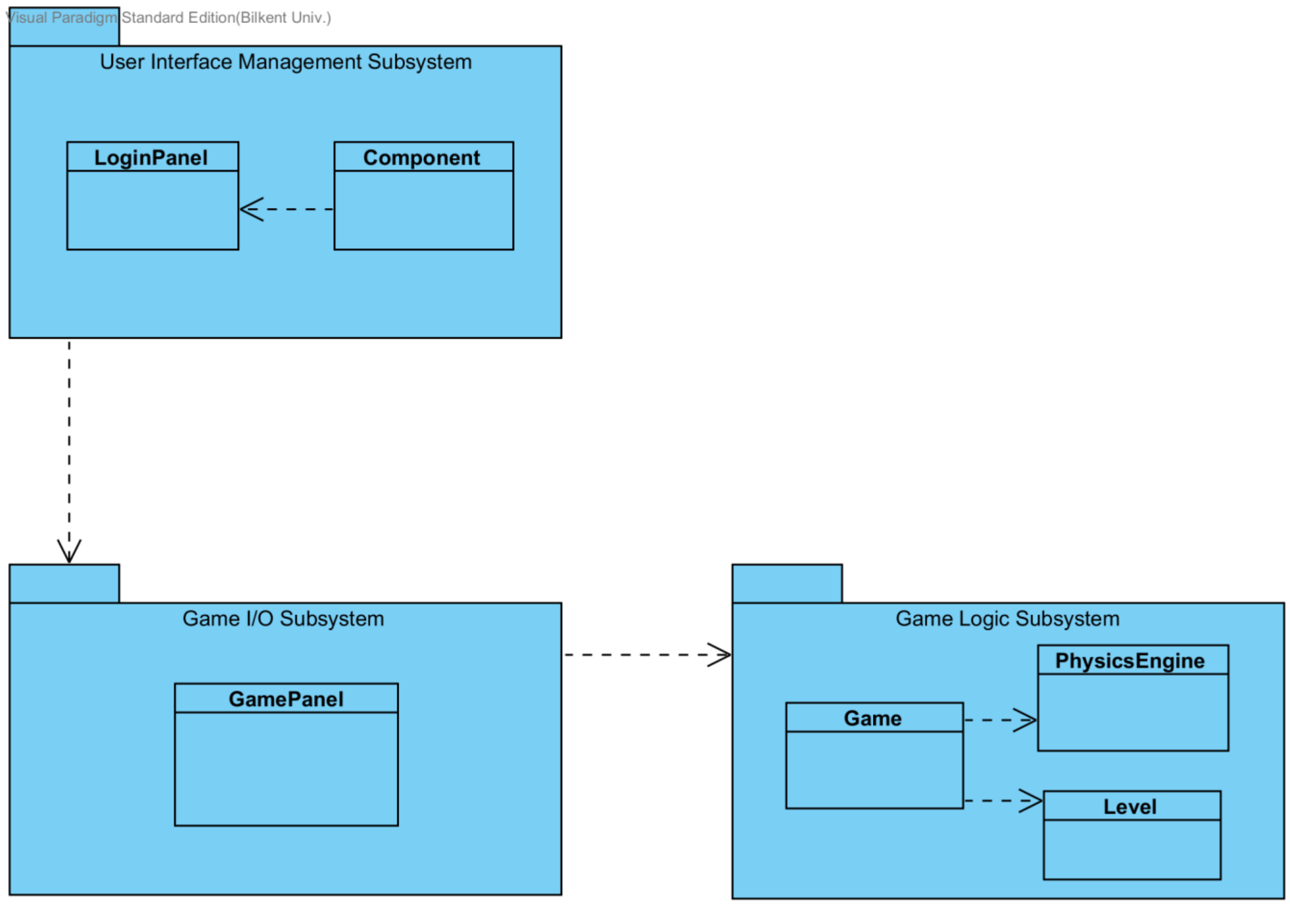
In this section it is explained that how Dungeon Escape is decomposed into subsystems, what hardware configurations and software features are, what is the general structure of account file system and security concept, how we deal with storing game data and finally what are the check points and how this system works in Dungeon Escape. In this game, we decided to use 3-tier architectural design because neither database nor network system is required in Dungeon Escape.

2.2 Subsystem Decomposition

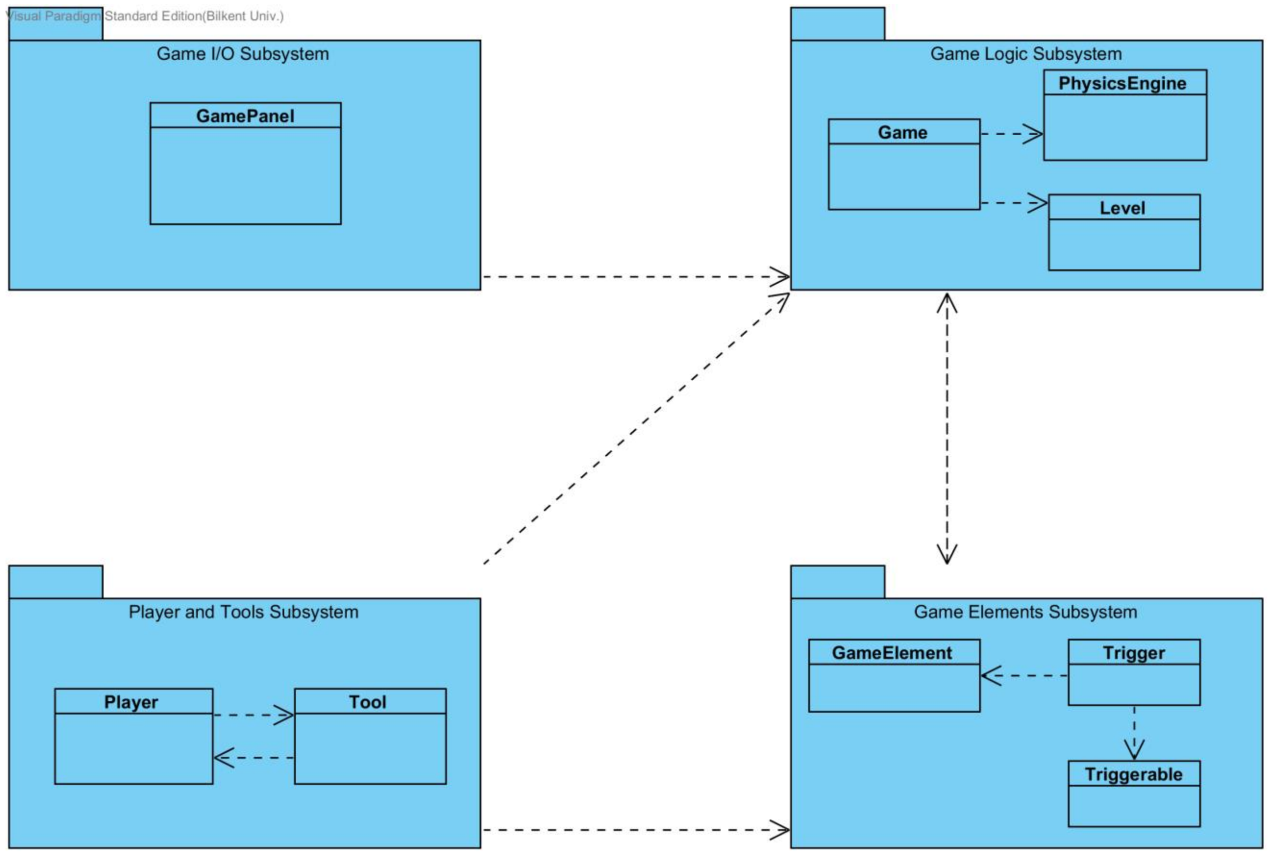
Among all architectural styles, we find out that the most suitable style is three-tier architecture. Since we have neither server nor database in our system, it only has Client and Controller components. Because of that three-tier is the best architectural style for our system. You can find the diagram describing three-tier principle of our design is depicted in the following figure.


Visual Paradigm Standard Edition(Bilkent Univ.)
User Interface Management Subsystem
Game Elements Subsystem
Game I/O Subsystem
Game Logic Subsystem
Player and Tools Subsystem

In presentation layer, we have classes that provides an interface to user for accessing the system. Therefore, the “LoginPanel” class, which is responsible for login process, is located in the UIMS package. The first interaction of the user with system starts at this level. Once the login process is done successfully, the system will direct to the “Component” class, which manages the choice of Menu type. “Component” will choose whether the submenu or the menu is desired and it will continue to do its operations. Afterwards, it delegates user’s choices to Game I/O and Game Logic Subsystems.



In the logic tier, the input of the user will be evaluated in the “Game I/O Subsystem”. The evaluation of the GIOS will direct to the “Game” class in the “Game Logic Subsystem.” GLS will construct the game layout by using the “PhysicsEngine” and “Level” classes. During the construction process of the GLS, the “Game” which is Façade class of the GLS will interact with GIOS to check the user inputs. After the logic is constructed, the system will direct to the Player and Tools Subsystem & Game Elements Subsystem to update its features.



