CS 319 - Object-Oriented Software Engineering

System Design Report

Dave Davesson

Group 6

Ali Atli

Berk Türk

Burak Özmen

Sefa Gündoğdu

Contents

1. Introduction

1.1 Purpose of the system

Dave Davesson is a platform game that is inspired by the famous MSDOS game, Dangerous Dave. Compared to its predecessor, it offers better graphics and some different features. Also, the levels are designed in such a way that it gives the same taste as the original. However, it differs from the Dangerous Dave by its new feature, multiplayer, which enables two players to fight with each other with their preferred weapon type. Also, the game will be user friendly, easy to play while having a good time.

1.2 Design Goals

End User Criteria:

Ease of Use: In designing games, it is important to make the game friendly and comfortable. Because if the game looks friendly to a user, it seems more attractive. So, the game will offer player ease with easy character management, easy interactions and uncomplicated interface. In main menu, there will be only buttons for understandable functions like high score, help and play game. Also, the game will only need several buttons for each player, so it is easy to master. In addition, the multiplayer part offers players to not get bored easily, because in multiplayer mode, every game's outcome can be different.

Ease of Learning: The game will have a help section to guide the new players about the buttons, the power-ups, win conditions for both modes and weapons that are used in multiplayer. Also, since it is a platform game with basic dangers and obstacles, the base of the game can be easily learned.

Maintenance Criteria:

Extendibility: For games to be successful, it is important them to be extendible. Since Dave Davesson is a platform game that has maze-like levels, the levels can easily be modified or new levels can easily be added and these modifications can cause little to no bugs in game because it only changes a section that is already working properly. Also, in multiplayer, different types can be added other than 1v1 fight, like players vs enemy(s).

Portability: We are decided that the game will be implemented in Java, because for the game to run properly on various operating systems, Java is one of the most optimized languages as long as the platform has JRE installed. By making the game portable, we gave in about some performance in other languages such as C++, but since it is a simple game, we think that the performance loss seems negligible.

Modifiability: Since the game can be easily modifiable, because the subsystems' optimization with each other will be done while implementing. So, when the user wants to modify the areas, such as adding new features or updating an existing one, the program will still continue properly as long as the core features still remain.

Performance Criteria:

Response Time: In games, it is important to have high and stable fps, because fps is a sign of how the game runs smoothly. We intend to make the game will update itself with every step of Dave to make it optimizable as possible. The program will give response to various things like collusions, movements and projectiles as fast as possible to give a smooth gaming experience.

Trade Offs:

Ease of Use and Ease of Learning vs. Functionality: Like we mentioned on previous sections, since it is a game, the learning curve has to be low in order for player to enjoy the game. So, we decided to make the usability our priority rather than functionality. Rather than complex features and interfaces, the program's functions will be as basic as it can for not making player's mind lost in different various features.

Performance vs. Memory: Based on details in our program about the implementation period, it is important to make the collusions, animations and Dave's movements as smooth as possible. Also, because of the reason that Dave's and enemies' every move will update the game simultaneously. Because of that, we intend to make performance our priority. However, the game will still use a good chunk of memory, because for levels, a lot of blocks and various collectibles will remain on the field. These objects and their location will be stored in memory and when the level called, the field will update itself.

1.3. Overview

This section represents the main goal of implanting this software. Basically, the main goal is to make the user enjoy the time while playing the game as much as possible. For achieving this goal, our design focuses are determined. Because of the focuses, we made some decisions for the software to implement according to a game’s need. We made ease of use and learning in our priority in comparison to functionality and performance and smoothness in comparison to memory.

