
Bilkent University

Department of Computer Engineering

# Object Oriented Software Engineering

*CS 319 Project: The Hungry Beast*

Final Report

Eren Erol
Hande Özyazgan
İrem Ergün
Yalın Eren Deliorman

Course Instructor: Özgür Tan

7.12.2015

# Contents

## 5.0 Object Design

This section covers object design of our project.

### 5.1 Pattern Applications

#### 5.1.1. Facade Design Pattern

**Description:** Façade Pattern is a structural design pattern. It provides a better interface to the whole system. It helps the understandability of the system and makes the complex codes to understand easier.

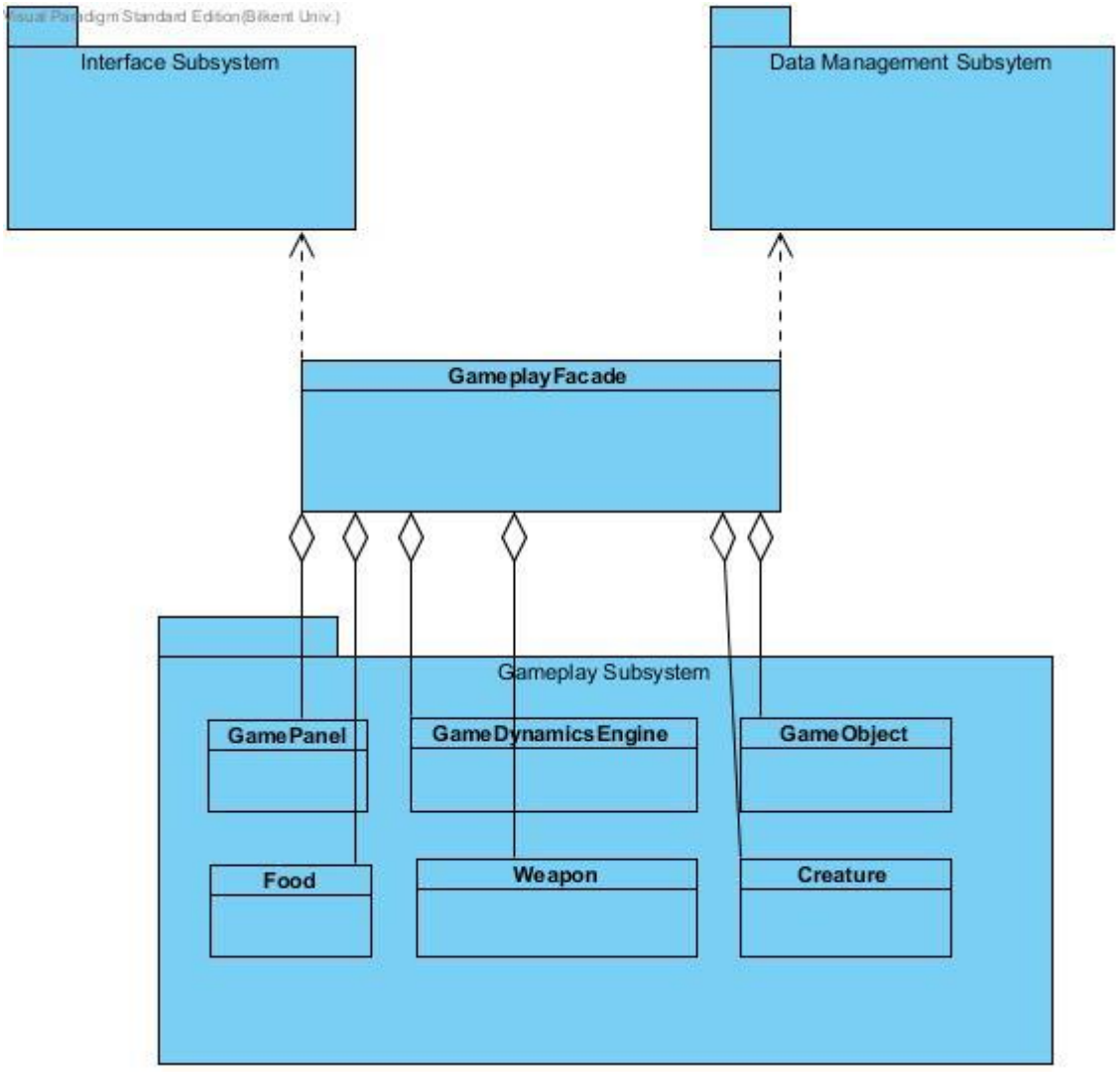


*Image 1: Shows Hungry Beast's Façade Design Pattern.*

**Usage:** Weapon, Food and Creature classes, which are abstract, uses Gameplay Subsystem. We used GameplayFacede as the connection between the subsystems. This Façade uses GamePanel, GameDynamicsEngine, GameObject, Food, Weapon