In the operation layer, Player and Tool Subsystem determine what the user can or cannot do during the game play. “Player” class which is inside the PTS, will set the abilities of the game character such as, jump height, running speed, etc. “Tool” class will contain the items that the game character has and the operations that can be done by these items. Player class in PTS extends the “GameElement” class in Game Elements Subsystem. Inside the GES there are two classes which defines the structures and styles of the objects in the game. “Trigger” class and “Triggrable” interface bring dynamism to the game. In other words, with the help of these instances, the interactivity of the objects will be extended.

2.3 Hardware/Software Mapping

In Dungeon Escape, mouse and keyboard will be used as hardware configuration. Mouse will be used in the menus and in gameplay both to aim and change camera angle. Keyboard will be used in register and login processes as well as during gameplay to select and use tools, interact with map etc. Also, since Dungeon Escape will have sound feature, if an earphone or speaker isn't available then the user wouldn't be able to use this feature. Java's 2D graphics library (Graphics2D class) uses GPU. In software mapping part, Dungeon Escape is implemented in Java programming language. Alongside with graphics2D library we used awt and swing libraries of java.

2.4 Persistent Data Management

There will be no setup process for Dungeon Escape to be played and all the game data will be loaded on the user's computer's hard disk drive. Data is kept in files and files will be saved locally (e.g. usernames, passwords, last checkpoints of users) will be stored in a text file instead of database. In this project, database system is not required because database creates too much complexity, there isn't any extreme data flow because we deal with the data flow issue by keeping the data in the user's hard drive. Therefore, game will access to stored data faster and will avoid unnecessary complexity in the memory. Any data that is used in Dungeon Escape will be available in the local files and users will be able, also encouraged, to modify visuals (e.g. background image, main character's animation pngs) and sounds.

2.5 Access and Security

In this project, the user has a one level authentication. User name and password are required to register the game. After registration, the player will have an account file which stores the user name, password and the user's game current check point. These information will be accessible only by the user and the user will be able to customize his/her account. In this account, user's current checkpoints will be stored in an order that can be saved when needed.

2.6 Boundary Conditions

If our loading mechanic (level class) is unable to locate or load the active account's data because of any reason (e.g. corrupted save text file, missing save text file) the game will start from the last checkpoint that can be detected. There are two types of check points. One of them is called default check point which are located at the beginning of the each level. Second type of them is safety check points. There are several safety check points in each level. In the case of spending too much time in a level. Default check points are used in the case that only the user fails before the safety check point. If no check points can be detected the game will start from level 1 (first default check point).

3 Subsystem Services

3.1 User Interface Management Subsystem

UIMS provides our software system with Login process and graphical components. Additionally, it manages the transitions between the panels and the different menu styles. The reference of this subsystem is provided with the “Component” class which is considered as a manager of panels and the interface of the UIMS. UIMS contains four main classes and five subclasses, which provide an interface that contains login panel and the menu panel for the user. “Component” class is façade class of this subsystem.

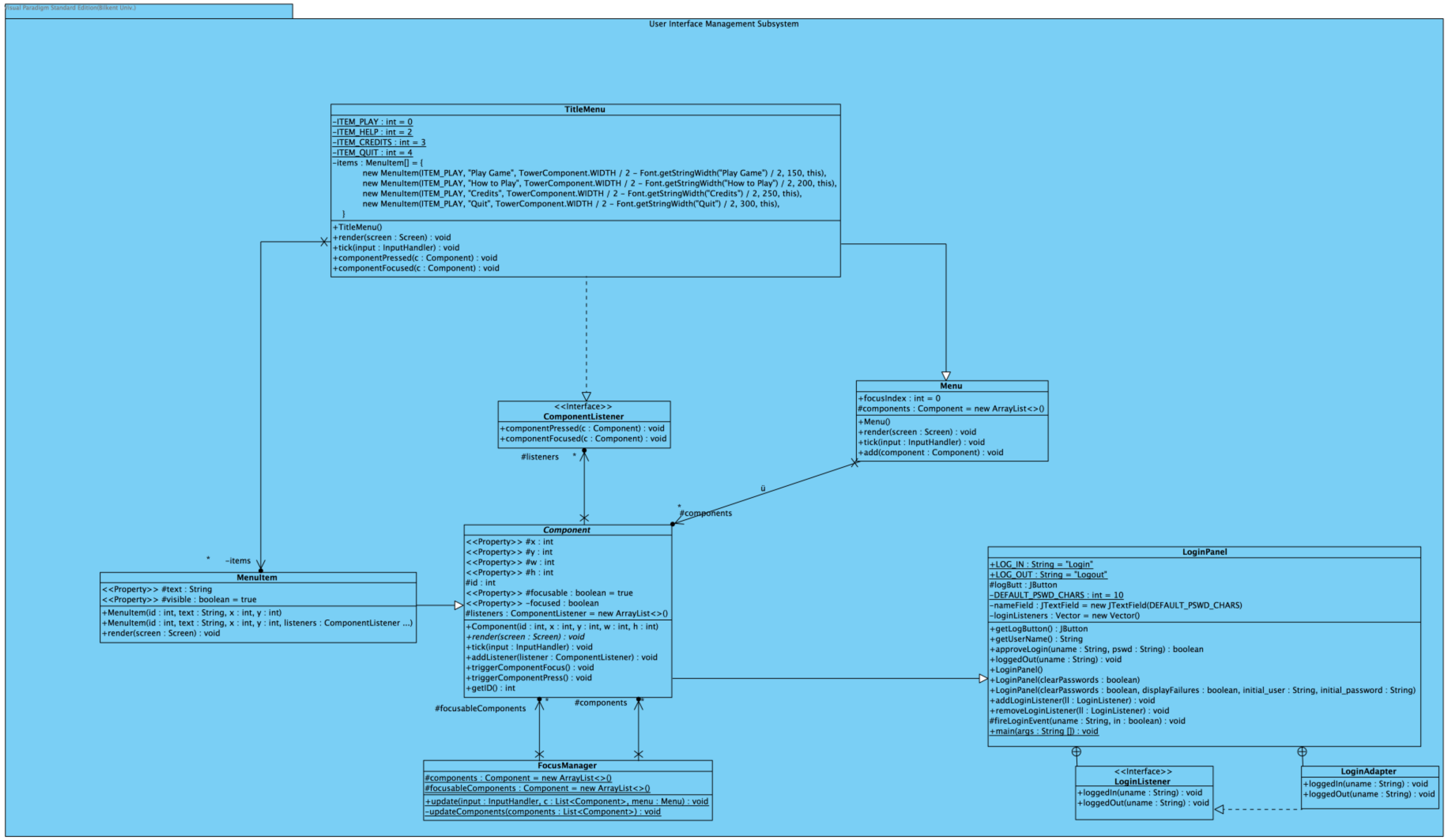

Visual Paradigm Standard Edition(Bilkent Univ.)
User Interface Management Subsystem
TitleMenu
-ITEM_PLAY : int = 0
-ITEM_HELP : int = 2
-ITEM_CREDITS : int = 3
-ITEM_QUIT : int = 4
-items : MenuItem[] = {
new MenuItem(ITEM_PLAY, "Play Game", TowerComponent.WIDTH / 2 - Font.getStringWidth("Play Game") / 2, 150, this),
new MenuItem(ITEM_PLAY, "How to Play", TowerComponent.WIDTH / 2 - Font.getStringWidth("How to Play") / 2, 200, this),
new MenuItem(ITEM_PLAY, "Credits", TowerComponent.WIDTH / 2 - Font.getStringWidth("Credits") / 2, 250, this),
new MenuItem(ITEM_PLAY, "Quit", TowerComponent.WIDTH / 2 - Font.getStringWidth("Quit") / 2, 300, this),
}
+TitleMenu()
+render(screen : Screen) : void
+tick(input : InputHandler) : void
+componentPressed(c : Component) : void
+componentFocused(c : Component) : void
<<Interface>>
ComponentListener
+componentPressed(c : Component) : void
+componentFocused(c : Component) : void
Menu
+focusIndex : int = 0
#components : Component = new ArrayList<>()
+Menu()
+render(screen : Screen) : void
+tick(input : InputHandler) : void
+add(component : Component) : void
#listeners
*
ü
*
#components
*
-items
MenuItem
<<Property>> #text : String
<<Property>> #visible : boolean = true
+MenuItem(id : int, text : String, x : int, y : int)
+MenuItem(id : int, text : String, x : int, y : int, listeners : ComponentListener ...)
+render(screen : Screen) : void
Component
<<Property>> #x : int
<<Property>> #y : int
<<Property>> #w : int
<<Property>> #h : int
#id : int
<<Property>> #focusable : boolean = true
<<Property>> -focused : boolean
#listeners : ComponentListener = new ArrayList<>()
+Component(id : int, x : int, y : int, w : int, h : int)
+render(screen : Screen) : void
+tick(input : InputHandler) : void
+addListener(listener : ComponentListener) : void
+triggerComponentFocus() : void
+triggerComponentPress() : void
+getID() : int
#focusableComponents
*
#components
*
FocusManager
#components : Component = new ArrayList<>()
#focusableComponents : Component = new ArrayList<>()
+update(input : InputHandler, c : List<Component>, menu : Menu) : void
-updateComponents(components : List<Component>) : void
LoginPanel
+LOG_IN : String = "Login"
+LOG_OUT : String = "Logout"
#logButt : JButton
-DEFAULT_PSWD_CHARS : int = 10
-nameField : JTextField = new JTextField(DEFAULT_PSWD_CHARS)
-loginListeners : Vector = new Vector()
+getLogButton() : JButton
+getUserName() : String
+approveLogin(uname : String, pswd : String) : boolean
+loggedOut(uname : String) : void
+LoginPanel()
+LoginPanel(clearPasswords : boolean)
+LoginPanel(clearPasswords : boolean, displayFailures : boolean, initial_user : String, initial_password : String)
+addLoginListener(ll : LoginListener) : void
+removeLoginListener(ll : LoginListener) : void
#fireLoginEvent(uname : String, in : boolean) : void
+main(args : String []) : void
<<Interface>>
LoginListener
+loggedIn(uname : String) : void
+loggedOut(uname : String) : void
LoginAdapter
+loggedIn(uname : String) : void
+loggedOut(uname : String) : void

