

Bilkent University

Engineering Faculty

Department of Computer Engineering

**SNAKE - THE EXTRAORDINARY ADVENTURE**

Analysis Report

CS 319

Object-Oriented Software Engineering

Course Instructor: Hüseyin Özgür TAN

Teaching Assistant: İstemi Bahçeci

Semester: 2015-2016 Fall

Section 2 - Group 5

Cihan ERYONUCU

Mehmet Çağrıbey GÜVENDİK

Serhat ÇETİNKAYA

Zülal BİNGÖL

**CONTENTS**

1. Introduction..................................................................................................................5
2. Requirement Analysis...................................................................................................6
   1. Overview................................................................................................................6
   2. Functional Requirements.......................................................................................7
   3. Non-Functional Requirements...............................................................................9
   4. Constraints............................................................................................................10
   5. Scenarios...............................................................................................................11
   6. Use Cases and Game's Use Case Model .............................................................14
   7. User Interface.......................................................................................................21
3. Analysis........................................................................................................................24
   1. Object Model........................................................................................................24
      1. Domain Lexicon.........................................................................................24
      2. Class Diagrams...........................................................................................26
   2. Dynamic Models...................................................................................................27
      1. State Chart Diagrams..................................................................................27
      2. Sequence Diagrams.....................................................................................29
4. Design………………………………………………………………………………...34
   1. Design Goals.........................................................................................................34
   2. Subsystem Decomposition………………………………………………………37
   3. Architectural Design.............................................................................................38
   4. Hardware-Software Mapping...............................................................................39
   5. Addressing Key Concerns....................................................................................39
      1. Persistent Data Management......................................................................39
      2. Access Control and Security......................................................................40
      3. Global Software Control........................................................................40
      4. Boundary Conditions.............................................................................40
5. Object Design..........................................................................................................43
   1. Pattern Applications.........................................................................................43
   2. Class Interfaces.................................................................................................46
   3. Specifying Contracts.........................................................................................62
6. Conclusion and Lessons Learned.............................................................................66

Softwares Used........................................................................................................67

**TABLE OF FIGURES**

[Figure 1: Use Case Model of Game 21](#_Toc437198004)

[Figure 2: Welcoming Panel Proposed User Interface 22](#_Toc437198005)

[Figure 3: Help Panel Proposed User Interface 22](#_Toc437198006)

[Figure 4: Credits Panel Proposed User Interface 23](#_Toc437198007)

[Figure 5: Game Panel Proposed User Interface 23](#_Toc437198008)

[Figure 6: Proposed Tokens for Baits in Game Play 24](#_Toc437198009)

[Figure 7: Class Diagram of Class Hierarchy 27](#_Toc437198010)

[Figure 8: Snake's BonusPointBait State Chart Diagram 28](#_Toc437198011)

[Figure 9: Snake's SuperSnakeBait State Chart Diagram 28](#_Toc437198012)

[Figure 10: Snake's State Chart Diagram for Death Scenarios 29](#_Toc437198013)

[Figure 11: ExtendingBait Sequence Diagram 30](#_Toc437198014)

[Figure 12: PoisonousBait Sequence Diagram 30](#_Toc437198015)

[Figure 13: ShorteningBait Sequence Diagram 31](#_Toc437198016)

[Figure 14: SuperSnakeBait Sequence Diagram 31](#_Toc437198017)

[Figure 15: UpsideDownBait Sequence Diagram 32](#_Toc437198018)

[Figure 16: BonusPointBait Sequence Diagram 32](#_Toc437198019)

[Figure 17: RegularBait Sequence Diagram 33](#_Toc437198020)

[Figure 18: Regular Movement Sequence Diagram 33](#_Toc437198021)

[Figure 19: Sequence Diagram for Panel Transitions 34](#_Toc437198022)

[Figure 20: Subsystem Decomposition 38](#_Toc437198023)

[Figure 21: Opaque Layering Architecture of the Game 39](#_Toc437198024)

[Figure 22: Hardware Mapping Deployment Diagram 40](#_Toc437198025)

[Figure 23 : Façade Pattern 43](file:///C:\Users\user\Desktop\ANALYSIS_DESIGN_REPORT_GROUP_5%20(1).docx#_Toc437198026)

[Figure 24: Observer Pattern 44](#_Toc437198027)

[Figure 25: Command Pattern 45](#_Toc437198028)

[Figure 26: GameManagement Detailed Class Definitions 46](#_Toc437198029)

[Figure 27: User Interface Sytem Class Hierarchy 47](#_Toc437198030)

[Figure 28 : GameManager Class 47](#_Toc437198031)

[Figure 29: Snake Class 48](#_Toc437198032)

[Figure 30: Arena Class 50](#_Toc437198033)

[Figure 31: Bait Class 51](#_Toc437198034)

[Figure 32: Representative Class Diagram for All of the baits 51](#_Toc437198035)

[Figure 33: BonusPoint Bait Class 52](#_Toc437198036)

[Figure 34: SuperSnake Bait Class 54](#_Toc437198037)

[Figure 35: Piece Class 55](#_Toc437198038)

[Figure 36: HelpPanel Class 56](#_Toc437198039)

[Figure 37: CreditsPanel Class 57](#_Toc437198040)

[Figure 38: GameOverPanel Class 57](#_Toc437198041)

[Figure 39: GamePanel Class 58](#_Toc437198042)

[Figure 40: GeneralPanel Class 59](#_Toc437198043)

[Figure 41: GAME\_GENERAL (driver of the game) 60](#_Toc437198044)

**1. INTRODUCTION**

We will implement a game called Snake - The Extraordinary Adventure. There are a lot of snake games with common scheme. In well-known snake game there is snake and the baits. User controls snake which is supposed to eat baits. With eating each bait, snakes’ length gets longer. If snakes’ head touches any part of its own, snake dies. Our game is inspired by Snake game but with different aspects, various events and more fun. Therefore, our Snake game will be different and difficult than any others.

Instead of just normal baits, there will be six different baits with different colors. Each bait has different effects which are dying, extending, bonus point, swapping head and tail, invulnerability to death. Hence, variations of baits will affect the entire gameplay significantly. Since every bait has different effects, users’ strategy should be depend on deciding what bait should be eaten for snakes’ length. In addition, scoreboard will be available in game. After snake dies, user will be able to see his/her score.

There will be also walls which appear in the game. Shape of the walls will be straight. Walls will complicate snakes’ movement and also affect game play radically. With combination of various baits and walls, game complexity and difficulty will be much higher than regular Snake game.

Our game will be implemented in Java programming language. We will avoid any operating system specific dependencies to be able to run our game in every platform with a Java virtual machine.

**2. REQUIREMENT ANALYSIS**

**2. 1. OVERVIEW**

At the base, the philosophy of the game is the same with all its ancestors: there is a snake which tries to eat the treats as much as possible and grows without biting himself. However, in this game, it is planned to add different treats that include shortening, poisonous, super snake, bonus point making, extending, upside down. The snake will follow various types of behaviors corresponding to the baits. The effects of the different baits on the snake are explained below;

**Regular Bait**: Snake follows usual behaviors; score gain is the regular point.

**Shortening Bait:** Snake's length decreases.

**Poisonous Bait:** Snake dies, game finishes.

**SuperSnake Bait**: The snake eats the baits within a given period of time without dying.

**BonusPoint Bait:** Within a period of time, the snake scores 5 times more points with each bait it eats.

**Extending Bait:** Snake's length increases.

**UpsideDown Bait:** Snake's head and tail change places, it starts move upside down.

The borders of the game panels will act as barriers for snake's movements. In regular state, the snake's life will end when its head crashes the walls.

**2. 2. FUNCTIONAL REQUIREMENTS**

**2.2.1. Playing the Game (Snake)**

When the player starts the game, the system initializes a game based on Snake’ rules.

The game ends when the player quits or death condition is satisfied.

**2.2.2. Controlling Snake**

In the game, the player will control the movements of the Snake by using left and right buttons in the keyboard.

**2.2.3. Displaying Help**

Information about the game play, names of different colored baits and their function will be included in help section; the player can access the help section.

Help section consists of two sections:

• Control of Snake

• Information about entire game

With Displaying Help feature, the player will be able to read the game rules, learn the controls and learn about scoring and ending conditions of the game.

**2.2.4. Exiting the Game**

Game Over state of the game is reached when the player crashes the borders of the arena or eats itself by controlling the Snake with left and right buttons of the keyboard. In Game Over state, the score of the player is shown on the game screen.

**2.2.5 Exiting from Application**

The user of the game can exit from the application by clicking on the cross sign at right upper corner.

**2.3. NON-FUNCTIONAL REQUIREMENTS**

**2.3.1. Graphics**

The graphics of the game will provide a fun game play. There will not be flickering and other graphical problems to create a smooth gaming.

**2.3.2. Controls and Response Time**

The response time of the game will be reduced to a minimum level in order to provide the players a smooth controlling.

**2.3.3. Capacity**

The game supports only one player at a time. The proposed system has not a multiplayer base; therefore one player can play the game.

**2.3.4. Availability**

The target audience of the proposed game is not be specified, so all of the age groups will have access to the game if the game is published publicly.

**2.3.5. Easy to Learn**

The game will be played by only two buttons from the keyboard: right and left buttons, so the game will be easy to play and user friendly.

**2.4. CONSTRAINTS**

**2.4.1. Implementation Language**

The implementation language of the Snake will be Java.

**2.4.2. Application Platform**

The program will be desktop application.

**2.4.3. Operating System Platforms**

Java Swing platform is planned to use in the GUI of the game.

**2.5. SCENARIOS**

1. **Regular Movement Scenario (MoveTheSnake):** Game is running normally. Let’s assume snake is moving to upwards. Player presses the right button. Snake turn his face to its right and continue its path. After a while Player presses left button this time, Snake faces its left and continues there. Player uses right and left buttons to direct the Snake.
2. **Eating Bait Scenarios (EatBait):** Player makes the Snake to move to the bait. When the Snake’s head finally meets with bait, Snake eats the bait and bait disappears. Bait affects the Snake according to its type.
   1. **Regular Bait Scenario (EatRegularBait)**

If the Snake encounters the regularly spawned black bait, Snake will grow longer and Player gains normal points.

* 1. **Shortening Bait Scenario (EatShorteningBait)**

While Eating Baits scenario is active there are regular baits and there are special baits with each has different color. When player eats purple colored special bait which is called “shortening”, the length of the snake is reduced by 5.