and Creature objects. Then makes the relations between interface subsystem and Data Management Subsystem.

### 5.1.2. Strategy Design Pattern

**Description:** Strategy Design Pattern is a software design pattern that enables an algorithm to behave differently at the runtime. It defines a family of algorithms, encapsulate them and change their usage at run time.

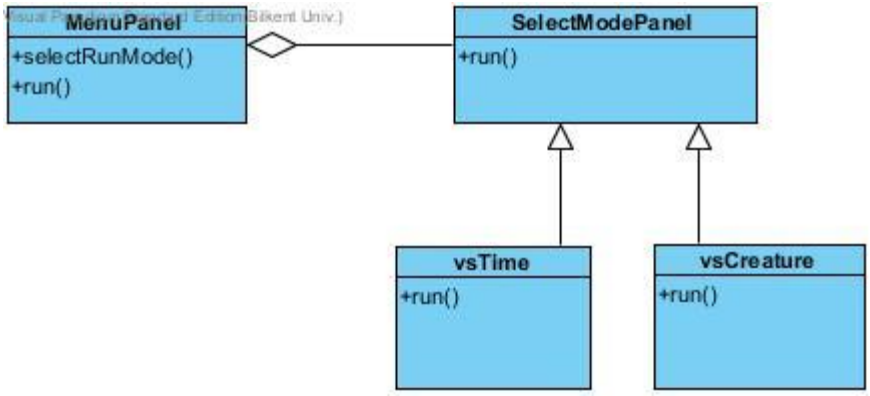


*Image 2: Shows Hungry Beast's Strategy Design Pattern.*

**Usage:** We used strategy design pattern in order to decide whether vs Creature game mode is going to run or vs Time mode. In SelectModePanel class, there is a Boolean variable ifTime for making this decision. Program will compile run() method but the method will choose the game mode in run time according to that Boolean variable value.

### 5.1.3. Composite Design Pattern

**Description:** Composite Design Pattern is a partitioning design pattern. It describes a group of objects that are going to be treated as a single instance of an object. It composes objects into tree structure to represent whole system.

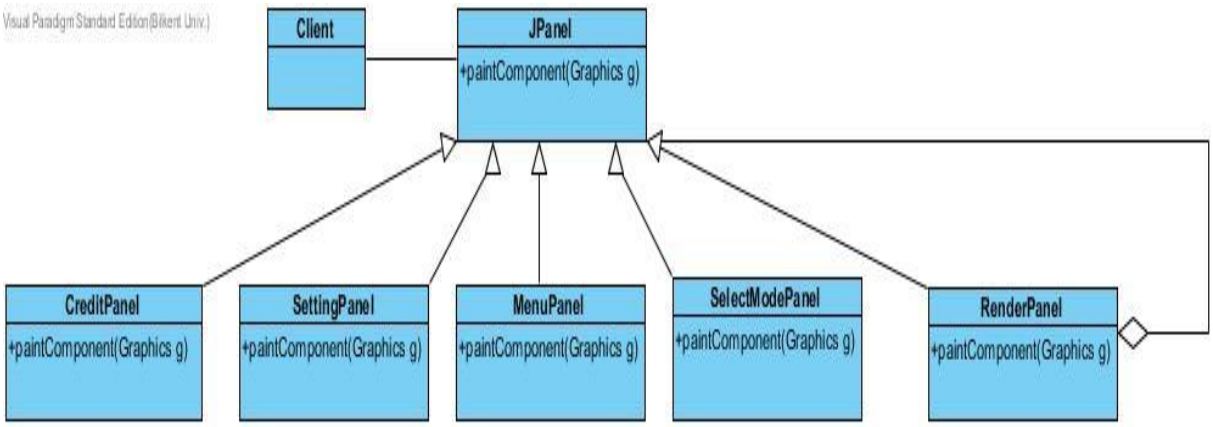


*Image 3: Shows Hungry Beast's Composite Design Pattern.*

**Usage:** We used a java class, JPanel, as a parent of the composite design structure. RenderPanel has an aggregation and inheritance relation with the JPanel. The resting panels which are CreditPanel, SettingPanel, MenuPanel, SelectModePanel has only inheritance relation. All classes except from JPanel have paintComponent(Graphics g) function which is derived from JPanel.