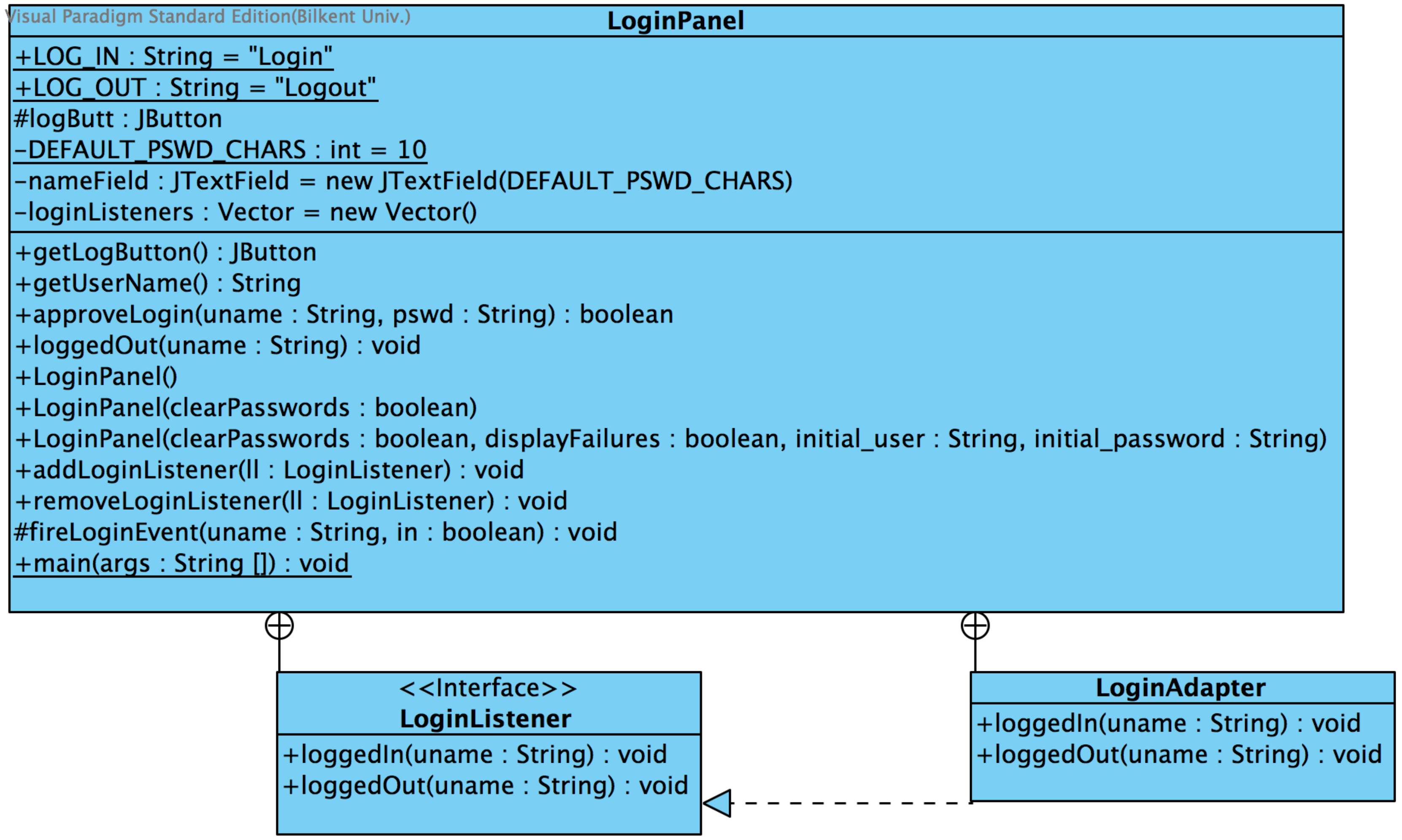
Figure: Package Diagram of UIMS

The “LoginPanel” will provide the “game access, after the basic login operation is done.

“Component” class provides a management system for checking the required class to display.

After getting the decision of the “Component” class, “Menu” class will provide the submenu, which will be displaying during the game and “TitleMenu” will provide the main menu at the beginning.

3.1.1 Login Panel Class


Visual Paradigm Standard Edition(Bilkent Univ.)
LoginPanel
+LOG_IN : String = "Login"
+LOG_OUT : String = "Logout"
#logButt : JButton
-DEFAULT_PSWD_CHARS : int = 10
-nameField : JTextField = new JTextField(DEFAULT_PSWD_CHARS)
-loginListeners : Vector = new Vector()
+getLogButton() : JButton
+getUserName() : String
+approveLogin(uname : String, pswd : String) : boolean
+loggedOut(uname : String) : void
+LoginPanel()
+LoginPanel(clearPasswords : boolean)
+LoginPanel(clearPasswords : boolean, displayFailures : boolean, initial_user : String, initial_password : String)
+addLoginListener(ll : LoginListener) : void
+removeLoginListener(ll : LoginListener) : void
#fireLoginEvent(uname : String, in : boolean) : void
+main(args : String []) : void
<<Interface>>
LoginListener
+loggedIn(uname : String) : void
+loggedOut(uname : String) : void
LoginAdapter
+loggedIn(uname : String) : void
+loggedOut(uname : String) : void