* 1. **Bonus Point Bait Scenario (EatBonusPointBait)**

While Eating Baits scenario is active there are regular baits and there are

special baits with each has different color. When player eats a cyan colored

special bait which is called “bonus point”, number of seconds appears on the

screen for countdown. Until countdown finishes each eaten bait makes 5 times

points of the normal bait.

* 1. **Poisonous Bait Scenario (EatPoisonousBait)**

While Eating Baits scenario is active there are regular baits and there are

special baits with each has different color. When player eats a green colored

special bait which is called “poisonous”, it causes a death scenario which is the

end of the game.

* 1. **Extending Bait Scenario (EatExtendingBait)**

In this scenario, there are normal baits and there are special baits with each has different color. If snake eats the special bait with blue color which is called “extending bait”, its length increases by 5 points which is muk.

* 1. **Upside Down Bait Scenario (EatUpsideDownBait)**

Snake has a tail and a head. This scenario includes the special bait which is ‘upside-down bait’. If snake eats the yellow bait which is ‘upside-down bait’, its head will become its tail and its tail will become its head. Therefore its head and tail will be swapped and direction of snake will be the way which former tail has directed.

* 1. **SuperSnake Bait Scenario (EatSuperSnakeBait)**

This scenario has the special bait which is ‘Super Snake Bait’ with red colored bait. If snake eats this red colored bait, it will become invulnerable any deathful events like tossing a wall, eating poisonous bait. However, SuperSnake bait does not prevent self destruction if snake eats himself.

**3. Death Scenarios**

When the following scenarios occur, the Snake dies and the Game is over. The TotalScore is calculated by the latest score of the Snake.

* 1. **Self-Destruction Scenario (SelfDestruct)**

The Snake expresses its regular movements in the game. Player controls its movements by the buttons. When the control buttons causes Snake’s head to touch its body, it dies. Game finishes.

* 1. **Crashing the Wall Scenario (CrashBorder)**

The Snake expresses his regular movements. Player controls its movements by Left & Right buttons. When the Snake does not turn and its head reaches the borders of the game or when the Snake turns to the corresponding direction while moving near the borders; Snake’s head crashes the borders and it dies. Game finishes.

* 1. **Poisonous Bait Scenario (EatPoisonousBait)**

The Snake expresses its regular movements in the game. EatBait scenario is active in the game. Green bait (poisonous) is on the game screen. It eats the green bait. It dies. Game finishes.

**2.6. USE-CASE MODELS**

**Use Case Model for Regular Movement:**

**Name**: MoveTheSnake

**Actors**: Player

**Entry condition:** None

**Flow of Events:** Player presses the left or right button**.** Snake turn his heads according to the pressed button

**Exit condition:** Turning process is completed

**Exceptions:** None

**Special Requirements:** None

**Use Case Model for Eating Regular Bait:**

**Name**: EatRegularBait

**Actors**: Player

**Entry condition:** Snake’s head is over the black Regular bait. In other words, Snake eats the black bait.

**Flow of Events:** Snake's length will increase in normal state. Normal points will be given.

**Exit condition**: One move after eating the bait.

**Exceptions:** None

**Special Requirements:** None

**Use Case Model for Eating Shortening Bait:**

**Name:** EatShorteningBait

**Actors:** Player

**Entry Condition:** Snake’s head touches the purple bait.

**Flow of Events:** The Snake moves regularly on the screen, then snake approaches to the purple bait.

**Exit Condition:** One move after from touching the bait The Snake’s length is reduced by 5.

**Exceptions:** If the length of The Snake is less than (starting length + then the effect of the shortening bait is ignored. If the snake has eaten “SuperSnakeBait” before, then the effect of the shortening bait is ignored.

**Special Requirements:** None

**Use Case Model for Eating Bonus Point Bait:**

**Name:** EatBonusPointBait

**Actors:** Player

**Entry Condition:** Snake’s head touches the cyan bait.

**Flow of Events:** The snake moves regularly on the screen, then snake approaches to the cyan bait and eats it.

**Exit Condition:** Number of seconds appears on the screen for countdown.

**Exceptions:** Until countdown finishes, the gain of every bait is raised 5 times.

**Special Requirements:** None

**Use Case Model for Eating Poisonous Bait:**

**Name:** EatPoisonousBait

**Actors:** Player

**Entry Condition:** Snake’s head touches the green bait.

**Flow of Events:** The Snake moves regularly on the screen. Then Snake’s head approaches to the green bait and touches. The snake dies. Game is over. Total Score is calculated and printed on the screen.

**Exit Condition:** The Snake dies.

**Exceptions:** If the snake has eaten “super snake bait” before, then the effect of the shortening bait is ignored.

**Special Requirements:** None

**Use Case Model for Eating Extending Bait:**

**Name:** EatExtendingBait

**Actors:** Player

**Entry Condition:** Snake eats the blue bait which is extending bait

**Flow of Events:** If player eats the blue bait, snake’s length will be extended 5 points more compared to regular bait

**Exit Condition:** One move after the eating extending bait

**Exception:** None

**Special Requirements:** None

**Use Case Model for Eating Upside Down Bait:**

**Name:** EatUpsideDownBait

**Actors:** Player

**Entry Condition:** Snake eats the yellow bait which is upside-down bait

**Flow of Events:** If player eats the yellow bait, snake’s tail and its head will be replaced. After that, direction of snake will be direction of former tail.

**Exit Condition:** One move after the eating upside-down bait

**Exception:** None

**Special Requirements:** None

**Use Case Model for Eating SuperSnake Bait:**

**Name:** EatSuperSnakeBait

**Actors:** Player

**Entry Condition:** Snake eats the red bait which is SuperSnakeBait.

**Flow of Events:** While playing the game, if snake eats the red bait which is ‘SuperSnakeBait’.

**Exit Condition:** After ten second, snake will be no more invulnerable or against facing self destruction.

**Exception:** None

**Special Requirements:** None

**Use Case Model for Self Destruction:**

**Name:** SelfDestructSnake

**Actors:** Player

**Entry Condition:** Snake’s head touches its body.

**Flow of Events:** The Snake moves regularly on the game screen. The Player makes the Snake’s head touch its body by controlling it with the left and right buttons of the keyboard. The Snake dies, game is over. TotalScore is calculated by the latest score of the Snake. TotalScore is printed on the screen.

**Exit Condition:**  the Snake dies.

**Exceptions:** None

**Special Requirements:** None

**Use Case Model for Crashing Border:**

**Name:** CrashBorder

**Actors:** Player

**Entry Condition:** Snake’s headtouches the border.

**Flow of Events:** The Snake moves regularly on the game screen. The Player makes the Snake’s head crush to the borders of the game screen by controlling it with the left and right buttons of the keyboard. The Snake dies, game is over. TotalScore is calculated by the latest score of the Snake. TotalScore is printed on the screen.

**Exit Condition:** The Snake dies.

**Exceptions:** When the Snake eats the red bait (SuperSnake), it enters into SuperSnakeBait Mode. In this mode, the Snake cannot die.

**Special Requirements:** None

**Use Case Model for Poisonous Bait:**

**Name:** EatPoisonousBait

**Actors:** Player

**Entry Condition:** Snake’s head touches green bait.

**Flow of Events:** The Snake moves regularly on the game screen. Snake’s head touches the green bait. The Snake dies, game is over. TotalScore is calculated by the latest score of the Snake. TotalScore is printed on the screen.

**Exit Condition:** The Snake dies.

**Exceptions:** When the Snake eats the red bait (SuperSnake), it enters into SuperSnakeBait Mode. In this mode, the Snake cannot die.

**Special Requirements:** None

**Use Case Model for the Game:**

The following figure indicates the use case model for the game.

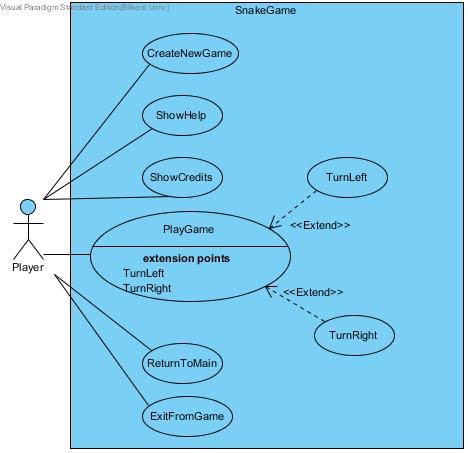


Figure 1: Use Case Model of Game

**2.7. USER INTERFACE**

The following figures represent the proposed user interface of the game, Snake - The Extraordinary Adventure.