## 5.2 Class Interfaces

Inheritence and Delegation conventions are explained and the interface of every single classes is shown below.

### Beast:

| **Beast** |
|---|
| +beastHealth : int |
| +Beast(String imagePath, float x, float y)<br>+left(): void<br>+right(): void<br>+translateX(float xDif): void<br>+updatePosition(float timePased): void<br>+collided(GameObject g): void<br>+getHealth(): int |

Figure 1: Beast Class

**Extends:** GameObject

This class creates a Beast object with given coordinates and an image. It has a constructor Beast(String imagePath, float x, float y) public property beastHealth(int) and left(void), right(void), translateX(float xDif): void, updatePosition(float timePased): void, collided(GameObject g): void, getHealth(): int as public methods.

### Weapon:

| **Weapon** |
| --- |
| #power: double |
| +Weapon(String imagepath, float x, float y)<br>+updatePosition(float timePassed):void |

Figure 2: Weapon Class

This class is an **abstract** that enable us to create Weapon objects. It has a protected property power(double) It has a public constructor Weapon(String imagepath, float x, float y) and public method updatePosition(float timePassed):void.

### Bullet:

| **Bullet** |
| --- |
| |
| +Bullet(x: float, y: float)<br>+collided(GameObject g): void |

**Extends**: Weapon

Figure 3: Bullet Class

This class creates a Bullet object with given coordinates. It has a public constructor Bullet(x: float, y: float) and a public method collided(GameObject g): void.

### LaserGun:

| **LaserGun** |
| --- |
| |
| + LaserGun(String imagepath, float x, float y)<br>+ collided(GameObject g): void |

**Extends**: Weapon

Figure 4: LaserGun Class

This class creates a LaserGun object with given coordinates.It has a public constructor LaserGun(String imagepath, float x, float y) and a public method collided(GameObject g): void.

### Wood:

| Wood |
| --- |
| |
| + Wood(String imagepath, float x, float y)<br>+ collided(GameObject g): void |

**Extends**: Weapon

Figure 5: Wood class

This class creates a Wood object with given coordinates and an image. It has a public constructor Wood(String imagepath, float x, float y) and a public method collided(GameObject g): void.

### Food:

| Food |
| --- |
| #foodHealth: double<br>#location: Vector2D<br>#enableInc: boolean |
| +Food(x: float, y: float) |

Figure 6: Food Class

This class is an **abstract** class that enables us to create Food. It has 3 protected properties that are foodHealth(double), location(Vector2D), enableInc(boolean) and a public constructor Food(x: float, y: float).

### Meat:

| Meat |
| --- |
| |
| +Meat(x: float, y: float) |

**Extends**: Food

Figure 7: Meat Class

This class is used to create Meat objects by using x and y coordinates. It is derived from Food class. It uses the Food class's properties by calling super(). It also has a unique foodhealth. It has a Meat(x: float, y: float) constructor.

### Sugar:

| Sugar |
| --- |
| |
| +Sugar(x: float, y: float) |

**Extends**: Food

Figure 8: Sugar Class

This class is used to create Sugar objects by using x and y coordinates. It is derived from Food class. It uses the Food class's properties by calling super(). It also has a unique foodhealth. It has a Sugar(x: float, y: float) constructor.

### Vegetable:

| Vegetable |
| --- |
| |
| +Vegetable(x: float, y: float) |

**Extends**: Food

Figure 9: Vegetable Class

This class is used to create Vegetable objects by using x and y coordinates. It is derived from Food class. It uses the Food class's properties by calling super(). It also has a unique foodhealth. It has a Vegetable(x: float, y: float) constructor.