2. Software Architecture

2.1. Subsystem Decomposition

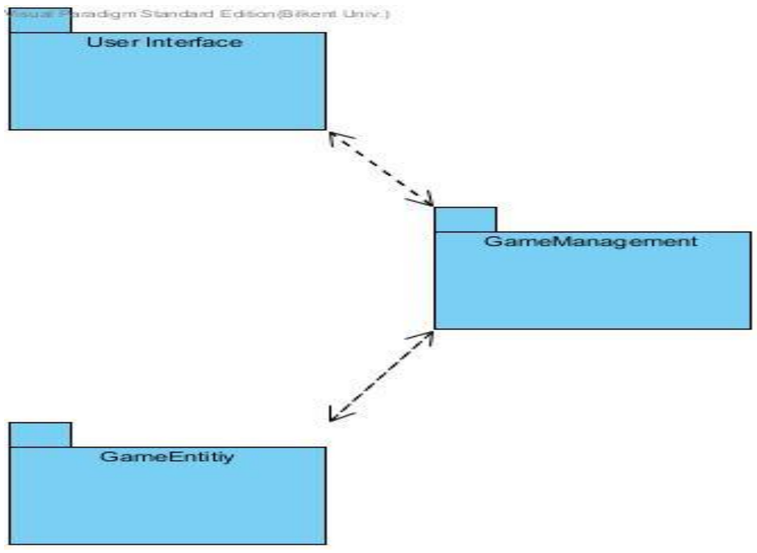


Figure 1 - Basic Subsystem Decomposition

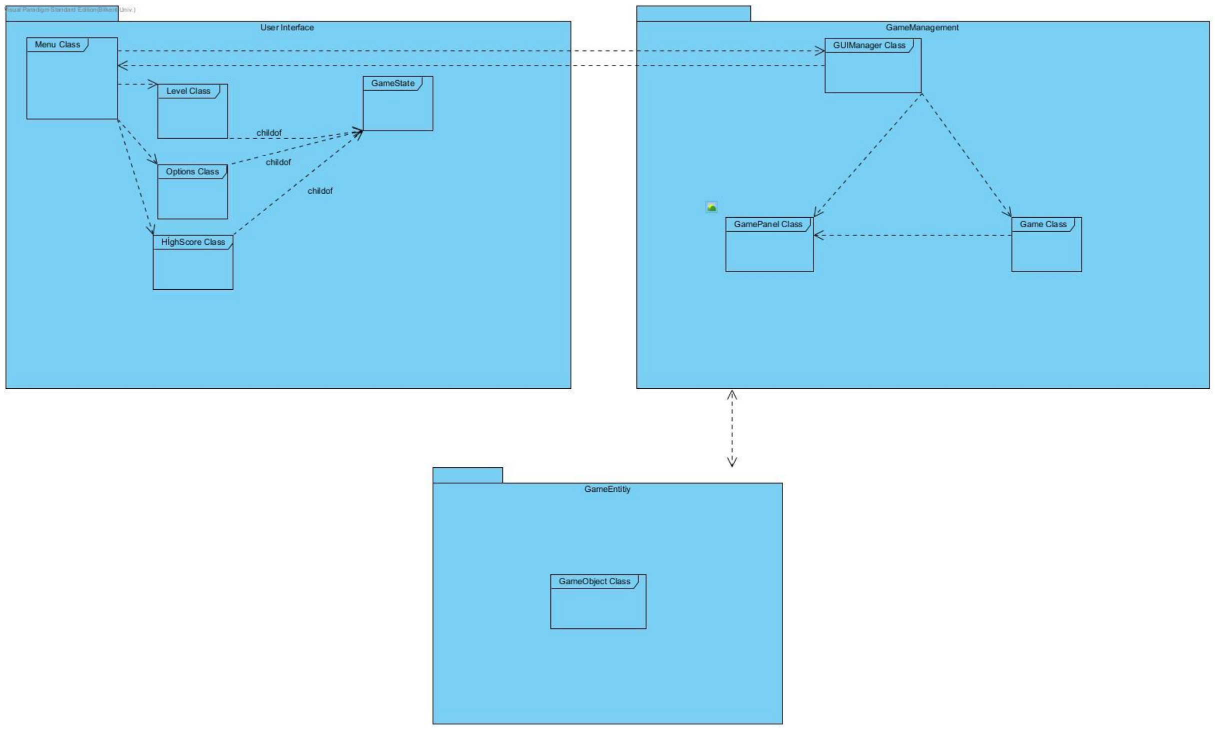


Figure 2 - Detailed Subsystem Decomposition

This section shows the system decomposition. The game system has different parts to show the organization. The decomposition helps to increase the game's fluency, like performance and efficiency while implementing the software. Also, decomposition helps us to meet non-functional requirements.

Figure -1 shows that the system has three different subsystems and Figure-2 shows them with more explanation. The subsystems are named: User Interface, Game Management and Game Entities. Each of them has different uses, but they are bounded to each other in a way to make the connection between the subsystems. Like its name, User Interface subsystem makes the first interaction between user and the game, and shows the crucial information, like the objects, remaining health and main menu. The other two subsystem are much bound each other, because some steps which are done in Game Management affects Game Entities.

2.2. Architectural Styles

2.2.1 Layers

Our system has decomposed into three layers, which are User Interface, Game Management and Game Entities. The User Interface layer has the highest hierarchy because it has the most importance for the user's interaction with the game and it is not bound to any layer. The second one, Game Management layer, is the one who controls the logic part of the game. The last layer, Game Entities, is the one who combines all necessary objects. The layer system uses closed architectural style, which means only layers who's above a layer can access the layer below.

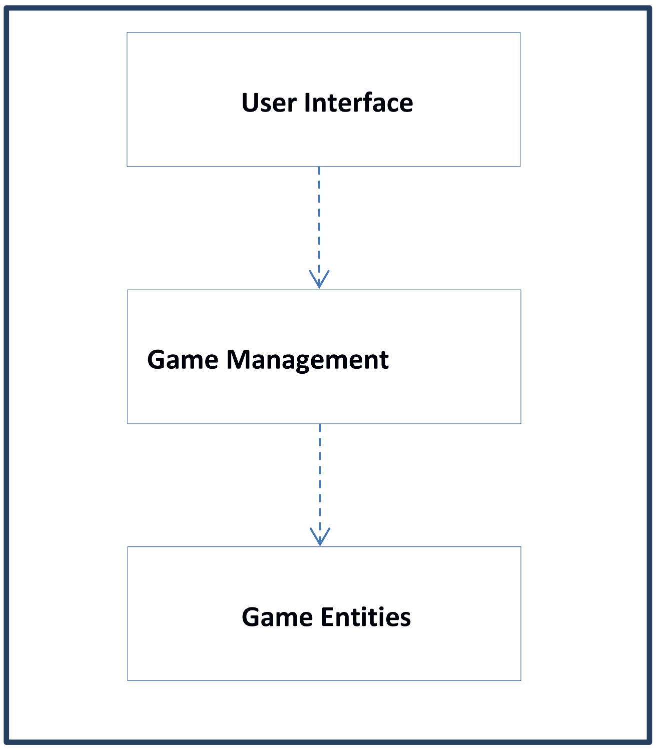


Figure 3 - Layers of System

2.3. Hardware / Software Mapping

Since our game is developed using Java programming language, it will require Java Runtime Environment (JRE) to work. Briefly, a decent computer in today's standards will be enough in order to play the game, because game will not have such complex graphics that force computers.