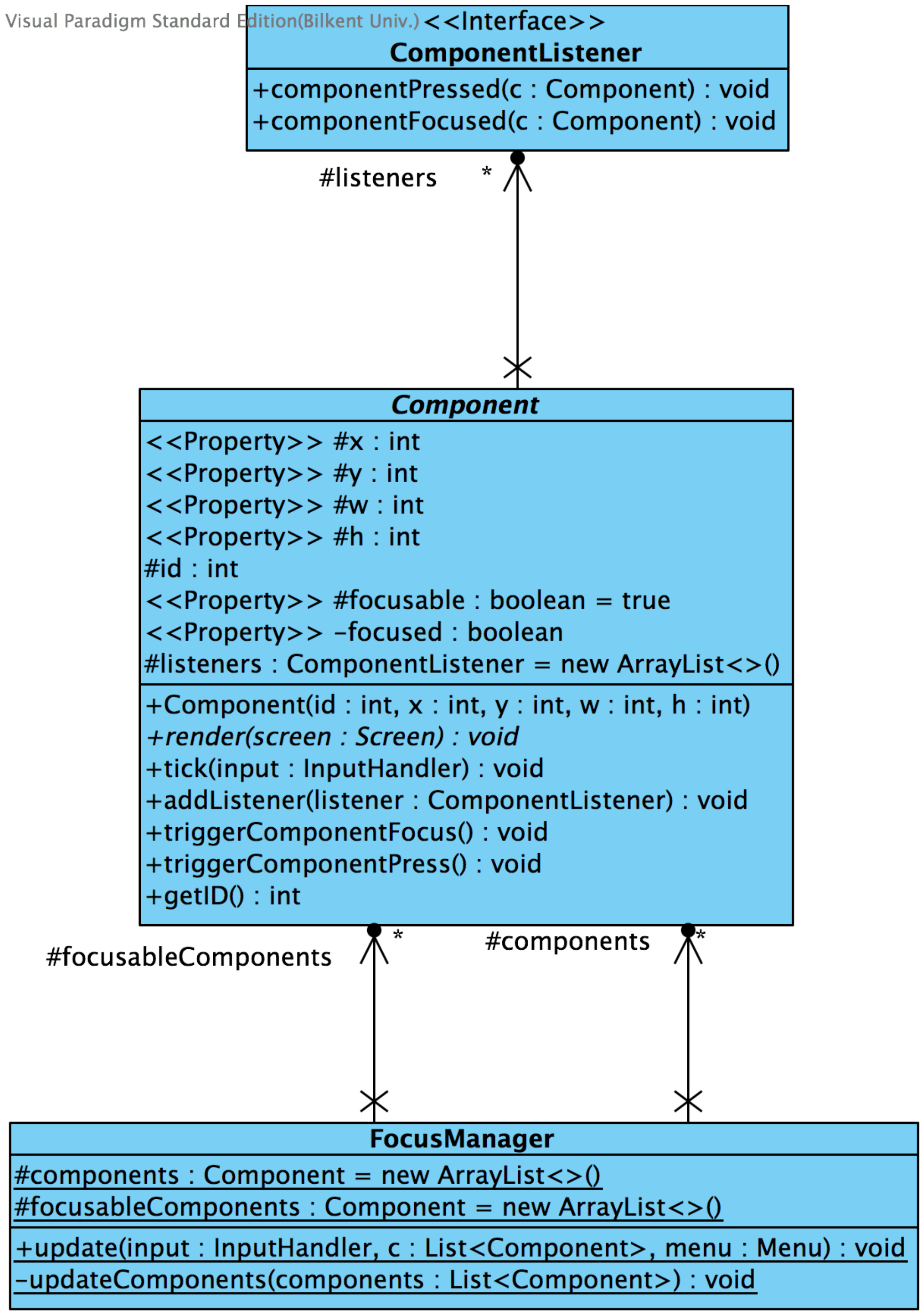
“LoginPanel” is the first class that will be instantiated when the game is first executed and it will display the Login Page for the user. If the user has an account and enters the right username and password, the “LoginPanel” will give an

access to the game.

loggedIn() = Returns the login situation of the user if the login process is succeeded..

LoggedOut() = Returns current the login situation of the user if the login process is failed.

3.1.2 Component Class

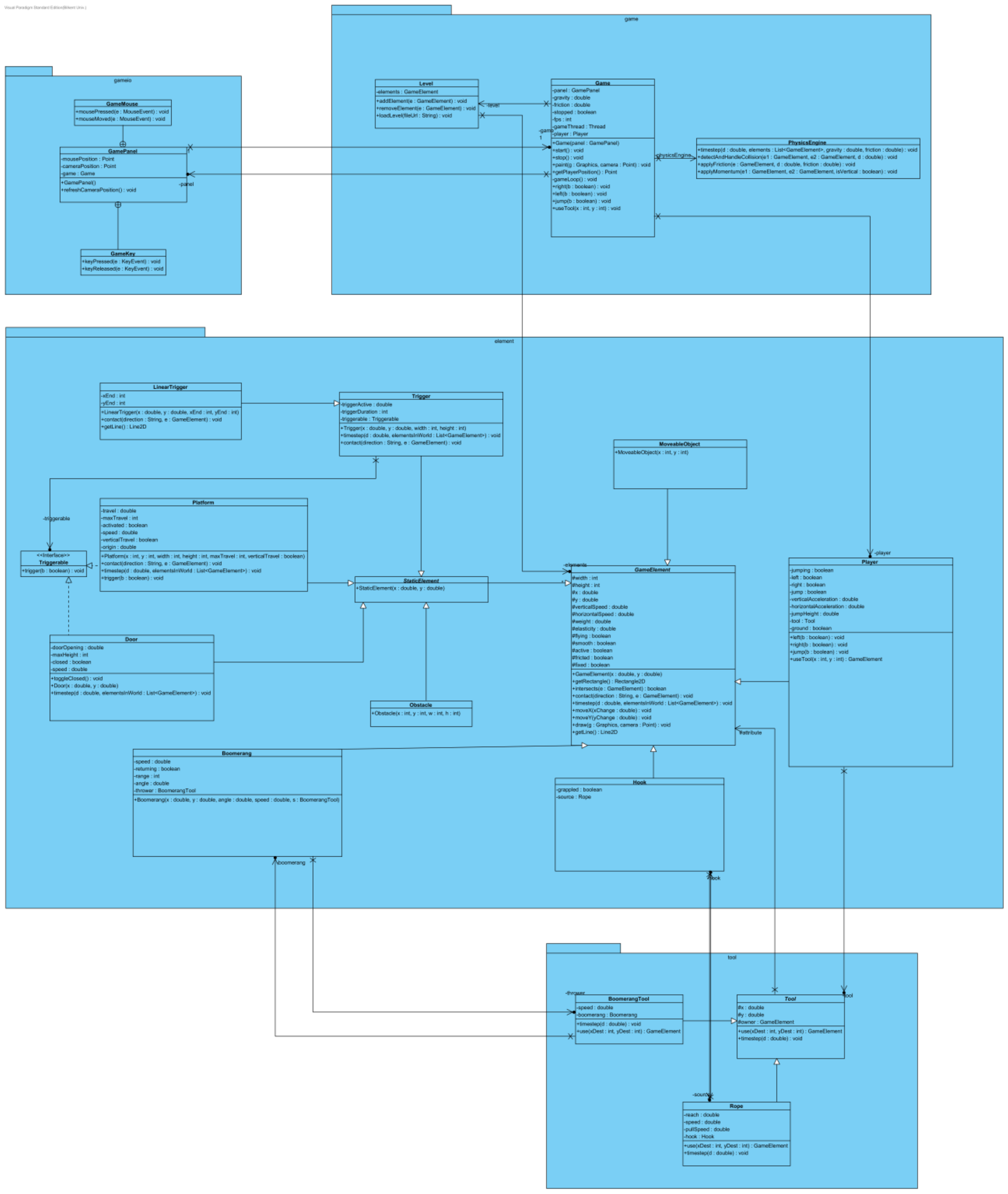


Visual Paradigm Standard Edition(Bilkent Univ.)
<<Interface>>
ComponentListener
+componentPressed(c : Component) : void
+componentFocused(c : Component) : void
#listeners
*
Component
<<Property>> #x : int
<<Property>> #y : int
<<Property>> #w : int
<<Property>> #h : int
#id : int
<<Property>> #focusable : boolean = true
<<Property>> -focused : boolean
#listeners : ComponentListener = new ArrayList<>()
+Component(id : int, x : int, y : int, w : int, h : int)
+render(screen : Screen) : void
+tick(input : InputHandler) : void
+addListener(listener : ComponentListener) : void
+triggerComponentFocus() : void
+triggerComponentPress() : void
+getID() : int
#focusableComponents
*
#components
*
FocusManager
#components : Component = new ArrayList<>()
#focusableComponents : Component = new ArrayList<>()
+update(input : InputHandler, c : List<Component>, menu : Menu) : void
-updateComponents(components : List<Component>) : void

“Component” class is decision maker for our game. It gives a result for choosing whether the required menu style is main menu or sub menu. “Focus Manager” controls the decision focuses of the “Component” class.

The getID() method will receive an id which states the Menu style decision.