### Vector2D:

| Vector2D |
| --- |
| -x: float<br>-y: float |
| +Vector2D(x: float, y: float)<br>+getX(): float<br>+getY(): float<br>+scale(value: float): Vector2D<br>+sum(vector: Vector2D): Vector2D<br>+normalize(): void<br>+normalized(): Vector2D<br>+magnitude(): float<br>+dotProduct(vector: Vector2D): float<br>+getProjection(vector: Vector2D): float<br>+substract(vector: Vector2D): Vector2D<br>+distance(vector: Vector2D): float<br>+ toString():String |

| |
|---|
| +crossProduct(vector: Vector2D): float<br>+setX(x: float): void<br>+setY(y: float): void |

Figure 10: Vector2D Class.

This class represents the Vector2D object which will be created for other objects and keep x and y coordinates. With this class, it is possible to use mutator and accessor methods for the objects which is created as Vector2D and it also enable us manage those objects. It has 2 private properties x(float) and y(float). It has a Vector2D(x: float, y: float) constructor and getX(), getY(), scale(value: float), sum(vector: Vector2D), normalize(), normalized(), magnitude(), dotProduct(vector: Vector2D), getProjection(vector: Vector2D), substract(vector: Vector2D), distance(vector: Vector2D), toString(), crossProduct(vector: Vector2D), setX(x: float), setY(y: float) as public methods.

### CreditPanel:

| **CreditPanel** |
|---|
| -serialVersionUID: long<br>-frame: MainFrame<br>-BufferedImage: img |
| +CreditPanel(frame: MainFrame)<br>+paintComponent(g: Graphics): void<br>+menu(): void |

**Extends:** JPanel

Figure 11: CreditPanel Class

This class is for the GUI and imports many things which are related to the GUI components. It uses many functions which are derived from JPanel and create a unique panel for the credit by arranging location, size and layout. It has also a button which enable the user go back from the CreditPanel with the help of actionListener implementation. Additionally, it is possible to draw an image with the method provided as paintComponent which is derived from the Java's library. It has 3 private properties; serialVersionUID(long), frame(MainFrame), BufferedImage(img). It has a CreditPanel(frame: MainFrame) constructor. It has 2 public methods paintComponent(g: Graphics), menu().

### GameDynamicEngine:

| GameDynamicsEngine |
| --- |
| -objects: ArrayList <GameObject><br>-rp: RenderPanel<br>-beast: Beast<br>-started: Boolean<br>-frame: MainFrame<br>-settings: SettingPanel<br>-modePanel: SelectModePanel<br>-newGobjPos: Vector2D<br>-goOn: Boolean<br>- shootBlock: Boolean<br>-time: int |
| +GameDynamicEngine(frame: MainFrame, settings: SettingPanel, modePanel: SelectModePanel)<br>+run(): void<br>+keyPressed(e: Keyevent): void<br>+keyReleased(e: Keyevent): void<br>+keyTyped(e: Keyevent): void<br>+menu(): void<br>+shoot(): void<br>+updatePos(float timePassed): void<br>+createEnemy(float probability): void<br>+checkCollision(): void |

**Extend:** Thread

**Implements:** KeyListener

Figure 12: GameDynamicsEngine Class

This class is for the GUI and imports many things which are related to the GUI components. It provides the game's dynamic engine as understood by the class name and it consists of a frame and panels. With this class, the program is able to understand the keyEvents with the help of Key's actionListener which is implemented from KeyListener. It also has a button to go to previous menus. It has private properties that are objects(ArrayList <GameObject>), rp(RenderPanel), beast(Beast), started(Boolean), frame(MainFrame), settings(SettingPanel), modePanel(SelectModePanel), newGobjPos(Vector2D), goOn(Boolean), shootBlock(Boolean), time(int). It has a GameDynamicEngine(frame: MainFrame, settings: SettingPanel, modePanel: SelectModePanel) constructor. It has keyPressed(e: Keyevent), keyRealeased(e: Keyevent), keyTyped(e: Keyevent), menu(), shoot(),updatePos(float timePassed), createEnemy(float probability), checkCollision() as public methods.