As hardware, it will require a mouse and a keyboard. In menu, user will need both but when the game is started, user will not need a mouse in order to play. Mouse will have no effect after the game is started.

2.4. Persistent Data Management

Firstly, the game will not use any database to store data. All game data will be stored in the client’s hard disk, because these data are need to be accessed in real time. High scores and the corresponding names will be stored in a text file to show the user when “High Scores” is selected from “Main Menu”. This text file will be updated automatically when there is a new high score and it is saved. Also, level files will be stored in text files in order to provide the user a choice to edit. The user will be able to edit level files, which are text files, using any text editor. Moreover, music files and images, such as background images or images of the object in the map, will be used and stored in hard drive.

2.5. Access Control and Security

There will be no need for user authentication system since the game will not require any network related functions and multiplayer feature will be used at the same computer. So, user authentication system for the game will be useless. Moreover, there will be no security issues about the game.

2.6. Boundary Conditions

Initialization: The game will not require any installation, because there will be an executable .jar file.

Termination: The game will be terminated when the user clicks the “X” from upper right. Also, the player can exit the current game to main menu via pause button.

Error: When there is an error about loading game files such as sounds or images, game will start anyway, for example, without sounds.

When there is an error about loading or finding level files which are text files, the game will not be able to start to game and will give an error.

3. Subsystem Services

3.1. Design Patterns

Façade Design Pattern: We used façade design pattern in two subsystems: Game Management and Game Entities. For Game Management, it has a relationship with User Interface subsystem with GUIManager class. For Game Entities, the relationship is with GameManagement subsystem and our façade class is Level which handles all changes and collusions that happens while in game and updates it with update() method.

3.2. User Interface Subsystem

User Interface Subsystem offers to the user the graphical components of the system. It has Menu, Options, HighScores and Level classes. These classes are child classes of GameState class. In general Menu class is used when the game is first run, Options class is in action when the user wants to change the settings and HighScores class is actively used by the Game Management Subsystem when the user chooses to see the HighScores. Level class is constructed when the player wants to start to play the game, after which at every map change the next level is constructed and contents drawn onto the screen. Below is the more detailed explanation of how the methods and the attributes of the User Interface Subsystem works. We should note that there is some incompatibility with the analysis report and the design report at hand regarding the Level class. In the first report we figured it would be easier to add new levels as a class when needed, therefore created seperate classes for different levels, yet realized there is a much more simple solution which is to read and create the levels from a text file, thus making it easier to add new levels to the game.

GameState Class

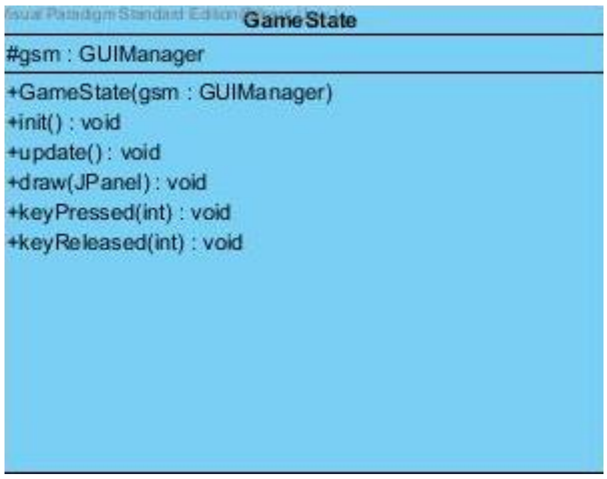
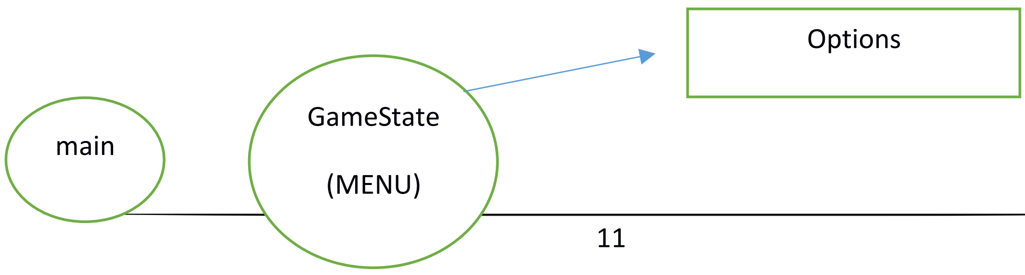


Figure 4 - GameState Class

You can think this class as state machine. Because the main purpose of this class is directing user. Some classes extend it: Menu, Options, High Scores. Also, because we will implement this game in java this GameState Class will extend JPanel. So, every class will be the JPanel. The methods will send to the child classes. Because in every child class have buttons there are keyPressed methods. You can think it as button listener.



For example, when the user clicks to the Options button, it will trigger the GameState(Menu) Class and it will change the panel, become the Options class(Panel). It will do this by using the Update method.

Attributes:

protected GUIManager guiManager: This attribute's function will be understood better in the following instances. To give a brief idea, it will be used to change the current visual. For instance if the user wants to start playing, this instances swapPanel(int) method will be called.

Constructor:

public GameState: It initializes the guiManager object.

Methods:

public abstract void init():

public abstract void update():

public abstract void draw(JPanel):

public abstract void keyPressed(int):

public abstract void keyReleased(int):

These functions will be explained in the children classes' overridden methods.