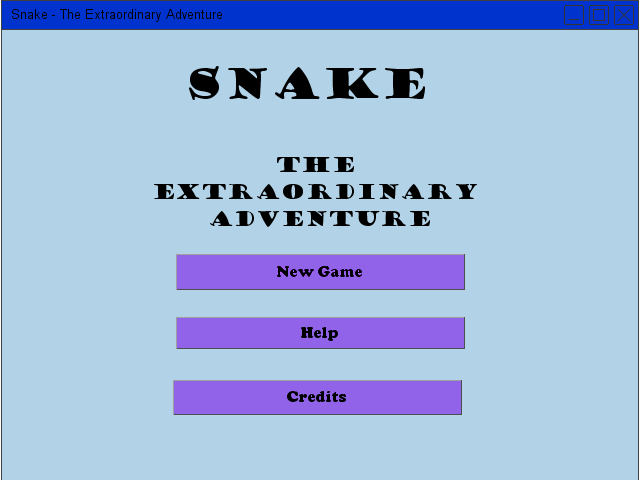
****

Figure 2: Welcoming Panel Proposed User Interface

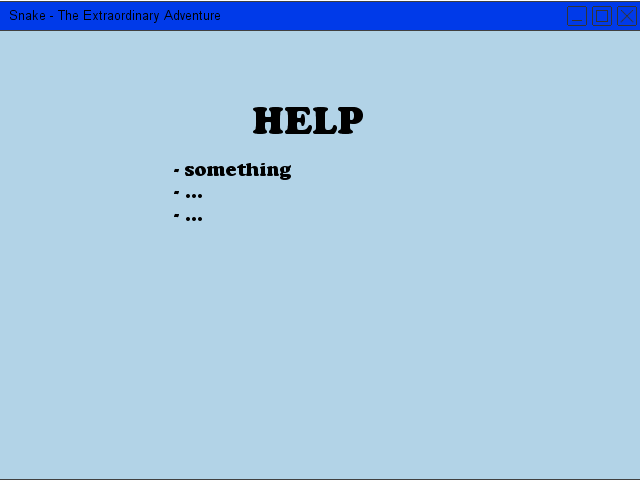


Figure 3: Help Panel Proposed User Interface

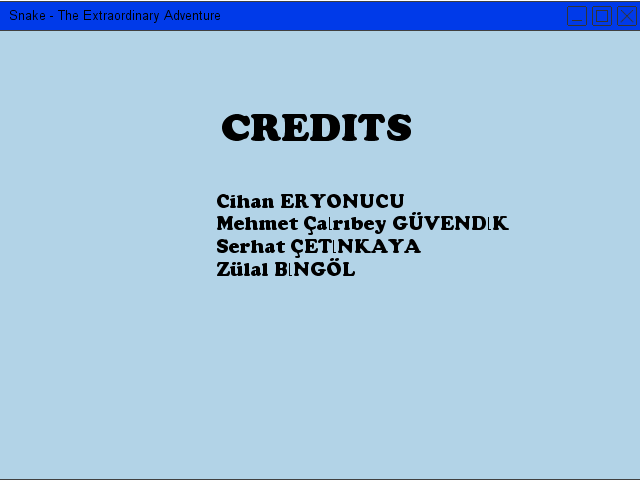
****

Figure 4: Credits Panel Proposed User Interface

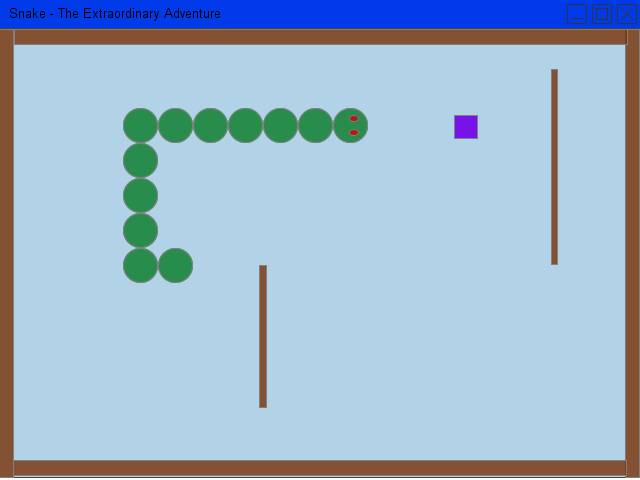
****

Figure 5: Game Panel Proposed User Interface

Here are the colors of the baits which will be used in the user interface;

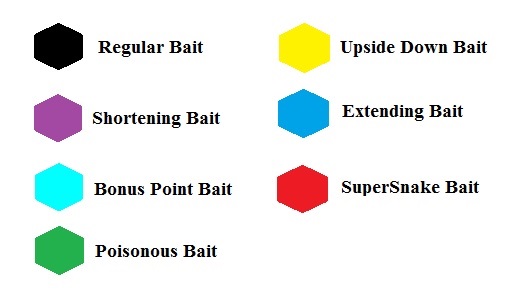


Figure 6: Proposed Tokens for Baits in Game Play

**3. ANALYSIS**

**3.1. OBJECT MODEL**

**3.1.1. DOMAIN LEXICON**

**Piece Class:** Piece class is one of the most important classes of the system. It is used in the body of the snake and the arena. Arena’s wall will be Piece’s. In addition snake will be consisting of Piece’s. They are both pieces because when there is collision of the piece’s snake will die whether it is because of the self-collision or collision between wall and snake.

**Arena Class:** Arena is basically the board that game will run.

**Snake Class:** Snake is consisting of Piece’s. It has the crucial methods in it such as die, eatBait and movement. It can be said that it is the core functional class of the game.

**Bait:** Bait is the food that the snake will try to eat the baits. This class is abstract it will be foundation for the baits.

**UpsideDown Bait class:** This bait will change the position of the head and tail. Score will be given normally.

**BonusPointBait Class:** This bait will give the player a 15 second time for the 5 times bonus point.

**RegularBait Class:** This will extend the snake’s length and give regular point.

**ShorteningBait Class:** This will shorten the snake’s length. Score will be given normally.

**SuperSnakeBait Class:** This will give the snake invulnerability to the wall deaths for 10 seconds. Score will be given normally.

**PoisonousBait Class:** This bait will kill the snake and end the game.

**ExtendingBait Class:** This extends the snake’s length twice as much compared to the normal growth. Normal range of points will be given.

**3.1.2 CLASS DIAGRAMS**

The following diagram shows the class hierarchy in the game.

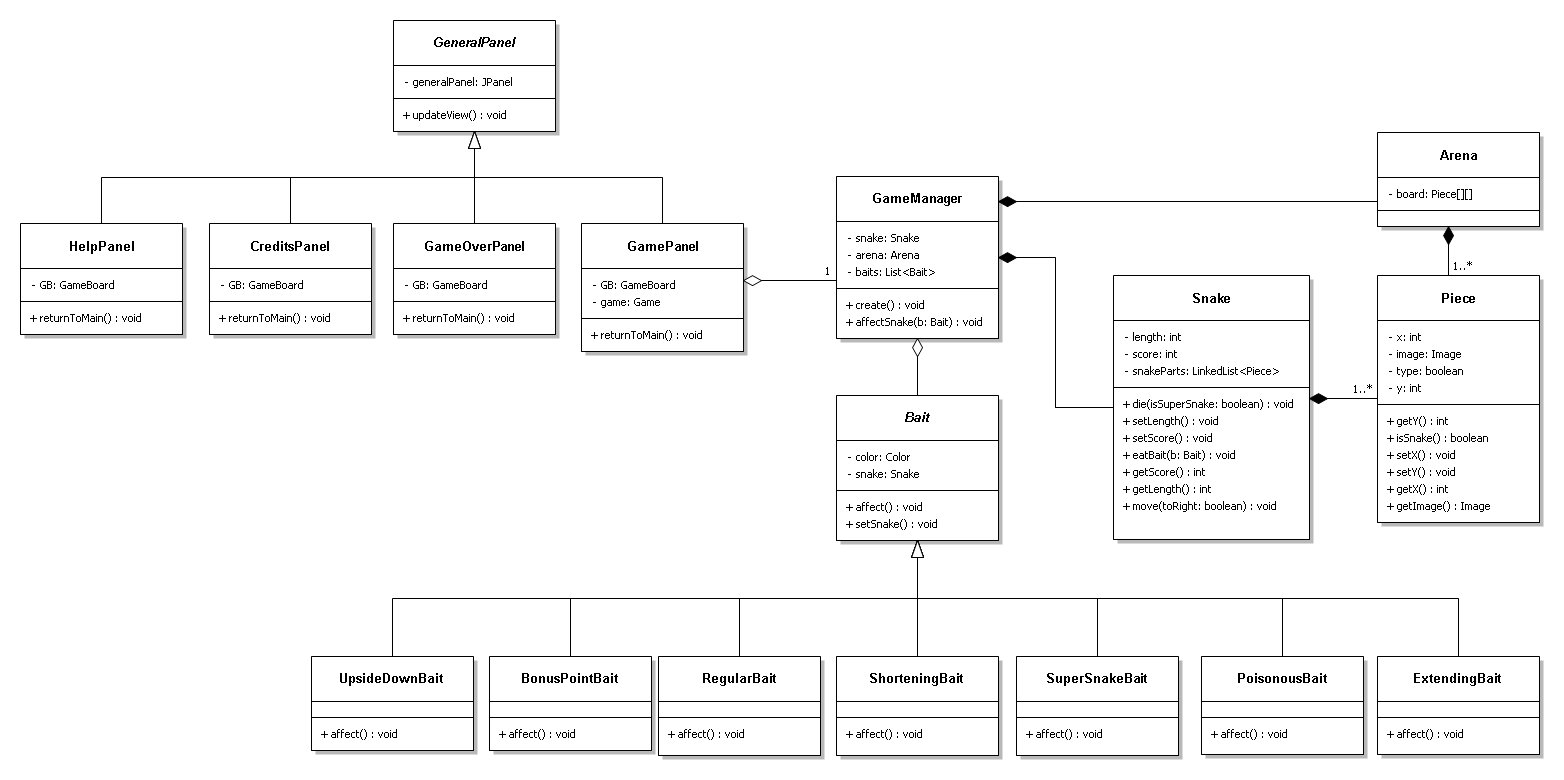


Figure 7: Class Diagram of Class Hierarchy

**3.2. DYNAMIC MODELS**

**3.2.1. STATE CHART DIAGRAMS**

The following figures illustrate the state chart diagrams given out of the proposed system.

**Snake's BonusPointBait State Chart Diagram:**

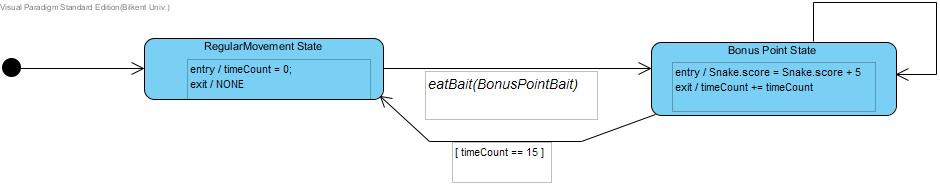


Figure 8: Snake's BonusPointBait State Chart Diagram

Here in this state chart diagram, the game enters the RegularMovement State without any condition and timeCount is initialized. When eatBait(BonusPointBait) method is called, game gets into BonusPoint State. The entry condition of this state is increasing the score of the Snake by 5 and the exit condition is incrementing timeCount by 1. As long as timeCount is not equal to 15, BonusPoint State repeats itself. When the timeCount reaches 15, the game returns back to RegularMovement State.

**Snake's SuperSnakeBait State Chart Diagram:**

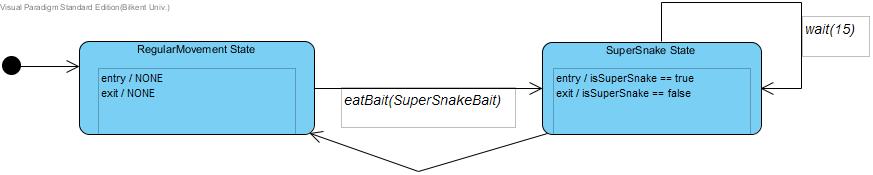
****

Figure 9: Snake's SuperSnakeBait State Chart Diagram

In the above state chart diagram, the game enters the RegularMovement State without any condition. When eatBait(SuperSnakeBait) method is called, game gets into SuperSnake State. As soon as this state becomes active, isSuperSnake variable is equalized to true and wait(15) method is called. During 15 seconds, this state continues. Then game exits from this state and isSuperSnake is equalized to false in exit. RegularMovement State becomes active again.