### ImageReader:

| ImageReader |
| --- |
| |
| +readPNGImage(filePath: String): BufferedImage<br>+ TransformColorToTransparency(image: BufferedImage, c1: Color, c2: color): Image<br>+ ImageToBufferedImage(image: Image, width: int, height: int): BufferedImage |

Figure 13: ImageReader

This class is for the GUI and imports many things which are related to the GUI components and reads the images which are used for the game. With this class, it is possible to draw images which are needed for game. In order not to get any error try-catch clauses is used in this class. It has 3 public methods which are readPNGImage(filePath: String), TransformColorToTransparency(image: BufferedImage, c1: Color, c2: color, ImageToBufferedImage(image: Image, width: int, height: int).

### Main:

| Main |
| --- |
| |
| main(args: String[]): void |

Figure 14: Main Class

This class is the core of the game, main panel and frame are created in this class to use.

### MainFrame:

| MainFrame |
| --- |
| -currentPanel: JPanel<br>-gde: GameDynamicsEngine |
| +MainFrame(currentPanel: JPanel, title: String)<br>+startGame(): void<br>+highscores(): void<br>+tutorial(): void<br>+credits(): void<br>+settings(): void<br>+backtoMenu(): void<br>+play(boolean gameMode): void<br>+ gameOver(int score): void |

**Extends:** JFrame

Figure 15: MainFrame Class

This class is for the GUI and imports many things which are related to the GUI components. It is derived from JFrame and uses its components. This class enables user to start game and also keeps the tutorial, credits and settings of the game. It has currentPanel(JPanel) and gde( GameDynamicsEngine) as private properties. It has a MainFrame(currentPanel: JPanel, title: String) constructor and startGame(), highscores(), tutorial(), credits(), settings(), backtoMenu(), play(boolean gameMode, gameOver(int score) as public methods.

**MenuPanel:**

| **MenuPanel** |
| --- |
| -serialVersionUID: long<br>-frame: MainFrame<br>-img: BufferedImage |
| +MenuPanel()<br>+setFrame(frame: MainFrame): void<br>+paintComponents(g: Graphics): void<br>+startGame(): void<br>+highscoreScreen(): void<br>+tutorialScreen(): void<br>+creditScreen(): void<br>+settingScreen(): void |

**Extends:** JPanel

Figure 16: MenuPanel Class

This class provides MenuPanel which represent the user options to be selected for the game and it is for the GUI and imports many things which are related to the GUI components. It is derived from JPanel and uses its components. In this class, many buttons are created and added to the panel and by implementing actionListeners, it is possible to move the game's screen which are highscorScreen, tutorialScreen, creditScreen and settingScreen. It has serialVersionUID(long), frame(MainFrame), img(BufferedImage) as private properties. It has MenuPanel() constructor. It has setFrame(frame: MainFrame), paintComponents(g: Graphics), startGame(), highscoreScreen(), tutorialScreen(), creditScreen(), settingScreen() as public methods.

### RenderPanel:

| RenderPanel |
| --- |
| -serialVersionUID: long<br>-objects: ArrayList <GameObject><br>-backgroundImage: BufferedImage<br>-timeLabel: String |
| + RenderPanel()<br>+ getGameObjects(): ArrayList<br>+ setGameObjects(objects: ArrayList): void<br>+ paintComponent(g: Graphics): void<br>+ getFormattedTime(int mili): void |

**Extends:** JPanel

Figure 17: RenderPanel Class

This class is for the GUI and imports many things which are related to the GUI components. It is derived from JPanel and uses its components. With this class, it is possible to get objects and even set their properties and with the paintComponents images are drawn. It has serialVersionUID(long), objects(ArrayList <GameObject>), backgroundImage(BufferedImage), timeLabel(String) as private properties. It has RenderPanel(), getGameObjects(): ArrayList, setGameObjects(objects: ArrayList): void, paintComponent(g: Graphics): void, getFormattedTime(int mili): void as public methods.

### SelectModePanel:

| SelectModePanel |
| --- |
| - serialVersionUID: long<br>-frame: MainFrame<br>-img: BufferedImage<br>- ifTime: boolean |
| + SelectModePanel(frame: MainFrame)<br>+ getIfTime(): boolean<br>+ setIfTime(b: boolean)<br>+ paintComponent(g: Graphics): void<br>+menu(): void<br>+run(): void |

**Extends:** JPanel

Figure 18: SelectModePanel Class

This class is for the GUI and imports many things which are related to the GUI components. It is derived from JPanel and uses its components. In this class the game is started and the mode is updated according to the selection of the user. It has serialVersionUID(long), frame(MainFrame), img(BufferedImage), ifTime(Boolean) as private properties. It has a constructor which is SelectModePanel(frame: MainFrame) and public methods getIfTime(): boolean, setIfTime(b: boolean), paintComponent(g: Graphics): void, menu(): void, run(): void.