Menu Class

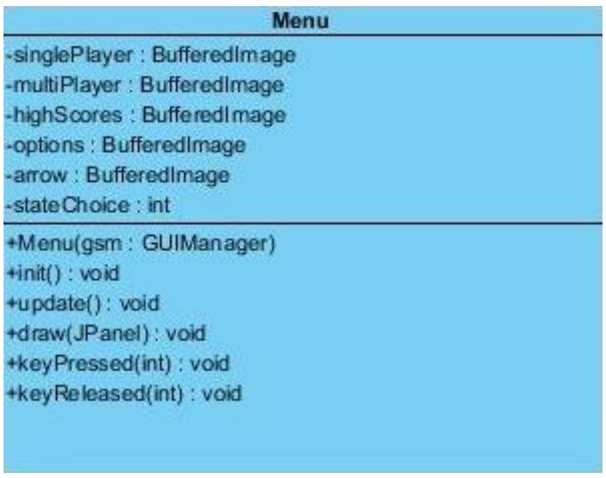


Figure 5 – Menu Class

Actually this class can be same with the GameState class because after user click game icon in the desktop, this class will open first. But we need GameState Class because of the other Panel classes. Also, the difference of the Menu from GameState methods and attributes. Because Menu class include Play Game button it has to has some game constructors such as the singlePlayer mode constructor, MultiPlayer Game Mode constructor. Because user will trigger the buttons and after this Menu class work on creating the MAP of the level 1 or MutliPlayer Map, locating objects by communicating with the GUIManager to create the GamePanel.

Attributes:

private BufferedImage singlePlayer: This is the link to single player game mode.

private BufferedImage multiPlayer: This is the link to multiplayer game mode

private BufferedImage highScores: This is the link to High Scores game state.

private BufferedImage options: Upon choosing this, options game state will appear on the screen.

private BufferedImage arrow: This is the pointer that will help user to identify which of the game states they want to see next. It will reappear alongside the different buttons when the user uses keyboard arrow keys to change her choice.

private int stateChoice: This is the integer that is updated along with the arrow instance. It will tell the Game Management Subsystem which of the next GameState child the user wants to see.

Constructors:

public Menu: This initializes the protected guiManager attribute for the use of this class. Also calls the init method described below.

Methods:

Public void init(): Initializes the BufferedImage instances by reading from image files.

It also initializes the stateChoice to '0' which corresponds to the choice of Single Player game mode, which will direct user to Level class' content.

Public void update(): This method is NOT overridden, as the Menu interface does not need updates.

public void draw(JPanel g): This method draws the BufferedImage instances that are initiated by the init() method to a panel to be shown to the user.

Public void keyPressed(int): This method will determine the place of the arrow object to appear on the JPanel as well as the stateChoice attribute with respect to the control strokes by the user.

Public void keyReleased(int): This method is not overridden as well.

HighScore Class

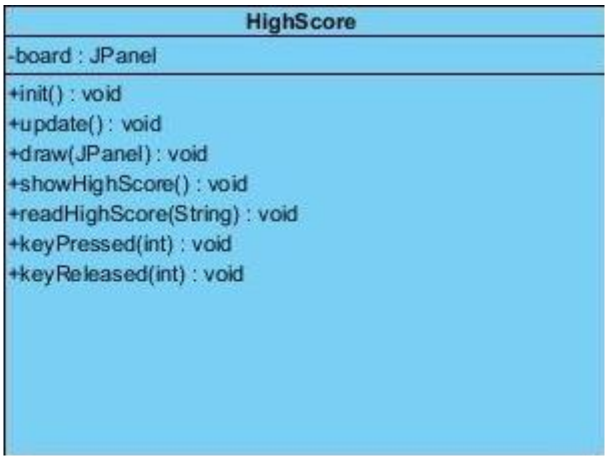


Figure 6 – HighScore Class

Attributes:

private JPanel board: This is the attribute on which the content of a text file that keeps track of high scores is written.

Constructors:

The constructor will use the following methods to read the high scores from a text file and prompt it to the user.

Methods:

Please note that unoverridden methods are not shown here. From now on the abstract methods in the GameState class will be shown only when they are overridden.

Public void showHighScore(): This will write the content of the loaded text file to the JPanel initiated above.

Public JPanel readHighScore(string): This method will take a string which is the location to the text file containing high score information, and will return a panel to be assigned and shown to the user.

Public void keyPressed(int): This method will redirect user to the main menu, when the backspace key is pressed.

Level Class

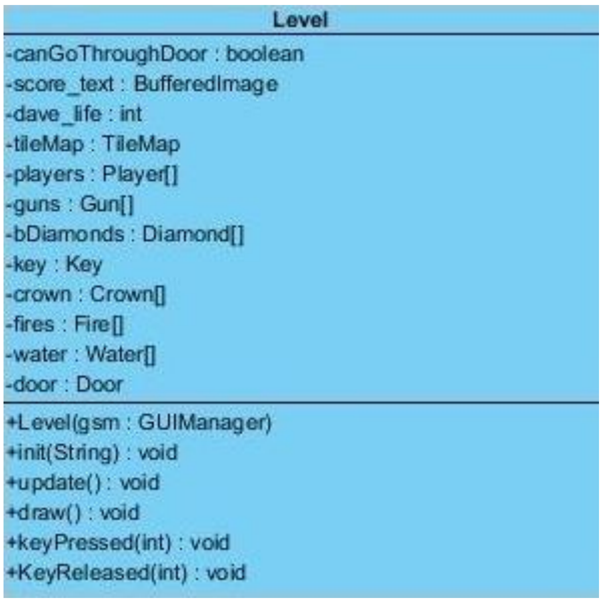


Figure 7 – Level Class

Attributes:

private boolean canGoThroughDoor: A boolean variable to denote whether the user has taken the key to open the door.

private BufferedImage score_text: This attribute will show the user current score she has.

private int dave_life: This variable will hold information about how many lives the Dave has.

private TileMap tileMap: This attribute's meaning will be understood better when we introduce the TileMap class. In short, TileMap class will hold a rectangular array to put different MapObject instances onto specific locations in a level. It will enable us to seperate colliding and non-colliding elements within the game.