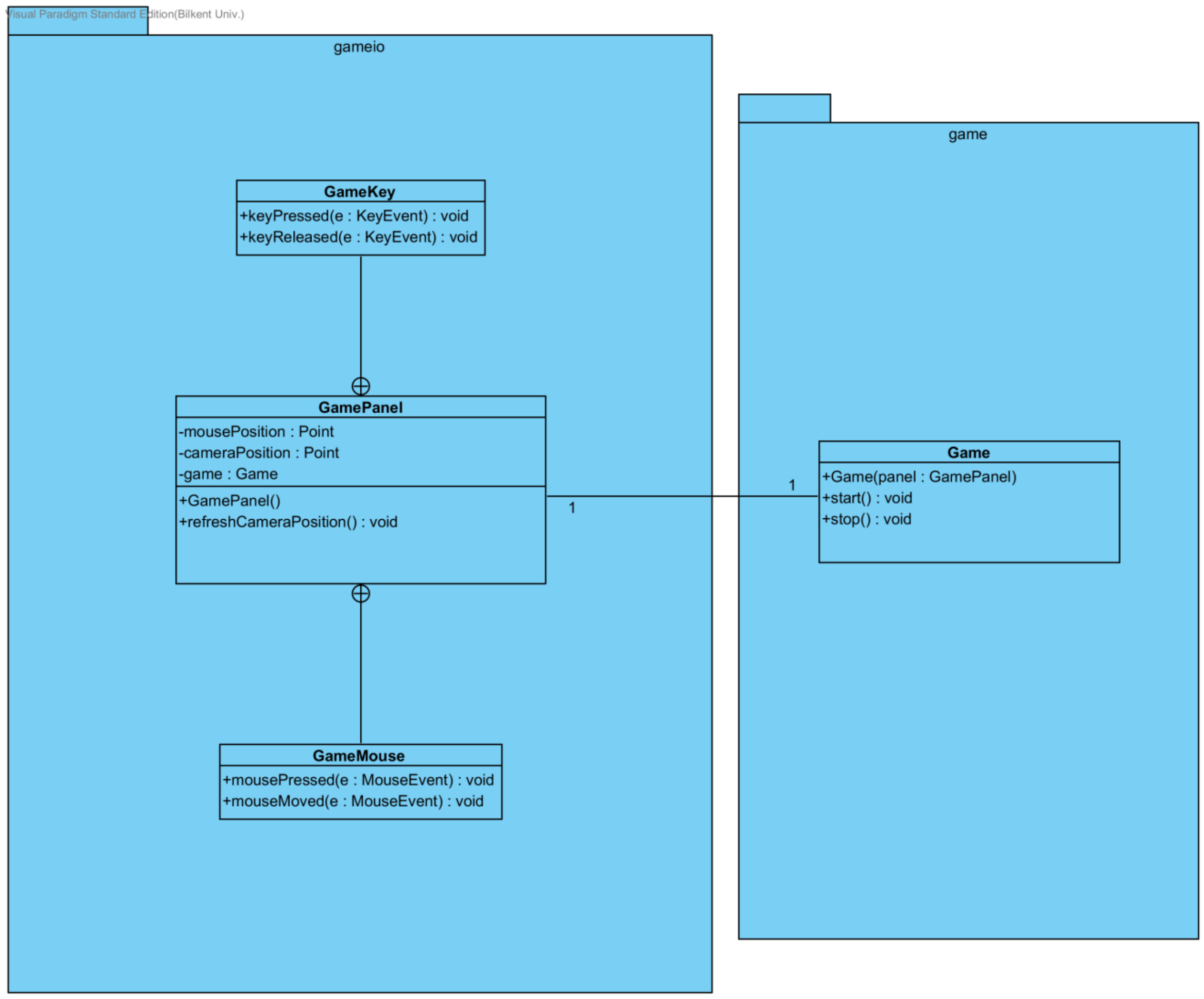
3.2 *Detailed Object Design (excluding UIMS)*


Visual Paradigm Standard Edition(Bilkent Univ.)
gameio
GameMouse
+mousePressed(e : MouseEvent) : void
+mouseMoved(e : MouseEvent) : void
GamePanel
-mousePosition : Point
-cameraPosition : Point
-game : Game
+GamePanel()
+refreshCameraPosition() : void
-panel
GameKey
+keyPressed(e : KeyEvent) : void
+keyReleased(e : KeyEvent) : void
game
Level
-elements : GameElement
+addElement(e : GameElement) : void
+removeElement(e : GameElement) : void
+loadLevel(fileUrl : String) : void
-level
Game
-panel : GamePanel
-gravity : double
-friction : double
-stopped : boolean
-fps : int
-gameThread : Thread
-player : Player
+Game(panel : GamePanel)
+start() : void
+stop() : void
+paint(g : Graphics, camera : Point) : void
+getPlayerPosition() : Point
-gameLoop() : void
+right(b : boolean) : void
+left(b : boolean) : void
+jump(b : boolean) : void
+useTool(x : int, y : int) : void
-game
1
-physicsEngine
PhysicsEngine
+timestep(d : double, elements : List<GameElement>, gravity : double, friction : double) : void
+detectAndHandleCollision(e1 : GameElement, e2 : GameElement, d : double) : void
+applyFriction(e : GameElement, d : double, friction : double) : void
+applyMomentum(e1 : GameElement, e2 : GameElement, isVertical : boolean) : void
element
LinearTrigger
-xEnd : int
-yEnd : int
+LinearTrigger(x : double, y : double, xEnd : int, yEnd : int)
+contact(direction : String, e : GameElement) : void
+getLine() : Line2D
Trigger
-triggerActive : double
-triggerDuration : int
-triggerable : Triggerable
+Trigger(x : double, y : double, width : int, height : int)
+timestep(d : double, elementsInWorld : List<GameElement>) : void
+contact(direction : String, e : GameElement) : void
-triggerable
<<Interface>>
Triggerable
+trigger(b : boolean) : void
Platform
-travel : double
-maxTravel : int
-activated : boolean
-speed : double
-verticalTravel : boolean
-origin : double
+Platform(x : int, y : int, width : int, height : int, maxTravel : int, verticalTravel : boolean)
+contact(direction : String, e : GameElement) : void
+timestep(d : double, elementsInWorld : List<GameElement>) : void
+trigger(b : boolean) : void
StaticElement
+StaticElement(x : double, y : double)
MoveableObject
+MoveableObject(x : int, y : int)
-elements
*
GameElement
#width : int
#height : int
#x : double
#y : double
#verticalSpeed : double
#horizontalSpeed : double
#weight : double
#elasticity : double
#flying : boolean
#smooth : boolean
#active : boolean
#fricted : boolean
#fixed : boolean
+GameElement(x : double, y : double)
+getRectangle() : Rectangle2D
+intersects(e : GameElement) : boolean
+contact(direction : String, e : GameElement) : void
+timestep(d : double, elementsInWorld : List<GameElement>) : void
+moveX(xChange : double) : void
+moveY(yChange : double) : void
+draw(g : Graphics, camera : Point) : void
+getLine() : Line2D
#attribute
-player
Player
-jumping : boolean
-left : boolean
-right : boolean
-jump : boolean
-verticalAcceleration : double
-horizontalAcceleration : double
-jumpHeight : double
-tool : Tool
-ground : boolean
+left(b : boolean) : void
+right(b : boolean) : void
+jump(b : boolean) : void
+useTool(x : int, y : int) : GameElement
Door
-doorOpening : double
-maxHeight : int
-closed : boolean
-speed : double
+toggleClosed() : void
+Door(x : double, y : double)
+timestep(d : double, elementsInWorld : List<GameElement>) : void
Obstacle
+Obstacle(x : int, y : int, w : int, h : int)
Boomerang
-speed : double
-returning : boolean
-range : int
-angle : double
-thrower : BoomerangTool
+Boomerang(x : double, y : double, angle : double, speed : double, s : BoomerangTool)
-boomerang
Hook
-grappled : boolean
-source : Rope
-hook
tool
-thrower
BoomerangTool
-speed : double
-boomerang : Boomerang
+timestep(d : double) : void
+use(xDest : int, yDest : int) : GameElement
Tool
#x : double
#y : double
#owner : GameElement
+use(xDest : int, yDest : int) : GameElement
+timestep(d : double) : void
-tool
-source
Rope
-reach : double
-speed : double
-pullSpeed : double
-hook : Hook
+use(xDest : int, yDest : int) : GameElement
+timestep(d : double) : void

This UML diagram provides an understanding about the interactions between the classes and main features of our software. These classes will be more individually detailed in the following pages.