**Snake's State Chart Diagram for Death Scenarios:**

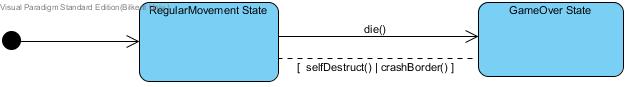
****

Figure 10: Snake's State Chart Diagram for Death Scenarios

Here in this state chart diagram, the game enters the RegularMovement State without any condition. If selfDestruct() method or crashBorder() method is called, the game exits from RegularMovement State and enter GameOver State by calling die() method.

**3.2.2. SEQUENCE DIAGRAMS**

Here are the sequence diagrams which control the flow of the game progressively.

**ExtendingBait Sequence Diagram:**

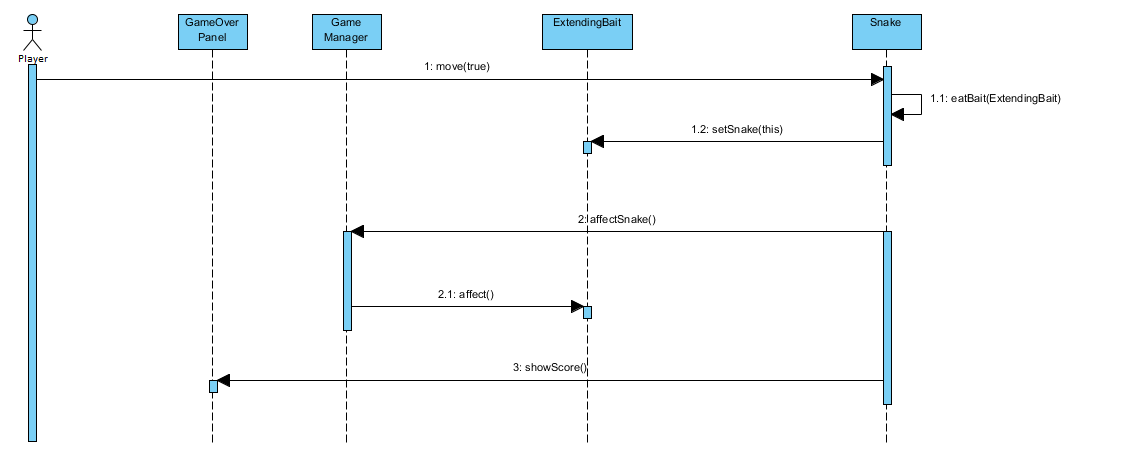


Figure 11: ExtendingBait Sequence Diagram

**PoisonousBait Sequence Diagram:**

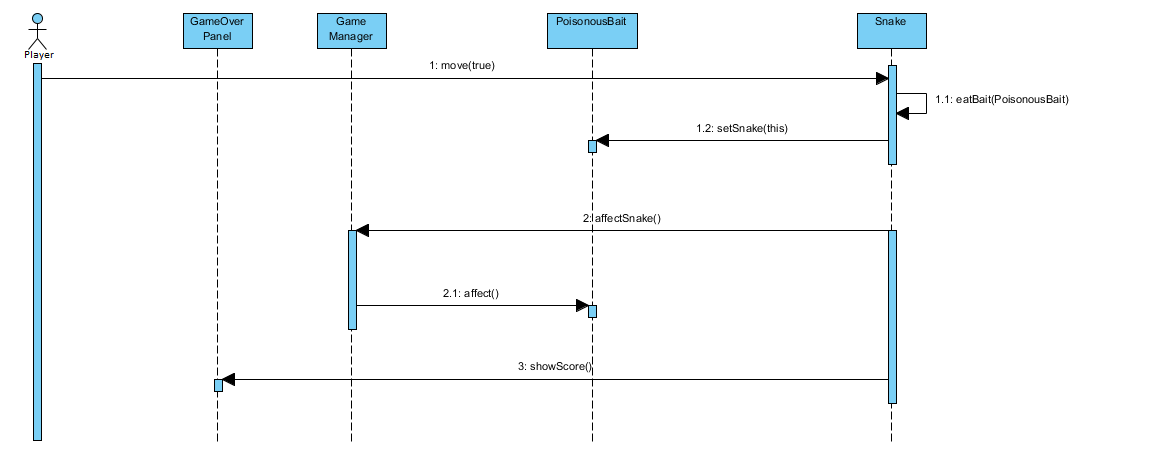


Figure 12: PoisonousBait Sequence Diagram

**ShorteningBait Sequence Diagram:**

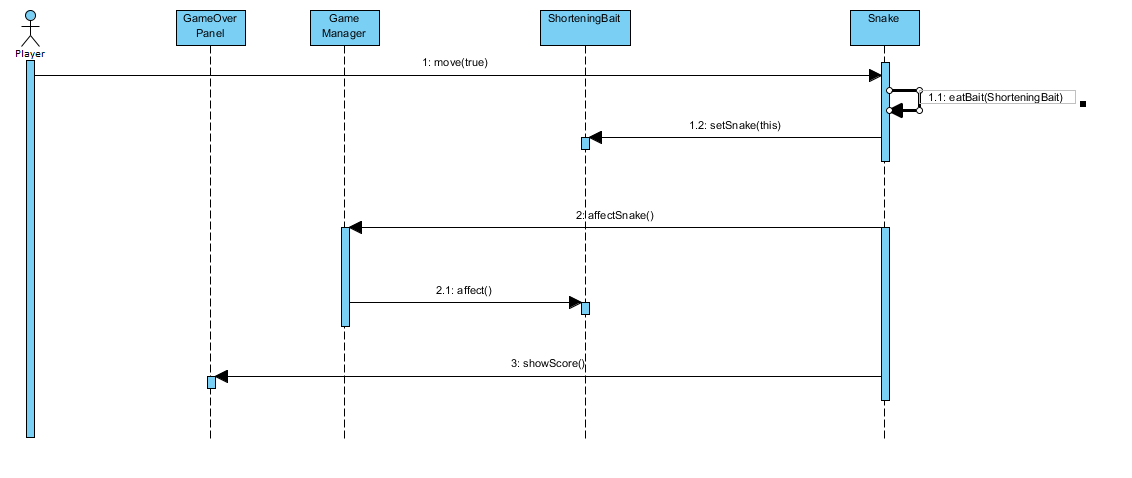


Figure 13: ShorteningBait Sequence Diagram

**SuperSnakeBait Sequence Diagram:**

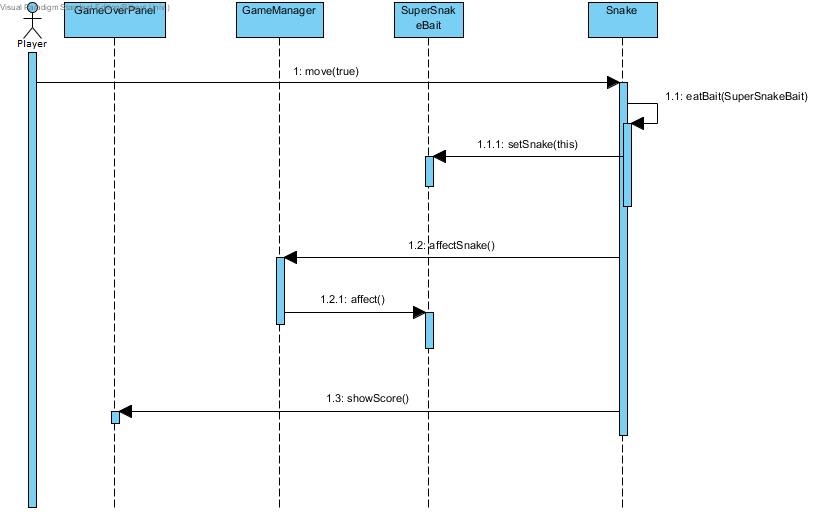


Figure 14: SuperSnakeBait Sequence Diagram

**UpsideDownBait Sequence Diagram:**

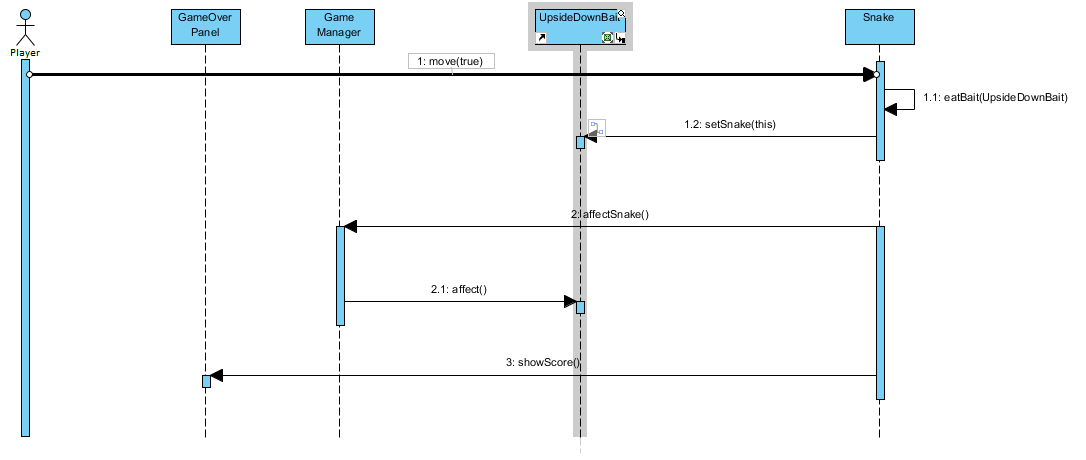


Figure 15: UpsideDownBait Sequence Diagram

**BonusPointBait Sequence Diagram:**

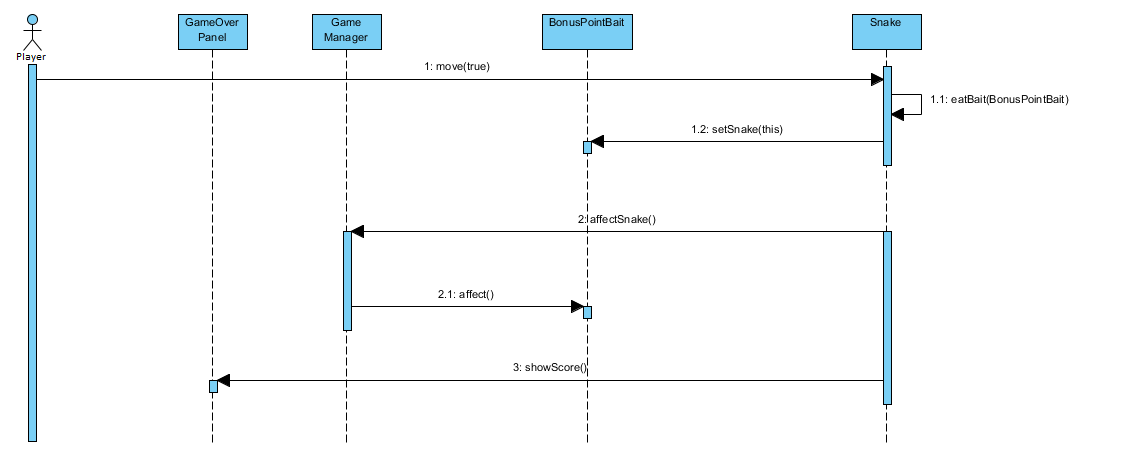


Figure 16: BonusPointBait Sequence Diagram

**RegularBait Sequence Diagram:**

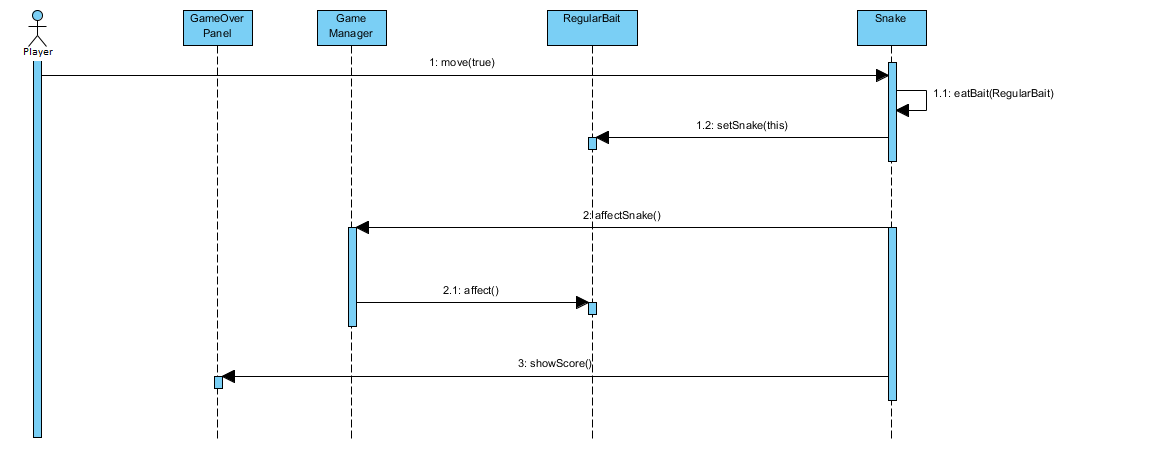


Figure 17: RegularBait Sequence Diagram

**Sequence Diagram for Regular Movement:**

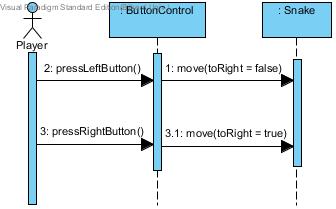
****

Figure 18: Regular Movement Sequence Diagram

**Sequence Diagram for Panel Transitions of the Game:**

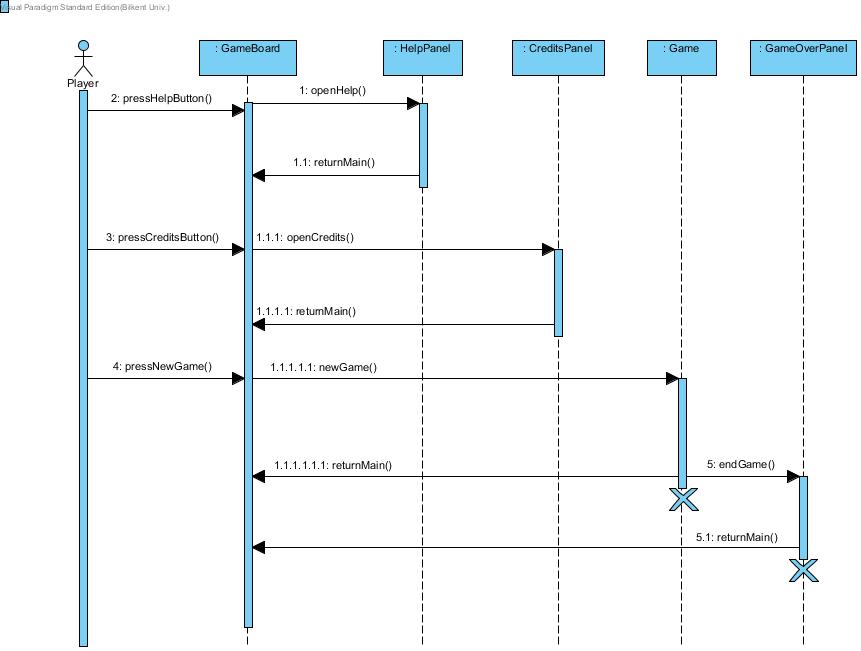
****

Figure 19: Sequence Diagram for Panel Transitions

1. **DESIGN**

*Purpose*

Snake is a classic 2D game well-known by almost everyone. In the game, the player controls the snake’s movements to eat different baits and enlarge its length in order to gain points and have the first place in high scores table. Interaction with the game and controlling snake is explained in the help panel before the player starts the game. The real challenge in the game is provided by different baits in order to get attention and interest of the player. The main purpose of our system is while player is having good time, measure his/her hand-eye coordination and reflexes.

* 1. **Design Goals**

It is very important to describe which qualities our system should focuses on before starting the implementation part of the system. Main design goals of our system are coming from the non-functional requirements. Most significant design goals can be seen below.

* + 1. **Maintenance Criteria**

**Reliability:** Snake is a reliable program because it is simple and contains no bugs or crush situations. Because our system does not have many inputs, the results of unexpected inputs are evaluated in order to prevent our program from crushing. With the each part of the designing system, every unconsidered situation that might crush the system is evaluated very carefully.

**Modifiability:** In our system modifying an existing functionality or adding another functionality would be very easy. To achieve this we used centralized control which means one subsystem controls everything.

**Portability:** Because of users want to achieve programs from different platforms such as tablets, mobile phones and PC’s, portability becomes a very significant issue while designing a system. Thus, our system will be implemented in Java to satisfy the platform independency thanks to JVM.