### SettingPanel:

| **SettingPanel** |
| --- |
| - serialVersionUID: long<br>-frame: MainFrame<br>-img: BufferedImage<br>-controls: boolean |
| +settingPanel(frame: MainFrame)<br>+getControls(): boolean<br>+setControls(controls: boolean): Boolean<br>+ paintComponent(g: Graphics): void<br>+menu(): void |

**Extends:** JPanel

Figure 19: SettingPanel Class

This class is for the GUI and imports many things which are related to the GUI components. It is derived from JPanel and uses its components. This class is used for the set the specific controls and these changed controls are reflected to the game. It has serialVersionUID(long), frame(MainFrame), img(BufferedImage), controls(Boolean) as private properties. It has settingPanel(frame: MainFrame), getControls(): Boolean, setControls(controls: boolean): Boolean, paintComponent(g: Graphics): void, menu(): void as public methods.

### Creature:

| **Creature** |
| --- |
| #damage: int |
| +Creature(String imagePath, float x, float y)<br>+updatePosition(float timePassed): void<br>+collided(GameObject g): void |

**Extends:** GameObject

Figure 20: Creature Class

It creates a creature object. It has a protected property damage(int). It has Creature(String imagePath, float x, float y) as constructor and updatePosition(float timePassed): void, collided(GameObject g): void as public methods.

### TutorialPanel:

| TutorialPanel |
| --- |
| -serialVersionUID: static final long<br>-frame: MainFrame<br>-img: BufferedImage |
| + TutorialPanel(MainFrame frame)<br>+ paintComponent(Graphics g): void<br>+ menu(): void |

**Extends:** GameObject

Figure 21: TutorialPanel Class

It creates the tutorial panel in the game. It has serialVersionUID: static final long, frame: MainFrame, img: BufferedImage as private properties. It has TutorialPanel(MainFrame frame), paintComponent(Graphics g): void, menu(): void as public methods.

### GameOverPanel:

| GameOverPanel |
| --- |
| -serialVersionUID: static final long<br>-frame: MainFrame<br>-img: BufferedImage |
| + GameOverPanel(MainFrame frame, int score)<br>+ paintComponent(Graphics g): void<br>+ menu(): void |

**Extends:** JPanel

Figure 22: GameOverPanel Class

It creates the game over panel in the game. It has serialVersionUID: static final long, frame: MainFrame, img: BufferedImage as private properties. It has

GameOverPanel(MainFrame frame, int score), paintComponent(Graphics g): void, menu(): void as public methods.

### GameObject:

| GameObject |
| --- |
| # image: BufferedImage<br># position: Vector2D<br># collideRadius: float |
| + GameObject(String imagePath, float x, float y, float collideRadius)<br>+ getImage(): BufferedImage<br>+ getPosition(): Vector2D<br>+ doesCollide(GameObject g): Boolean<br>+ updatePosition(float timePased): abstract void<br>+ collided(GameObject g): abstract void |

Figure 23: GameObject Class

It creates a GameObject in the game. It has image: BufferedImage, position: Vector2D, collideRadius: float as protected properties. It GameObject(String imagePath, float x, float y, float collideRadius), getImage(): BufferedImage, getPosition(): Vector2D, doesCollide(GameObject g): Boolean, updatePosition(float timePased): abstract void, collided(GameObject g): abstract void as public methods.

### DavidDavenport:

| DavidDavenport |
| --- |
| |
| + DavidDavenport(float x, float y) |

**Extends:** Creature

Figure 24: DavidDavenport Class

It creates a DavidDavenport object. It has DavidDavenport(float x, float y) as a constructor.

### OkanTekman:

| OkanTekman |
| --- |
| |
| + OkanTekman(float x, float y) |

**Extends:** Creature

Figure 25: OkanTekman Class

It creates an OkanTekman object. It has OkanTekman(float x, float y) as a constructor.