3.3 *Game I/O Subsystem*

3.3.1 GamePanel Class



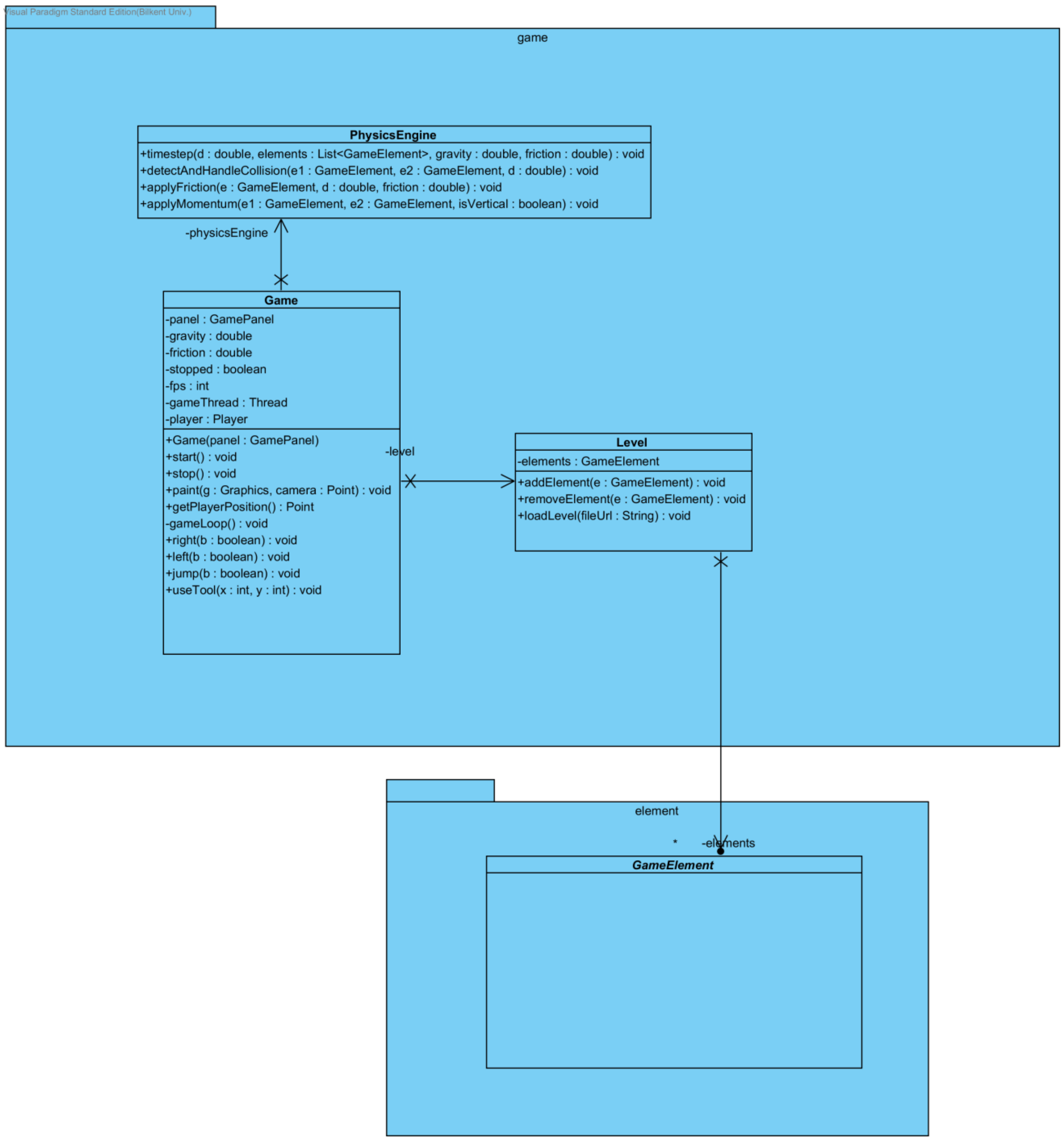
Visual Paradigm Standard Edition(Bilkent Univ.)
gameio
GameKey
+keyPressed(e : KeyEvent) : void
+keyReleased(e : KeyEvent) : void
GamePanel
-mousePosition : Point
-cameraPosition : Point
-game : Game
+GamePanel()
+refreshCameraPosition() : void
GameMouse
+mousePressed(e : MouseEvent) : void
+mouseMoved(e : MouseEvent) : void
1
1
game
Game
+Game(panel : GamePanel)
+start() : void
+stop() : void

Display and input control is done by a subclass of JPanel called GamePanel. This class is responsible for displaying the elements within the game, calculate the offset of display (offset is dynamic depending on the position of the mouse cursor) and inform the game itself on any mouse click or keystroke.

refreshCameraPosition(): Calculates the position of the object drawing offset depending on the player’s position within the game and the position of the mouse cursor.

3.4 *Game Logic Subsystem*

3.4.1 Game Class



Visual Paradigm Standard Edition(Bilkent Univ.)
game
PhysicsEngine
+timestep(d : double, elements : List<GameElement>, gravity : double, friction : double) : void
+detectAndHandleCollision(e1 : GameElement, e2 : GameElement, d : double) : void
+applyFriction(e : GameElement, d : double, friction : double) : void
+applyMomentum(e1 : GameElement, e2 : GameElement, isVertical : boolean) : void
-physicsEngine
Game
-panel : GamePanel
-gravity : double
-friction : double
-stopped : boolean
-fps : int
-gameThread : Thread
-player : Player
+Game(panel : GamePanel)
+start() : void
+stop() : void
+paint(g : Graphics, camera : Point) : void
+getPlayerPosition() : Point
-gameLoop() : void
+right(b : boolean) : void
+left(b : boolean) : void
+jump(b : boolean) : void
+useTool(x : int, y : int) : void
-level
Level
-elements : GameElement
+addElement(e : GameElement) : void
+removeElement(e : GameElement) : void
+loadLevel(fileUrl : String) : void
element
*
-elements
GameElement

This class is responsible for executing the physics engine on a set of GameElement objects on specific time intervals. It contains a Level object which contains a list of GameElements. GameElement objects are the building block of the levels within the game. Game class periodically calls the timeStep function of PhysicsEngine class which moves the GameElements one by one depending on their existing velocities and applies gravity and friction. PhysicsEngine also handles the collisions between objects and applies momentum on collisions.

start(): This method creates a new thread and calls the gameLoop method. Since this is a never ending loop until the game is paused or the level is completed, a new thread is necessary in order to keep the other functionalities of the program going.

gameLoop(): This method creates a loop which runs the timeStep method of PhysicsEngine class once every specific time period (60 times a second by default).

stop(): Stops the engine thread, can be started again by recalling start method.

3.4.2 PhysicsEngine Class

This class consists of a set of methods without any properties.

timestep(double): This method determines how the physics inside the game is going to work. It takes the reference of GameElement list and cross checks each GameElement for collisions and takes action depending on the type and elasticity of the elements colliding. Also applies gravity, friction and momentum on GameElement objects.

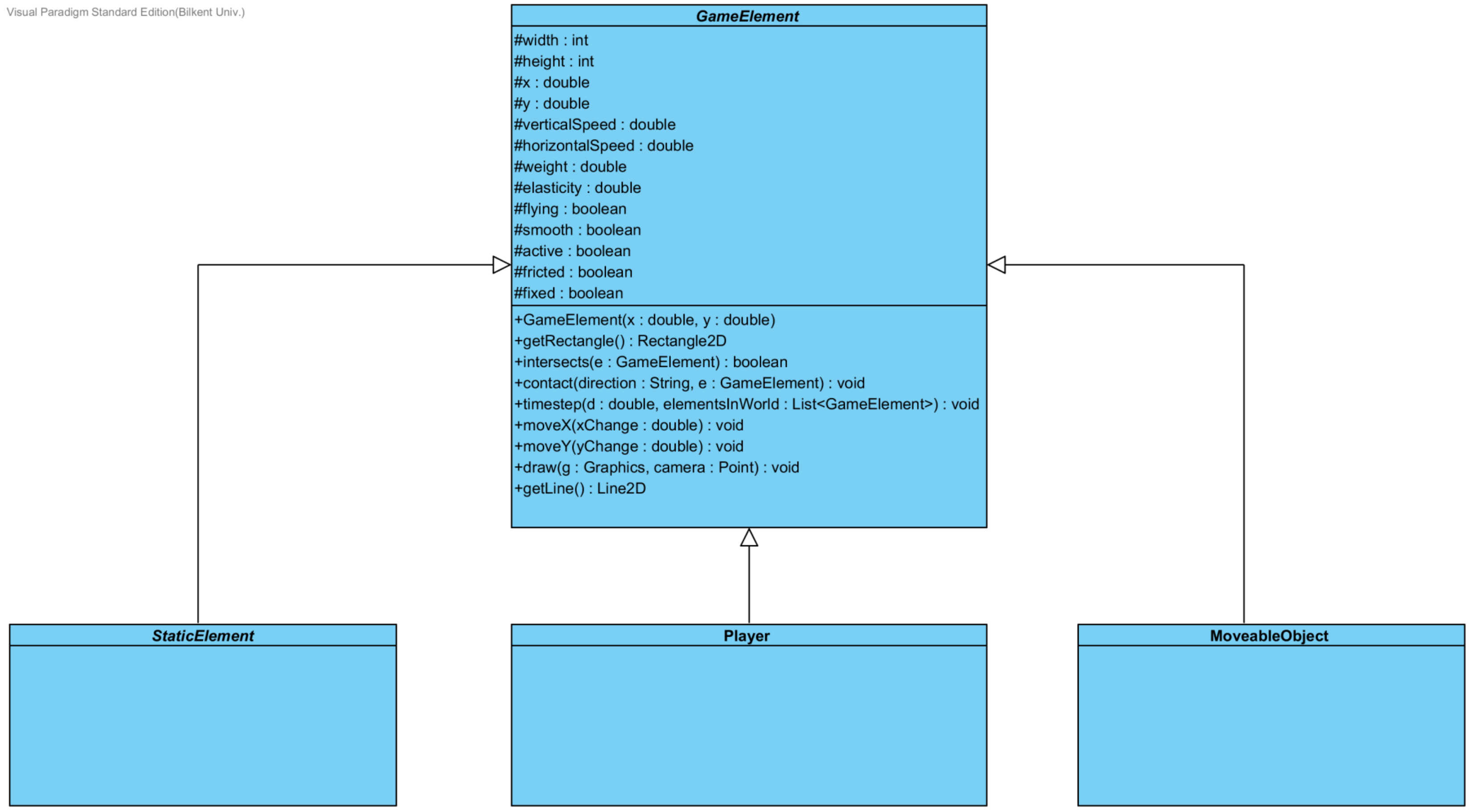
3.4.3 Level Class

This class keeps the reference of a GameElement list which practically means it keeps the reference of the actual specific level.

loadLevel(String): This method loads the GameElement objects from a file with a specific parsing algorithm.