**User-friendliness:** Easiness in the usage of the game controllers is a significant issue in respect of good design. Because programs that makes everything easy for the user are more attractive for players. Our program is not making the gameplay boring by making the playing easy, but it is making the control and usage easy by less inputs and with a help panel which shows everything should be known and provides users a game which they can play without any prior knowledge.

**Low-Cost:** It is vital to provide a program that users can use with minimum hardware and low system requirements. Our system does not require any additional hardware except a keyboard and a mouse. Also it does not use function stack or memory too much, so our program does not require any additional costs. Test parts will also require almost nothing and a low budget will be enough for developing the software.

* + 1. **End-User Criteria**

**Ease of Use:** Because of the fact that our system is a game, game should provide good entertainment for the player. So system should provide game to user as easy to understand. In other words, system will provide player friendly and easy to perceive for player interfaces for menus, by which player will easily find desired operations, navigate through menus and perform the desired operations. Therefore through all operations and menus, system will provide easy and understandable view to point of user.

**Ease of Learning:** Since snake is a popular game and the game is not difficult to understand, user does not have to learn anything new. However since game contain different baits which all baits have different color, user supposed to know which bait have which color. It is important for the user to obtain information about the game concepts, for this understanding system will provide an instructive help menu, which he/she will be informed and getting warmed up. The point and logic of the game is quite simple which makes it entire game very easy.

**Ease of Remembering:** The game provides well-known snake game with different baits. User supposed to know the usage of directions. Also user should know colored baits have different effects. By reaching the help menu, player can understand baits color and its effects. Therefore, remembering game concepts and its rules is easy for the user.

* + 1. **Trade-Offs**

**Performance vs Portability:** We chose to implement our program in Java programming language. Because it provides us platform independent implementation with JVM. On the other side we sacrifice high performance because if we implement our program in C++ or C# its performance would be higher. In our view of design portability is more important than high performance because using other programming language changes the performance slightly but does not gives us any portability.

* + 1. **Definitions and Abbreviations with References**

2D [1]: 2 Dimensional (https://en.wikipedia.org/wiki/2D)

PC [2]: Personal Computer (https://tr.wikipedia.org/wiki/Ki%C5%9Fisel\_bilgisayar)

JVM [3]: Java Virtual Machine (https://en.wikipedia.org/wiki/Java\_(programming\_language)

* + 1. **Overview for Design Goals**

In this section we expressed purpose of our system, that is providing fun-time to user and challenge him by different scenarios. To achieve this purpose we also defined our main design goals which are basically portability, user-friendliness and high maintainability. In this respect we sacrificed from some functionalities to gain others and we explained these trade-offs in this section.

* 1. **Sub-System Decomposition**

*Overview*

In this part, the collection of classes of the game system and their interactions with each other are explained. The associations between the classes and the associations are identified within the subsystems. In terms of UML, the classes are shown as packages. The architecture of the system will be MVC, so the subsystems packages follow this path.