Please note that the classes introduced from now on are children classes of MapObject abstract class, whose introduction is in the Game Entities Subsystem.

private Player[] players: This array, depending on the mode of the game, will hold the information about the size, location and speed of the Dave objects.

private Gun[] guns: This will denote the Gun objects on the map.

private Diamond[] bDiamonds: This class represents the diamond objects on the panel.

private Key key: This object is mandatory to be collected in order for the user to open the door.

private Crown[] crown: This object is again referenced in the prior reports as goods that can be collected by the Dave objects to increase the score.

private Fire[] fires: This fire class is an obstacle, upon interacting closely will vanish one life point from the Dave.

private Water[] water: This has the same function as the Fire class but the visual representations are different.

private Door door: Finally, a door object placed on the map to allow user to finish the level.

Constructor:

Constructor takes as an argument a GUIManager object and initializes its protected guiManager attribute to the given argument. It also calls the init() function described below.

Methods:

public void init(String):

This method takes a string to a text file that describes the level at hand in a space seperated file. And tiles the objects with respect to the number it reads from the text file. For instance if it read '5' from the text file, it will tile a fire object to the TileMap object. The images representing the MapObject objects will be predefined, and it will load them from those image files.

public void update():

This method will be called multiple times in a second. The call rate of the function will be imposed by the Game Management Subsystem. Everytime the Game Mangement Subsystem updates the game state this method will be called and the positions, number of lives and other game state informations held in the MapObject children classes will be updated alongside with the tileMap attribute.

public void draw(): After it initializes the MapObject attributes, first, this will draw them to the JPanel object of GUIManager class. And from there it will be shown to the user in the Game class which contains the main method. On the other hand, this method will be called everytime a call to the update method is done.

public void keyPressed(int): This method will listen the events from the keyboard and update the game state variables accordingly. The reason there is a distinction between pressing and releasing a key is that, the two have different meanings in the game. For instance, when one of the arrow keys is pressed, the motion will take place, however, when released it will be stopped.

public void keyReleased(int): When the pressed key is released the player object's attributes are changed and made immutable until the user presses another key.

3.3. Game Management Subsystem

In this subsystem we have a total of three classes. These classes control other classes by propagating a call that will trigger a change in the whole system. This system also controls how often the update is made, the size of the frame, which of the game states we are currently in and so forth. The classes in this subsystem are as follows: Game, GUIManager and GamePanel.

Game Class

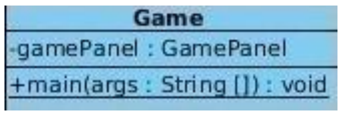


Figure 8 – Game Class

This class has the main method. It will create an instance of GamePanel object.

Attributes:

private GamePanel gamePanel: This object has all the necessary methods to control the game. See GamePanel class for more information.

Methods:

public static void main(String[]): This method will create a JFrame object as well as GamePanel object. It adds the GamePanel object to the main frame,

GamePanel Class

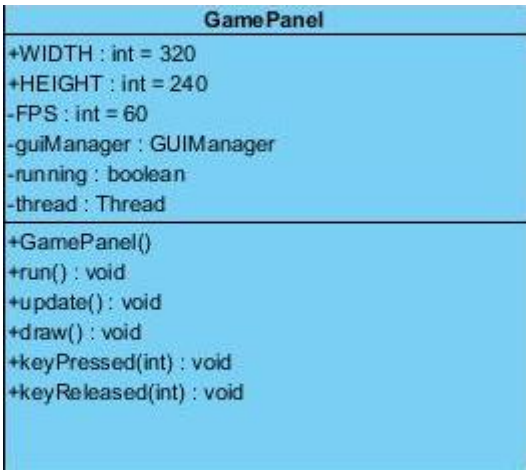


Figure 9 – GamePanel Class

This class is not involved with the low level details of the system. It just asks from its attributes to update its state in a very abstract way. It will catch KeyEvent objects, then send necessary commands to its attributes.

Attributes:

public static final int WIDTH: The width of the game window.

public static final int HEIGHT: The height of the game window.

private int FPS: This variable holds how many time at a second the update() function is propagated within the system.

private GUIManager guiManager: This attribute is a bridge between the management subsystem classes and the user interface elements.

private boolean running: This will hold the information regarding the status of the game.

private Thread thread: The program will have only one Thread object. It will allow us to manage the timing and starting or cessation of the system.

Methods:

public void run(): This method is run throughout the duration of the game. It has a while loop inside, that updates and draws the system FPS many times in a second. To accommodate timing, the thread is sometimes put to waiting.

public void update(): This method as well as the methods following this method do not add any more complexity to the system. These methods just call the guiManager objects update() method. guiManager then transmits this call to its GameObject attributes.

public void draw(): Calls the draw() method of the guiManager attribute.

public void keyPressed(int): Issues a call to the keyPressed(int) method of the guiManager object.

public void keyReleased(int): Issues a call to the keyReleased(int) method of the guiManager object.