3.5 Game Elements Subsystem

3.5.1 GameElement Class


Visual Paradigm Standard Edition(Bilkent Univ.)
GameElement
#width : int
#height : int
#x : double
#y : double
#verticalSpeed : double
#horizontalSpeed : double
#weight : double
#elasticity : double
#flying : boolean
#smooth : boolean
#active : boolean
#fricted : boolean
#fixed : boolean
+GameElement(x : double, y : double)
+getRectangle() : Rectangle2D
+intersects(e : GameElement) : boolean
+contact(direction : String, e : GameElement) : void
+timestep(d : double, elementsInWorld : List<GameElement>) : void
+moveX(xChange : double) : void
+moveY(yChange : double) : void
+draw(g : Graphics, camera : Point) : void
+getLine() : Line2D
StaticElement
Player
MoveableObject

This class itself is abstract and can't be instantiated. All elements with physical presence on a level extends this class.

x, y, width, height: Position and dimensions of this element's rectangular area.

verticalSpeed, horizontalSpeed: Velocity of the element on specified axis.

weight, elasticity: Affects the way this object's collisions are handled (momentum).

flying: If this property is true, this element is not affected by gravity.

smooth: If this property is false, this element will not be physically affected by collisions (But will still be informed in case it collides with another object).

active: This object will be removed from the level by the PhysicsEngine on next timestep if this property is false.

fricted: If this property is false, friction will not be applied on this object.

fixed: If this property is true, this object will not be physically affected by the PhysicsEngine but other objects will still collide with it, meaning that on ly the colliding object will be affected by the collision.

getRectangle(): All objects in the game have rectangular hit boxes. intersects method calls this method to check if there is any collision between two elements.

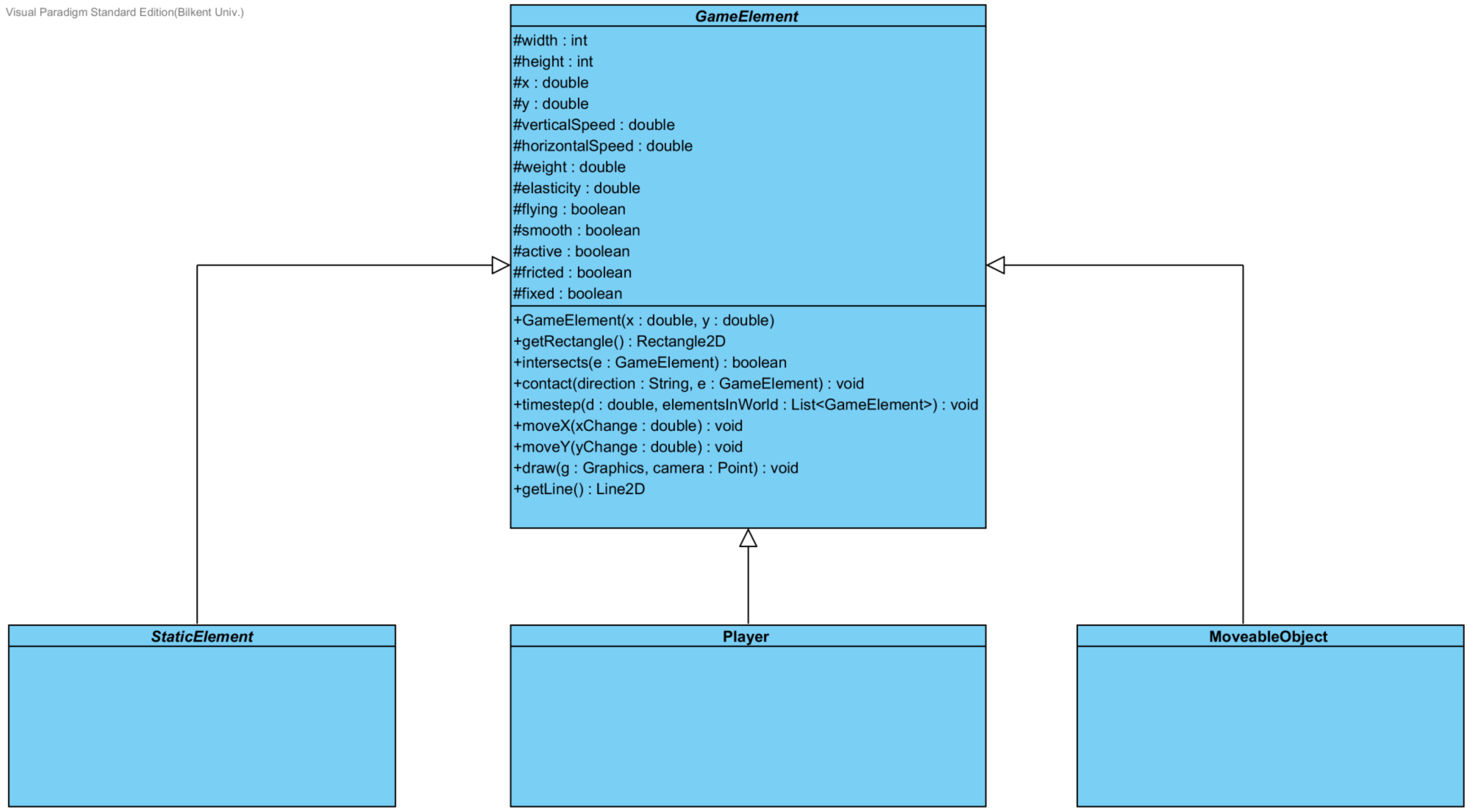
intersects(GameElement): Checks if the object collides with said another GameElement.

contact(String, GameElement): PhysicsEngine informs the object of the collision and gives the reference of the colliding object and gives its direction (left, right, top, bottom). It is an empty method by default, extending element can override it for element specific actions on contacts.

timestep(double, List<GameElement>): This method is called on every loop of PhysicsEngine timestep call (not to be confused with timestep method inside PhysicsEngine). It is empty by default but some elements might require specific actions to be performed other than physics every timestep, it can be overridden to fulfill that possible element specific requirement.

draw(Graphics, Point): Draws the object on a given Graphics object with a given offset. Should be overridden to make sure all objects on the level look different.

3.5.2 StaticElement and Triggers



Visual Paradigm Standard Edition(Bilkent Univ.)
GameElement
#width : int
#height : int
#x : double
#y : double
#verticalSpeed : double
#horizontalSpeed : double
#weight : double
#elasticity : double
#flying : boolean
#smooth : boolean
#active : boolean
#fricted : boolean
#fixed : boolean
+GameElement(x : double, y : double)
+getRectangle() : Rectangle2D
+intersects(e : GameElement) : boolean
+contact(direction : String, e : GameElement) : void
+timestep(d : double, elementsInWorld : List<GameElement>) : void
+moveX(xChange : double) : void
+moveY(yChange : double) : void
+draw(g : Graphics, camera : Point) : void
+getLine() : Line2D
StaticElement
Player
MoveableObject

StaticElement: These objects are just implementations of GameElements which have their “fixed” property true. However this does not mean that they are stationary in the level. They do not have unpredictable movements like the Player or MoveableObject but door and platform can have a periodic predictable movement if they are triggered.

Trigger: A trigger is a non-smooth object by default. When a non-static element passes through this object, this object calls the trigger method of a set Triggerable (e.g. opening a door, raising a platform).