*Overall Subsystems*

The packages of the system are View - UserInterface, Controller-GameManagement and Entities Package in general. The architecture of the system follows MVC pattern in this case. The player of the game reaches the controller (GameManagement) via user interface. While playing the game, the controller manipulates the entities of the game. Then the controller updates the view-user interface accordingly. Here is diagram for the associations between the subsystems in the Snake game showing the subsystem decomposition;

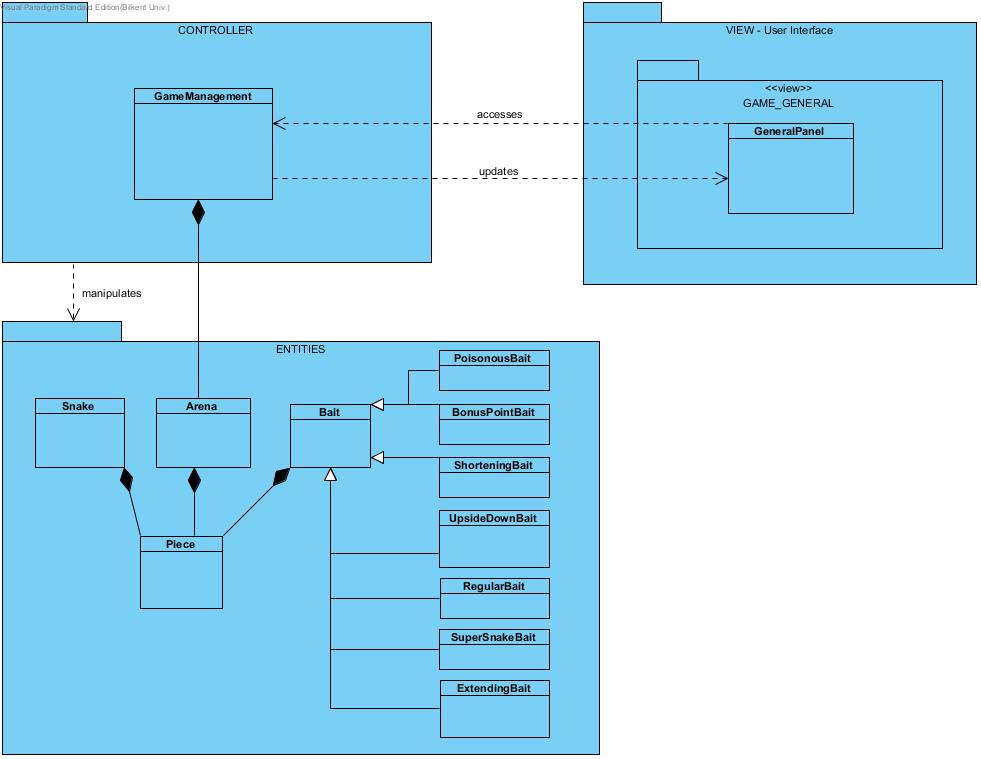


Figure 20: Subsystem Decomposition

One of the design patterns that is used in our system design is Model View Controller pattern, also known as MVC pattern. In MVC pattern, Controller handles the users actions, steers the Model according to it and updates the view in order to notify user about the changes. In our system design, View includes GeneralPanel subsystem, Controller includes GameManagement subsystem and Model includes Snake, Arena, Piece and Bait objects. As the user changes the direction of the snake GameManagement receives the action and manipulates objects in the Model view according to it and every switch in objects is shown in the matching view.

* 1. **Architectural Design**

In order to get an efficient implementation, opaque layering will be used in the game for maintainability. There are three main layers in the system: User Interface (Game Panel), Controller (GameManager) and Entities as the last layer. The following diagram illustrates the overall architecture;

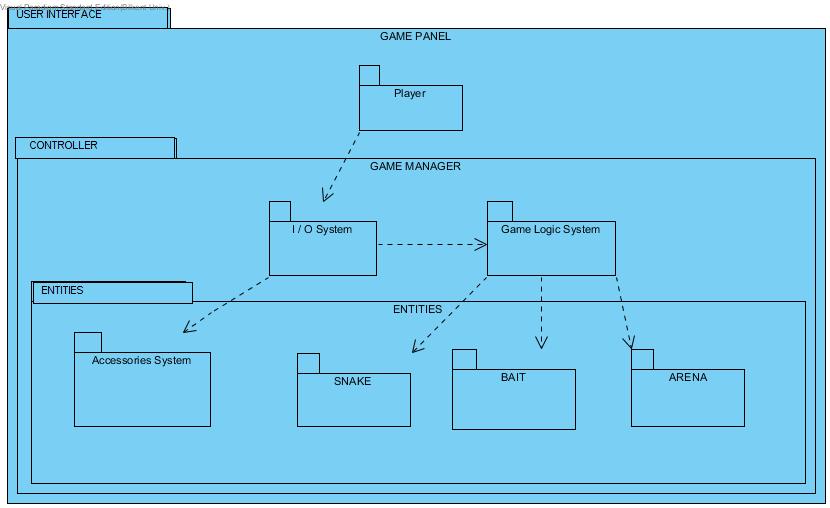


Figure 21: Opaque Layering in Architecture of the Game

Since opaque layering is used, Player package can only interact with I/O System package where direct connecting to Snake, Bait, and Arena and Accessories packages is impossible. GameLogic System package can have a direct access to Snake, Bait and Arena packages. I/O System package can reach to GameLogic System and Accessories packages.

While decomposing the subsystem of the game, the MVC architecture was used. The detailed view of the application of MVC was displayed above. The following diagram represents the overview of the MVC on the game;

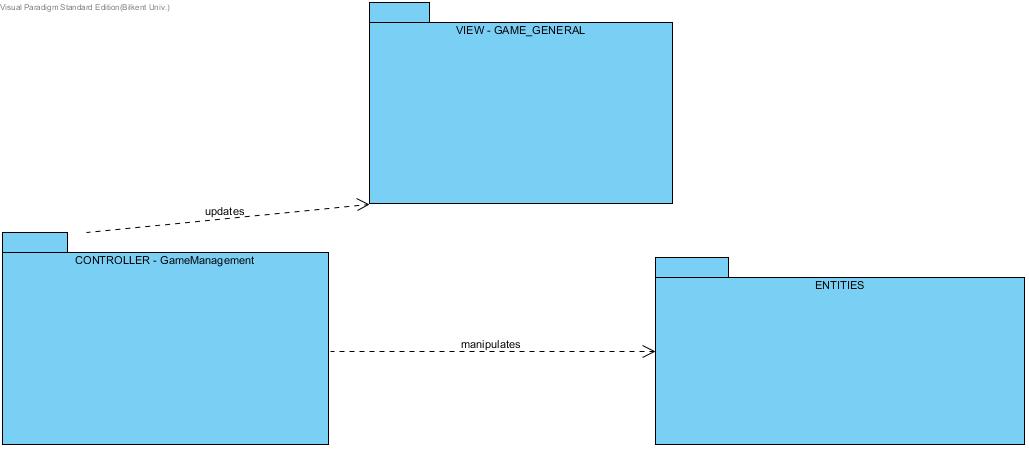


Figure : MVC pattern in general

* 1. **Hardware-Software Mapping**

In Snake, keyboard and mouse will be used as hardware components. Mouse will be active for menu actions. The user-player will be able to click on the Help Button to see the Help Panel, Credits Button to see the Credits Panel of the authors of the software and NewGame Button to be navigated for playing the game. When the game begins, right and left buttons of the keyboard will be used for directing the snake. Here is a deployment diagram for mapping of the hardwares in the game:

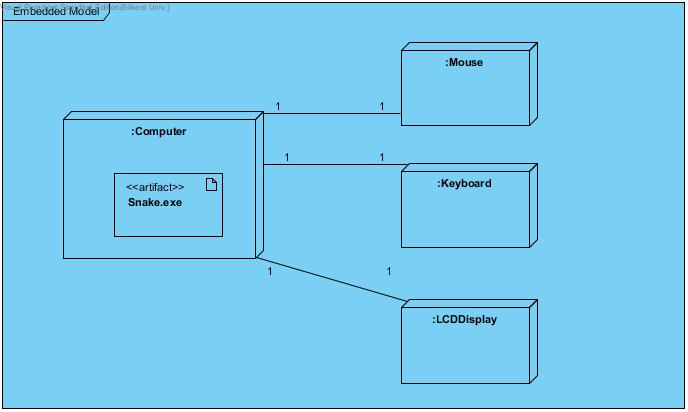


Figure 23: Hardware Mapping Deployment Diagram

* 1. **Addressing Key Concerns**
     1. **Persistent Data Management**

File system will be used for persistent data management in the game. The writer and the reader of the data will be both the software itself. The only data which will be achieved are the high scores of the game and the file format of the achieve will be text files. In order to notify the user when a high score is recorded, the game will scan through the text file and compare the new score with the existing ones.

* + 1. **Access Control and Security**

The game will support single user. Since multiple users will not be possible, there is will be no need for account controlling authentication.

* + 1. **Global Software Control**

Among three control flow mechanisms, which are Procedure-driven control, Event-driven control and Threads, we have decided to use Event-driven control mechanism for our application. In an Event-driven mechanism, the events and event handlers determine flow of the system. Since we are using Model View Controller (MVC) design pattern, Event-driven mechanism is the most appropriate choice since it separates the control view and model components. Our system has a centralized design since we have decided to have only one control object which is Game Control Panel that is responsible for managing almost all the events in each subsystems.

* + 1. **Boundary Conditions**

**Initialization:** Our system does not access any data at startup time. However there will be high score file which contains the highest scores that user had done. Also we do not register any services at startup time.

Snake game does not have an executable file so it does not require installation. It is executed as a simple .jar file.

**Termination:** Termination of Snake occurs in these cases: If the player decides to quit before, during or after the game; they can terminate by clicking the “x” button on the upper right corner of the console panel, fulfilling the death scenarios, or by choosing “quit game” option from the main menu.

**Failure:** If any of the files used in the game are corrupted at any point before the initialization or during the execution of the game, the game will be terminated with an error message. If snake occurred in same position of the wall at beginning, the game will be terminated with an error message. However, minor errors in performance will not result in termination.

1. **OBJECT DESIGN**
   1. **Pattern Applications**

***Façade Pattern***

The Façade Pattern is a structural design pattern that simplifies the interface to a complex system. It is usually composed of all the classes, which make up the subsystems of the complex system. It provides methods to the system that does what is expected by system, using the methods in the libraries and packages.

Since the Façade Pattern provides a useful interface for the subsystems, we used this pattern in our subsystem decomposition.

GameManager class contains common operations inside of it and provides an interface for these classes. It brings the common things together. In addition, Bait interface brings all sub bait classes together. Bait interface contains the common operations such affect.