GUIManager Class

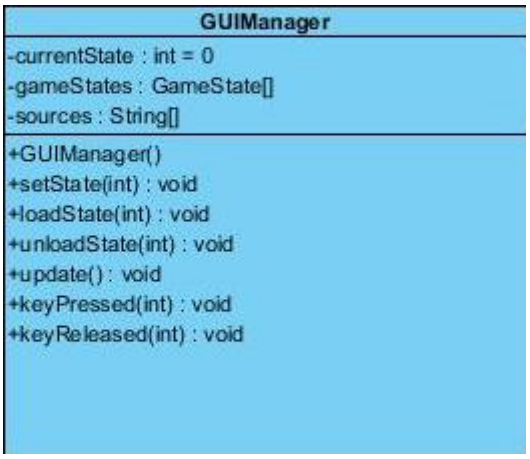


Figure 10 – GUIManager Class

This class serves as a bridge between the User Interface Subsystem and the Game Management Subsystem. It controls which of the views is shown to the user, from which text file that describes a certain map is read and many other core functionalities.

Attributes:

private int currentState: This holds the current state information. There is a one to one correspondence between the ID of the state and the order in the sources string array.

private GameState[] gameStates: This array holds the game states, such as Menu, Level, HıghScores. Generic array type allows us to call the abstract functions, thus making the management easier.

private String[] sources: When the user chooses a certain state to display, from this source file the content of the state is created. It is particularly useful for Level objects, as the architecture of the map is described in a space seperated text file. This also allows the experienced user to create new levels without adding new classes or changing any of the code.

Contructor:

The constructor sets the current state to '0' and creates the sources[] array by reading the names of the text files.

Methods:

public void setState(int): This function takes an integer specifying which GameState object load and draw. It first unloads the current state, then loads the state that is given as an argument and then draws it.

public void loadState(int): This method takes an integer that specifies the index of the name of the text file inside sources array. Than creates a new GameState object and assigns it to its corresponding index in the gameStates array.

public void unloadState(int): This basically sets null the index of the gameStates array given by the argument.

public void draw(): This function does not add extra complexity to the program, just calls the gameState objects' draw function after every update.

public void update(): Just like draw() function it also calls the update function of the GameState objects.

public void keyPressed(int): Same as the last the methods, transmits the call from GamePanel object to the GameState objects' overridden keyPressed() methods.

public void keyReleased(int): Same as above.

3.4. Game Entities Subsystem

GameObject Class

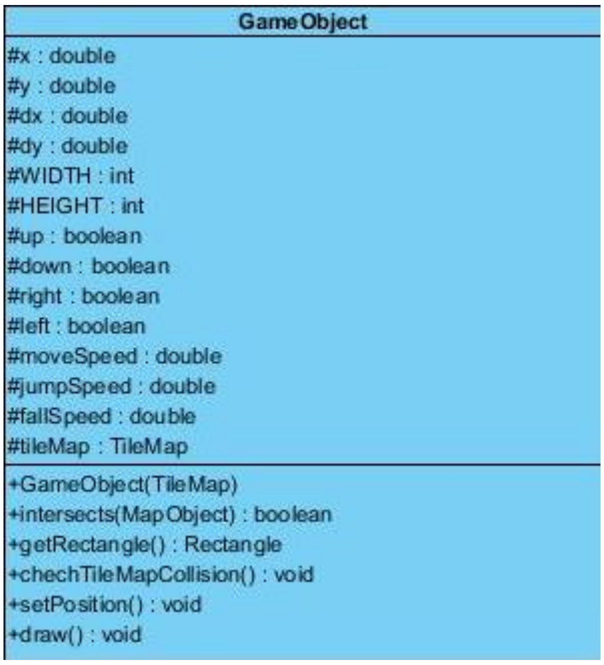


Figure 11 – GameObject Class

This is an abstract class, all the objects that appear on the screen, ranging from the bricks, fire, water, gun, key, door, Dave to other goods, are inherited from this class.

Therefore, the classes extended from this class are omitted in this report. A list of children classes as follows: Water, Fire, Key, Bullet, Gun, Door, Player, Diamond, Crown, Vine.

Attributes:

protected double x: This variable holds the horizontal position.

protected double y: This variable holds the vertical position of the object.

protected double dx: This variable is used when there is change in the coordinates of the object, in case the Dave moves or the Map slides.

protected double dy: The same as above, except that it holds the vertical displacement.

protected int width, height: These variables specify the size of the object to be displayed on the JPanel.

protected boolean up, down, right, left: These boolean variables denote which direction the objects are facing. According to these variables

protected double moveSpeed, jumpSpeed, fallSpeed: These variables hold how much displacement occurs when pressed the control buttons in the horizontal and vertical directions respectively. One might also need to define the maximum speed when falling occurs for instance, but we address this problem during the coding phase.

protected TileMap tileMap: Explanation about the TileMap class will come later. In short, it has as its attribute Tile rectangular array that places, represents the placings of the objects. The reason it is a protected attribute is that, every GameObject children instances will be able to refer to the same TileMap object enabling us to check the collisions and resolve them later on.

Constructors:

Constructor takes a TileMap object and sets its tileMap attribute to it. As discussed above this eases our job when we try to address the collisions of the objects.