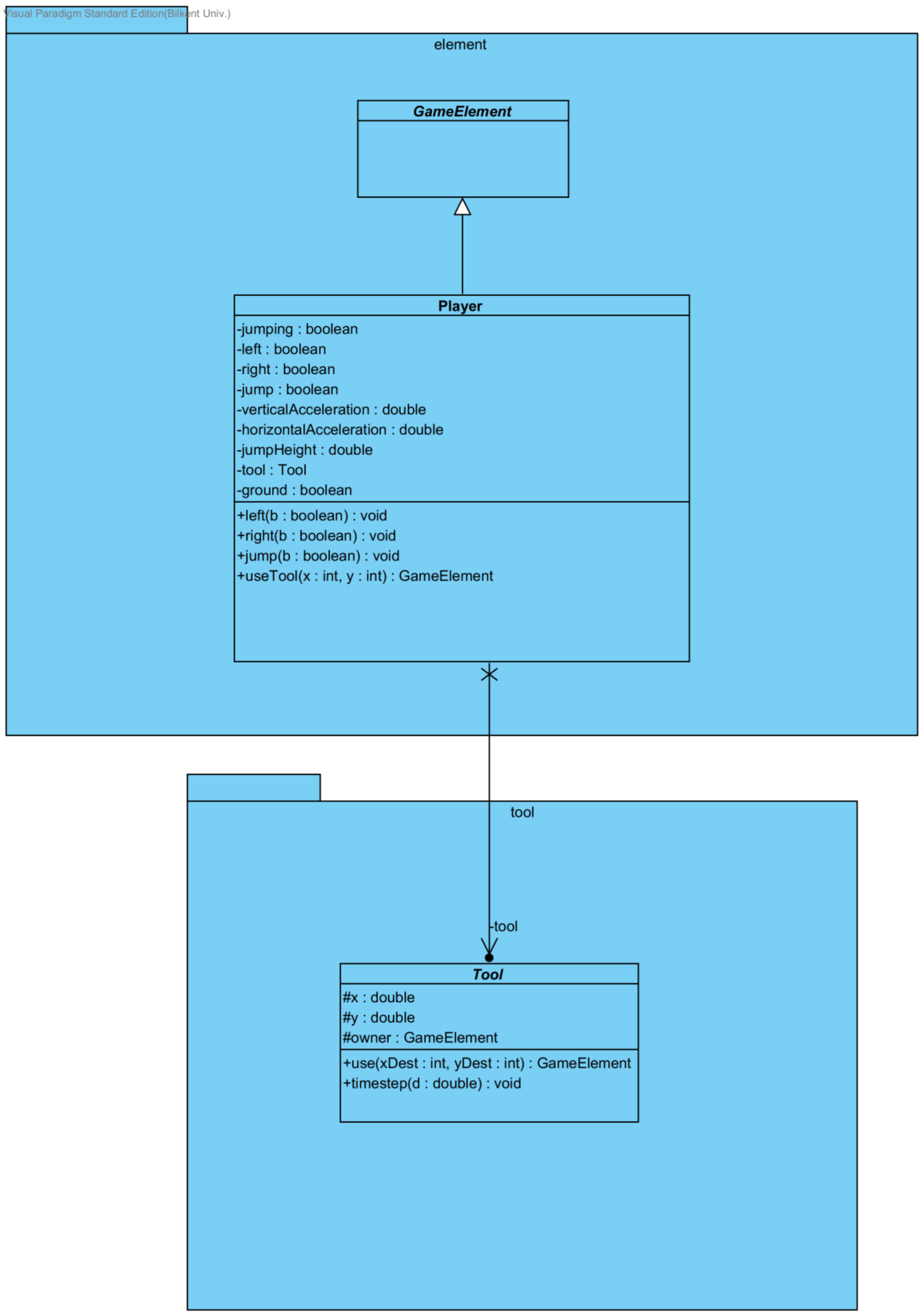
Triggerable: This is an interface which can be implemented by anything. It can be an element which moves accordingly when triggered or an ordinary object which may end the level when triggered.

Door: This element is a thin tall object which decreases height towards upwards when triggered. It may block the path of the player in-game, player must find a way to trigger it to get through.

Platform: This element is similar to Door but it is more complex. It can be set to move any amount of pixels and its direction and axis can be set. Its timestep function is overridden to check for any non-fixed objects and drag them with it (to ensure that a player standing on top of it during movement does not fall off).

3.6 *Player and Tools Subsystem*

3.6.1 Player Class



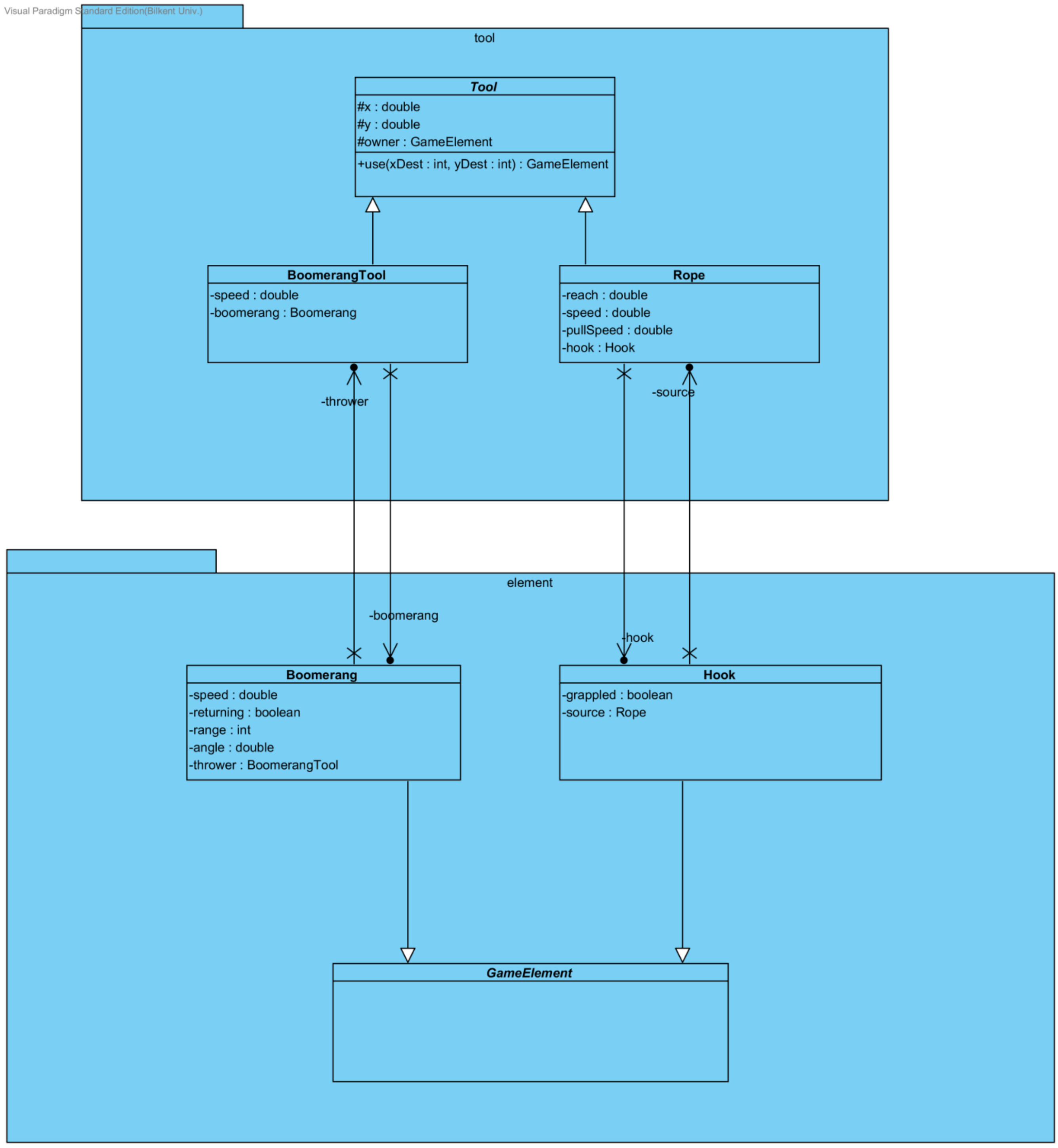
Visual Paradigm Standard Edition(Bilkent Univ.)
element
GameElement
Player
-jumping : boolean
-left : boolean
-right : boolean
-jump : boolean
-verticalAcceleration : double
-horizontalAcceleration : double
-jumpHeight : double
-tool : Tool
-ground : boolean
+left(b : boolean) : void
+right(b : boolean) : void
+jump(b : boolean) : void
+useTool(x : int, y : int) : GameElement
tool
-tool
Tool
#x : double
#y : double
#owner : GameElement
+use(xDest : int, yDest : int) : GameElement
+timestep(d : double) : void

Player class is one of the most important elements in the game. This is the element which represents the main character of the game and it is directly controlled by the user. It has the ability to jump up to a set height and can run horizontally with a set speed.

tool: A tool object which can be used by the user with a mouse click.

use(): Receives mouse location from GamePanel and passes it over to the Tool class for execution.

3.6.2 Tool Class



This abstract class enables the user to utilize mechanics other than the regular game mechanics such as running and jumping. This class can be extended in order to give new abilities to the user and mechanics to the game.

BoomerangTool: This tool is more like a control class to enable the user to throw a boomerang which comes back to the player after reaching its maximum range or after any collision. It can activate triggers and push other moveable objects. The player will be able to use this tool on certain levels (depending on the level design) and these levels will possibly require the user to use this tool cleverly to activate triggers remotely or to relocate moveable objects.

Rope: Just like BoomerangTool, this class controls the Hook object it produces when it is used. Hook goes away from the player until the maximum range is reached or until it collides with another object. But unlike Boomerang object, this object pulls the player to its point of collision (if it collides with anything).

4 Glossary

2D [1]: 2 Dimensional

FPS [2]: Frames per Second

UIMS: User Interface Management Subsystem

GES: Game Elements Subsystem

GIOS: Game I/O Subsystem

GLS: Game Logic Subsystem

PTS: Player and Tools Subsystem