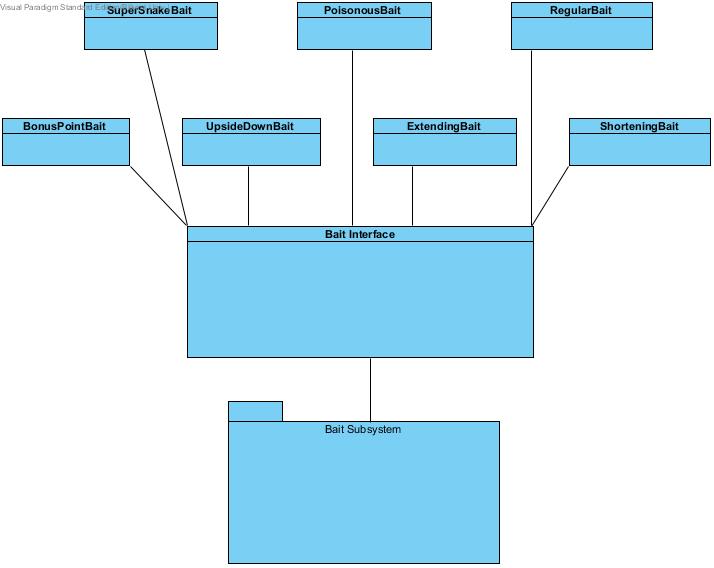
****

Figure : Façade Pattern

***Observer Pattern***

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

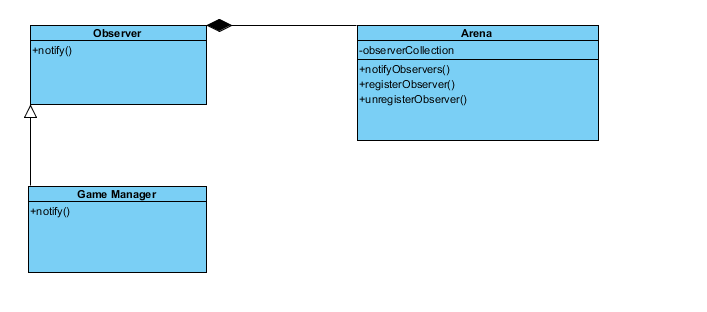


Figure : Observer Pattern

The GameManager class plays the role of Observer and the Piece class is the Subject class. When Player take one Piece and make a movement, that means the location of that Piece has been changed and the Piece will notify to the Game Manager to check the movement whether it is valid or not.

***Command Pattern***

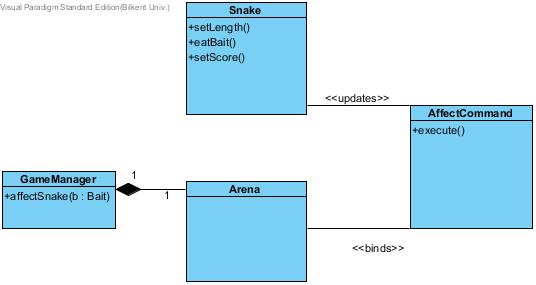
******

Figure : Command Pattern

In our proposed system, all of the interactions done in the Arena (game platform) are controlled by GameManager. When the Snake's state changes by means of eating any kind of bait, self-destruction or crashing the walls; an affect command is sent by the Arena class. This command changes the Snake object accordingly.

* 1. **Class Interfaces**

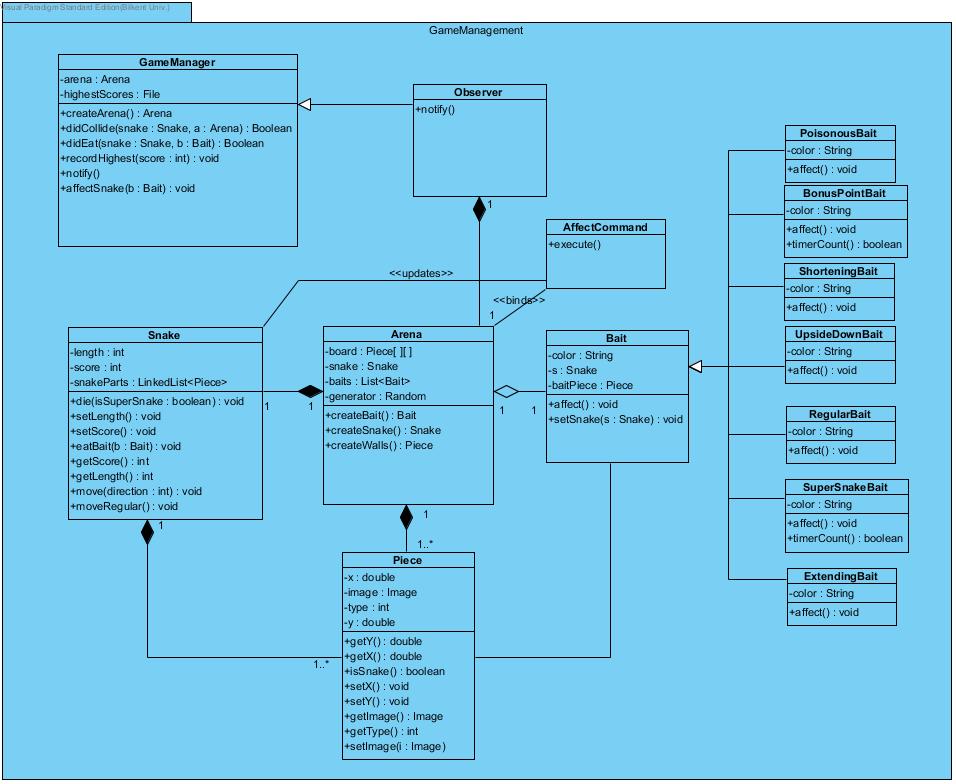


Figure 27: GameManagement Detailed Class Definitions

Our GameManagement Subsystem consists of six classes in general. There are seven extra Bait classes which extend Bait Class. This subsystem holds the core of the game and directs the objects. The above diagram illustrates the detailed class definitions of the system.

The below class diagram shows the detailed class definitions of the user interface of the software. GAME\_GENERAL is the class where in software is driven with the main method and all the classes are the specific panel which appear relatively throughout the system.

### C:\Users\user\Desktop\okul\CS 319 - 2\Userinterface class.jpg

Figure 28: User Interface Sytem Class Hierarchy

### GameManager Class

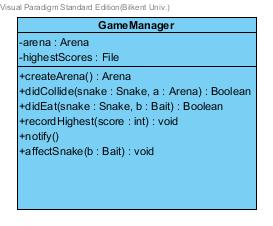


Figure : GameManager Class

### *Attributes:*

* Arena arena : GameManager holds an instance of Arena class to control the game instructions.
* File highestScores : Holds the record of the highest scores of the game.

#### Constructors:

#### public gameManager(): It is the implementation for credit interface with no parameters.

### *Methods:*

* void affectSnake(Bait b): Method affect the snake with various affect according the color of bait.
* void recordHighest (int score): Scans the highestScores file and records the score if it is the new highest score.
* boolean didCollide (Snake s, Arena a): Return true if the Snake object hits any of the Arena piece by comparing coordinates.
* boolean didEat(Snake s, Bait b): Return true if the Snake object eats the given Bait piece by comparing coordinates.
* void createArena(): Creates an Arena object in initialization of the game.
* notify(): Notifies the Arena about the changes as a part of Observer Pattern.

Snake Class

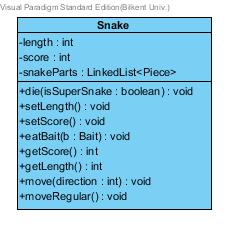


Figure : Snake Class

#### Attributes:

* private int length: This integer holds the current length of the snake and updated when it changes.
* private int score: This integer holds the current total score of the snake and updated when it changes.
* private LinkedList snakeParts: This LinkedList holds the Pieces that the current snake consist of and it is updated with the every single move of the snake.

#### Constructors

* public Snake( ): Creates a Snake object without any parameters in the exact part of the Arena.

#### Methods:

* public void die(bool isSuperSnake): This method ends the game if certain conditions is satisfied. If the boolean isSuperSnake is true die method does not ends the game.
* public void setLength(): This method sets the length of the snake to desired length without any parameters.
* public void setScore():This method sets the score of the snake to desired score without any parameters.
* public void eatBait(Bait b): This method eats the bait that is given in the parameter and lets it apply its effects on the snake.
* public int getLength(): This method gets the current length of the snake.
* public int getScore(): This method gets the current score of the snake.
* public void move(int direction):This method enables snake to turn around. If boolean toRight is true it moves to right, else it moves to left.
* public void moveRegular() : This method controls the regular movement of the Snake.

Arena Class:



Figure 31: Arena Class

### *Attributes:*

* Piece[][] board : 2D Array of Piece objects as a platform for the game
* Snake s: Represents the Snake instance in the game and carries the properties
* List <Bait> baits: This is like a bag of Bait object which will pop baits randomly on the game platform.
* Random generator: Generates random integers for selecting the baits.

#### Constructors:

* public class Arena(): It is implementation for arena with no parameters.

*Methods:*

* void createBait() : Creates the List of baits and sends a selected bait to the game platform.
* void createSnake() : Initiates the Snake object.
* void createWalls(): Creates the borders of the platform.

Bait Class

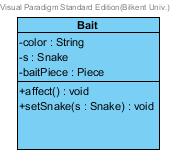


Figure 32: Bait Class

#### Attributes:

* private string Color: This attribute holds the color of the bait.
* Piece baitPiece : Baits are represented with a single Piece object.
* private Snake s: For changing the Snake's attributes, a Snake instance is held.

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game.
* public void setSnake(): Necessary method for reaching Snake object.

#### Constructors

* public Bait(double coordinate\_X, double coordinate\_Y ): Creates a Bait object without any parameters in the exact part of the Arena.

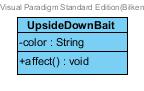


Figure 33: Representative Class Diagram for All of the baits

#### UpsideDown Bait Class:

#### Attributes:

* private string Color: This string holds the color of the bait which is "yellow".

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, the method exchanges the coordinates of head and the tail of the Snake, so that it starts to move in the opposite direction.

#### Constructors

* public UpsideDown(double coordinate\_X, double coordinate\_Y): Creates a Bait object with given parameters.

BonusPointBait Class:

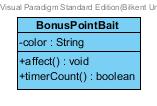


Figure : BonusPoint Bait Class

#### Attributes:

* private string Color: This string holds the color of the bait which is "cyan".

*Methods:*

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, within a period of time, the snake scores 5 times more points with each bait it eats.
* public boolean timerCount() : Determines the duration of the bait's effects.

#### Constructors

* public BonusPointBait( double coordinate\_X, double coordinate\_Y):Creates a Bait object with given parameters.

#### ExtendingBait Class:

#### Attributes:

* private string Color: This string holds the color of the bait which is "blue".

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, its length increases by 5 units (pieces).

#### Constructors

* public ExtendingBait( double coordinate\_X, double coordinate\_Y):Creates a Bait object with given parameters.

#### PoisonousBait Class:

#### Attributes:

* private string Color: This string holds the color of the bait which is "green".