Methods:

public boolean intersects(MapObject): This function takes a MapObject object as an argument and by using the getRectangle() methods which return the bounding box of an object, it decides whether the two objects intersect.

public Rectangle getRectangle(): This will return a bounding rectangle around that object.

public void checkTileMapCollision(): This method gets the rectangle information using the above methods and check whether the Tile objects inside the TileMap[][] 2D array are colliding, benefiting from the information regarding whether an object is rigid or impervious. The Tile objects will have attributes denoting this useful information.

public void setPosition(): Upon getting instructions from the higher order classes keyPressed(int) method, the new position of the object is calculated and updated.

public void draw(): Updated position information is then applied onto the panel.

TileMap Class

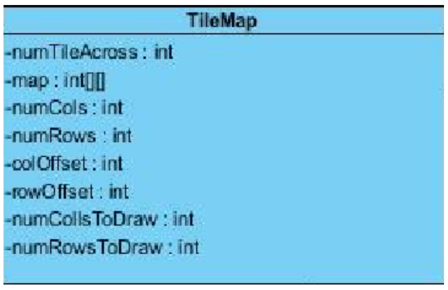


Figure 12 – TileMap Class

This is the important class of the system. Because user will use this class to much while playing the game. We will use tile map organization. map[][] array will show the organization of the colons and rows in terms of tiles by using ones and zeros.

```
Ex:
int map = [
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
  [1,0,0,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
  [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,1],
  [1,0,0,1,1,0,0,0,1,1,1,1,1,0,0,0,0,1,1,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,0,0,0,0,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,1],
  [1,0,0,0,1,1,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,1],
  [1,0,1,1,1,1,1,0,0,0,0,0,1,0,0,1,1,1,1,0,1,1,1,1],
  [1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,1,1,1,1],
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
];
```

By using this array we will place the images of the game. numRowsToDraw, numColsToDraw variables are to detecting the how many repetition will be done by method that place tiles.

Tile Class

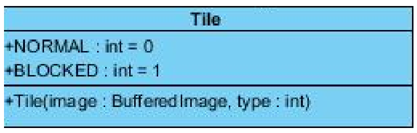
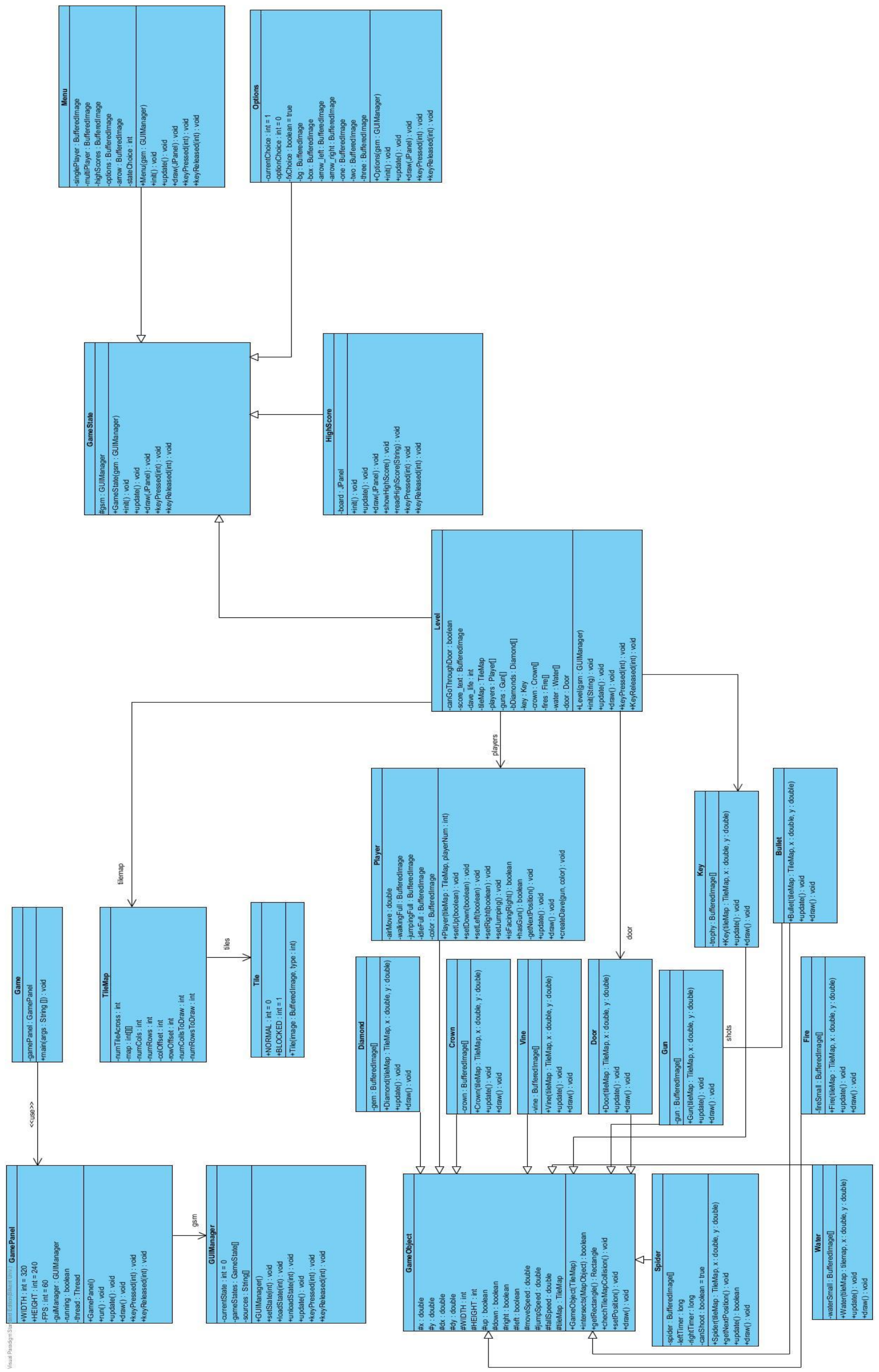


Figure 13 – Tile Class

This is the class that will understand the movements of dave and will provide information to the game panel to move or place dave in appropriate location. It will detect collusions and the location of the dave such as can it on any row or column etc.

3.5. Detailed Class Diagram



Menu
-singlePlayer : BufferedImage
-multiPlayer : BufferedImage
-highScores : BufferedImage
-options : BufferedImage
-arrow : BufferedImage
-stateChoice : int
+Menu(gsm : GUIManager)
+init() : void
+update() : void
+draw(JPanel) : void
+keyPressed(int) : void
+keyReleased(int) : void
Options
-currentChoice : int = 1
-optionChoice : int = 0
-fxChoice : boolean = true
-bg : BufferedImage
-box : BufferedImage
-arrow_left : BufferedImage
-arrow_right : BufferedImage
-one : BufferedImage
-two : BufferedImage
-three : BufferedImage
+Options(gsm : GUIManager)
+init() : void
+update() : void
+draw(JPanel) : void
+keyPressed(int) : void
+keyReleased(int) : void
GameState
#gsm : GUIManager
+GameState(gsm : GUIManager)
+init() : void
+update() : void
+draw(JPanel) : void
+keyPressed(int) : void
+keyReleased(int) : void
HighScore
-board : JPanel
+init() : void
+update() : void
+draw(JPanel) : void
+showHighScore() : void
+readHighScore(String) : void
+keyPressed(int) : void
+keyReleased(int) : void
Level
-canGoThroughDoor : boolean
-score_text : BufferedImage
-dave_life : int
-tileMap : TileMap
-players : Player[]
-guns : Gun[]
-bDiamonds : Diamond[]
-key : Key
-crown : Crown[]
-fires : Fire[]
-water : Water[]
-door : Door
+Level(gsm : GUIManager)
+init(String) : void
+update() : void
+draw() : void
+keyPressed(int) : void
+keyReleased(int) : void
players
tilemap
Player
-airMove : double
-walkingFull : BufferedImage
-jumpingFull : BufferedImage
-dieFull : BufferedImage
-color : BufferedImage
+Player(tileMap : TileMap, playerNum : int)
+setUp(boolean) : void
+setDown(boolean) : void
+setLeft(boolean) : void
+setRight(boolean) : void
+setJumping() : void
+isFacingRight() : boolean
+hasGun() : boolean
-getNextPosition() : void
+update() : void
+draw() : void
+createDave(gun, color) : void
Key
-trophy : BufferedImage[]
+Key(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
Bullet
+Bullet(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
Game
-gamePanel : GamePanel
+main(args : String []) : void
TileMap
-numTileAcross : int
-map : int[][]
-numCols : int
-numRows : int
-colOffset : int
-rowOffset : int
-numColsToDraw : int
-numRowsToDraw : int
tiles
Tile
+NORMAL : int = 0
+BLOCKED : int = 1
+Tile(image : BufferedImage, type : int)
Diamond
-gem : BufferedImage[]
+Diamond(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
Crown
-crown : BufferedImage[]
+Crown(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
Vine
-vine : BufferedImage[]
+Vine(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
door
Door
+Door(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
Gun
-gun : BufferedImage[]
+Gun(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
shots
Fire
-fireSmall : BufferedImage[]
+Fire(tileMap : TileMap, x : double, y : double)
+update() : void
+draw() : void
<<use>>
GamePanel
+WIDTH : int = 320
+HEIGHT : int = 240
-FPS : int = 60
-guiManager : GUIManager
-running : boolean
-thread : Thread
+GamePanel()
+run() : void
+update() : void
+draw() : void
+keyPressed(int) : void
+keyReleased(int) : void
gsm
GUIManager
-currentState : int = 0
-gameStates : GameState[]
-sources : String[]
+GUIManager()
+setState(int) : void
+loadState(int) : void
+unloadState(int) : void
+update() : void
+draw() : void
+keyPressed(int) : void
+keyReleased(int) : void
GameObject
#x : double
#y : double
#dx : double
#dy : double
#WIDTH : int
#HEIGHT : int
#up : boolean
#down : boolean
#left : boolean
#right : boolean
#moveSpeed : double
#jumpSpeed : double
#fallSpeed : double
#tileMap : TileMap
+GameObject(TileMap)
+intersects(MapObject) : boolean
+getRectangle() : Rectangle
+checkTileMapCollision() : void
+setPosition() : void
+draw() : void
Spider
-spider : BufferedImage[]
-leftTimer : long
-rightTimer : long
-canShoot : boolean = true
+Spider(tileMap : TileMap, x : double, y : double)
+getNextPosition() : void
+update() : boolean
+draw() : void
Water
-waterSmall : BufferedImage[]
+Water(tileMap : tilemap, x : double, y : double)
+update() : void
+draw() : void
Visual Paradigm Standard Edition