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, the Snake dies and game finishes.

#### Constructors

* public PoisonousBait( double coordinate\_X, double coordinate\_Y):Creates a Bait object with given parameters.

#### ShorteningBait Class:

#### Attributes:

* private string Color: This string holds the color of the bait which "magenta".

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, the length of the snake is reduced by 5, down by 5 units of Pieces visually is the length is more than 5 originally.

#### Constructors

* public ShorteningBait( double coordinate\_X, double coordinate\_Y):Creates a Bait object with given parameters.

#### RegularBait Class:

#### Attributes:

* private string Color: This string holds the color of the bait which is "black".

#### Methods:

* public void affect(): This method applies the effects of eating this bait to the snake and the game. In this case, the regular bait just increases the score by one.

#### Constructors

* public RegularBait( double coordinate\_X, double coordinate\_Y): Creates a Bait object with given parameters.

#### SuperSnakeBait Class:

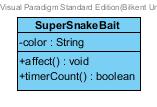


Figure : SuperSnake Bait Class

#### Attributes:

* private string Color: This string holds the color of the bait which is "red".

#### Methods:

* public void affect(): This method applies the effects of eating this special bait to the snake and the game. In this case, within the given time, the snake's die method is not called in death situations except self-destruct.
* Public timerCount(): Calculating the time until the end of effect.

#### Constructors

* public SuperSnakeBait( double coordinate\_X, double coordinate\_Y):Creates a Bait object with given parameters.

Piece Class:

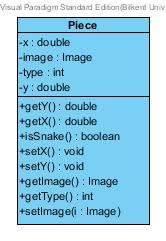


Figure 36: Piece Class

### *Attributes:*

* private double x: setting x coordinate as double for accuracy
* private double y: setting y coordinate as double for accuracy
* private int type: Determines whether the Piece object is a Snake piece, Arena piece or a Bait piece.
* private Image m: creating image object for piece

#### Constructors:

* public class Piece(int x, int y, Image m, boolean boo): It is implementation for credit interface with no parameters.

### *Methods:*

* int getX(): to get x coordinate
* int getY(): to get y coordinate
* boolean isSnake(): checking whether specific piece is obtained by snake or not
* void setX(int x): to set x coordinate
* void setY(int y): to set y coordinate
* Image getImage(): to get Image object

**Interface Classes:**

HelpPanel Class:

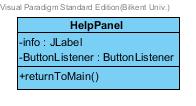


Figure : HelpPanel Class

*Attributes:*

* private JLabel info: This will store the simply help information for the players. It is JLabel type because it will be fitted to the panel.
* private ButtonListener listener: This is the listener for the back button.

*Constructors:*

* public HelpPanel: Initializes the helpInfo attribute by default content. It requires no parameters.

*Methods:*

* public void returnToMain(): This method will make the game return to the main menu.

CreditsPanel Class:

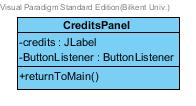


Figure : CreditsPanel Class

### *Attributes:*

* private JLabel info: This will store the simply help information for the players. It is JLabel type because it will be fitted to the panel.
* private ButtonListener listener: This is the listener for the back button.

#### Methods:

* public void returnToMain(): It will simply return to main menu after user has seen credits.

#### Constructors:

* public credit\_interface(): It is implementation for credit interface with no parameters.

#### GameOverPanel Class:



Figure : GameOverPanel Class

#### Attributes:

* private JLabel info: This will store the simply help information for the players. It is JLabel type because it will be fitted to the panel.
* private ButtonListener listener: This is the listener for the back button.

#### Methods:

* public void returnToMain (): It will simply return to main menu after user has seen the score.

*Constructor:*

#### public gameOverInterface(): It is implementation for game over interface without any parameter.

GamePanel Class:

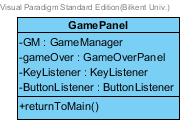


Figure : GamePanel Class

### *Attributes:*

* private ButtonListener listener: This is the listener for the back button.
* GameOverPanel gameOver: GamePanel hold the instance of GameOverPanel in order to open this panel when game finishes.
* KeyListener keyListener: In order to determine the actions of the left and right buttons, this panel has an instance of KeyListener of ActionListener.
* GameManager GM: For playing the game and showing the game platform, the panel holds the instance of GameManager class.

#### Methods:

* public void returnToMain(): It will simply return to main menu after user has seen credits.

#### Constructors:

* public GamePanel(): It is implementation for credit interface with no parameters.

GeneralPanel Class:

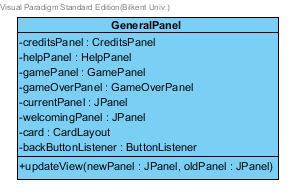


Figure : GeneralPanel Class

### *Attributes:*

* private backButtonListener listener: This is the listener for the back button.
* GamePanel GamePanel, CreditsPanel creditsPanel, HelpPanel helpPanel, GamePanel, GameOverPanel gameOverPanel: GeneralPanel holds the instances of all of the panels in order to manage the panel transition of the software.
* JPanel welcomingPanel: Shows the initial view of the software with the help, newGame, credits buttons.
* CardLayout card: The layout of the panel transitions.
* JPanel currentPanel:

#### Methods:

* public void updateView(JPanel newPanel, JPanel oldPanel): Changes the panels accordingly.

#### Constructors:

* public GeneralPanel(): It is implementation for credit interface with no parameters.

GAME\_GENERAL:

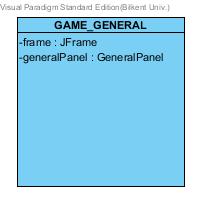


Figure : GAME\_GENERAL (driver of the game)

### *Attributes:*

* JFrame frame : Holds the GeneralPanel in which is the software is run.
* GeneralPanel generalPanel: GAME\_GENERAL holds the instance of GeneralPanel to initialize the software.

*Method:*

* public static void main(String[] args) : The software is initialized.
  1. **Specifying Contracts**

// If the bait is RegularBait its color should be black

1. context RegularBaitinv: color = “black”

// If the bait is ShorteningBait its color should be purple

2. context ShorteningBaitinv: color = “magenta”

// If the BonusPointBait is active, the score is multiplied by 5 everytime

3. Snake.score \*= 5;

// If the bait is BonusPointBait its color should be cyan

4. context BonusBaitinv: color = “cyan”

// If the bait is PoisonousBait its color should be green

5. context PoisonousBaitinv: color = “green”

// If the bait is ExtendingBait its color should be blue

6. context ExtendingBaitinv: color = “blue”

// If the bait is UpsideDownBait its color should be yellow

7. context upside-down Bait inv: color = “yellow”

// If the bait is SuperSnakeBait its color should be red

8. context superSnakeBaitinv: color = “red”

// If the Snake collides with the borders of the platform

9. context GameManagerinv:

self.didCollide() = true

// In order to decide to eat, didEat method should occur

10. context GameManagerinv:

self.didEat() = true

// In order to finish BonusPoint state

11. context GameManagerinv:

BonusPointBait.timeCount = true;

// Coordinates of baits should not expand the arena

12. context Arena inv:

bait.getCoordinates() <= self.getCoordinates()

// Setting initial length of snake to 3

13. context Snake inv:

length = 3

// Starting coordinates of snake of X setting into initial\_x

14. context Snake inv:

startCoor = initial\_x

// Startingcoordiantes of snake of Y setting into initial\_Y

15. context Snake inv:

startCoor = initial\_y

// Deciding whether the piece belongs to snake or arena

16. context Piece inv:

self.isSnake() = true

// Checking the bait in arena or not

17. context Bait inv:

self.getXCoordinates() <= Arena.getXCoordinates()

// Checking for the highest score

18. context GameManagerinv:

isHighest(Snake.score, highestScores) = true;

// Setting initial score to 0

19. context Snake inv:

Snake.score = 0

// Setting the length of the snake to the desired length

20. context Snake inv:

setLength() = desiredLength;

// Setting the score to the desired score

21. context Snake inv:

setScore() = int desiredScore

// Setting the X dimension of the piece

22. context Piece inv:

setX() = int coordinate\_X

// Setting the Y dimension of the piece

23. context Piece inv:

setY() = int coordinate\_Y

//Returning the type of the Piece as an integer

24. context Piece inv:

self. getType()

//Returning the X coordinate of Piece as a double

25. context Piece inv:

self.getX()

//Returning the X coordinate of Piece as a double

26. context Piece inv:

self.getY()

//Returning the length of the Snake as an integer

27. context Snake inv:

self. getLength()

//Returning the score as an integer

28. context Snake inv:

int snakeScore = getScore()

//Storing the highest score obtained at the end of the game

29. context GameManager inv:

recordHighest(score :int)

//Lets the bait to impose its effects on Snake if eating action occurs

30. context Snake inv:

eatBait(b :Bait)

1. **Conclusions and Lessons Learned**

In the analysis part of the report, we tried to explain what our project is, what it does, which functionalities it has, what we aim to do and how we will do it. We understood what requirements, scenarios do. Firstly, we specify game features, functional and non-functional requirements. Functional and non-functional requirements, scenarios help us before considering use case diagrams. Additionally, we saw more detailed image of the project. We used case tools such as Visual Paradigm and using Unified Modeling Language (UML). UML diagrams are really helping us to show how our program works. We created Use Case diagrams, Sequence diagrams, State diagrams by using Visual Paradigm. In the system design part of the report, we have explained design goals and key concerns. We divided our system into subsystems in order to look at them in a more focused way. We defined the most fitting architectural designs into our game and mapped the hardwares accordingly. Finally, we adjusted our system into three different design patterns which are façade, command and observer patterns with the detailed class diagrams and we defined the contracts in our system.

**Lessons Learned**

In this course, we learn how to design a project within a detailed documentation. In each phase of the project, we add up something. We changed some parts while thinking critically after each step. We also use the topics, which we have learnt in the class. For instance, we learn about functional and nonfunctional requirements, scenarios and we also learn how to make object modeling and dynamic modeling. While we are making object modeling we learn how to draw class diagram and in dynamic and object modeling through use case diagram, sequence diagram, class diagram, state diagram. In addition, we specified design goals, key concerns, architectural patterns and design patterns which are also discussed in lessons. We have also learnt how to apply patterns for our project.

**SOFTWARES USED**

Softwares:

* Class & State Chart & Sequence Diagrams: Visual Paradigm 12.2

Standard Edition

NClass Standard Edition

* User Interface Models: ForeUI 4.00

Easynth Solution Inc. Ltd.