**OkanTekman:**

| OkanTekman |
| --- |
| |
| + OkanTekman(float x, float y) |

**Extends:** Creature

Figure 26: WilliamSawyer Class

It creates a WilliamSawyer object. It has WilliamSawyer (float x, float y) as a constructor.

## 5.3 Specifying Contracts using OCL

### Vector2D:

1. **Context** Vector2D:: getX(): float

   **post:** return x

After the getX() method, x coordinate is returned.

2. **Context** Vector2D:: getY(): float

   **post:** return y

After the getY() method, y coordinate is returned.

3. **Context** Vector2D:: scale(float value): Vector2D

   **post:** return new Vector2D(x*value, y*value)

After the scale(float value): method, a new vector is created with x*value and y*value coordinates.

4. **Context** Vector2D:: sum(Vector2D vector): Vector2D

   **post:** return new Vector2D(vector.getX()+x, vector.getY()+y)

After the sum(Vector2D vector) method, x and y coordinates is summed up with the previous values.

**5. Context** Vector2D:: normalize(): void

**post:** x=x/magnitude();

y=y/magnitude()

After the normalize() method, x and y coordinates are divided by their magnitudes.

**6. Context** Vector2D:: normalized(): Vector2D

**post:** return new Vector2D(x/magnitude(),y/magnitude())

After the normalized() method, x and y coordinates are divided by their magnitudes and obtained in 1 Vector2D.

**7. Context** Vector2D:: magnitude(): float

**post:** return (float)Math.sqrt(x*x+y*y)

After the magnitude() method, root of x^2 + y^2 is returned.

**8. Context** Vector2D:: dotProduct(Vector2D vector):float

**post:** return this.x*vector.getX()+this.y*vector.getY()

After the dotProduct(Vector2D vector) method, dot product of x and y coordinates are obtained.

**9. Context** Vector2D:: getProjection(Vector2D vector): Vector2D

**post:** return vector.scale((float)(dotProduct(vector)/(Math.pow(vector.magnitude(), 2))))

After the getProjection (Vector2D vector) method, dot product of a Vector2D is divided by its magnitude square is returned.

**10. Context** Vector2D:: subtract(Vector2D vector): Vector2D

**post:** return new Vector2D(this.x-vector.getX(), this.y-vector.getY())

After the subtract (Vector2D vector) method, the coordinates difference of two distinct vectors are obtained.

**11. Context** Vector2D:: distance(Vector2D vector): float

**post:** return (float)(Math.sqrt(Math.pow(this.x-vector.getX(),2)+Math.pow(this.y-vector.getY(),2)))

After the distance (Vector2D vector) method, the distance between two point is obtained.

**12. Context** Vector2D:: crossProduct (Vector2D vector): float

**post:** return getX()*vector.getY()-vector.getX()*getY()

After the crossProduct (Vector2D vector) method, cross product of a Vector2D coordinates are obtained.

**13. Context** Vector2D:: setX(float x): void

**pre:** Vector2D vector = new Vector2D(x,y);

**post:** this.x=x;

x coordinate of Vector2D is setted to given value.

**14. Context** Vector2D:: setY(float y): void

**pre:** Vector2D vector = new Vector2D(x,y);

**post:** this.y=y;

y coordinate of Vector2D is setted to given value.

Beast:

**15. Context** Beast:: left(): void

**post:** position.setX(position.getX()-(float)1.5)

```
if(position.getX()<0)
  position.setX(0);
```

Beast will go to left if there is a place left to go.

**16. Context** Beast:: right(): void

**post:** position.setX(position.getX()+(float)1.5)

if(position.getX()>775)

position.setX(775)

Beast will go to right if there is a place left to go.

**17. Context** Beast:: translateX(float xDif): void

**post:** position.setX(position.getX()+xDif)

Beast will move in x coordinate with the xDif parameter.

**18. Context** Beast:: collided(GameObject g): void

**post:** if(g instanceof Creature){

beastHealth-=((Creature)g).getDamage();

System.out.println(beastHealth);

}

If a creature will hit to beast, beast's health will decrease by the creature's damage.

**19. Context** Beast:: getHealth (): int

**post:** return beastHealth

Returns the beast's health after the function call.

Creature:

**20. Context** Creature:: updatePosition(float timePassed): void

**post:** position=position.sum(new Vector2D(0,-0.3f).scale(timePassed))

It updates the creature's position according to the passed time.

GameObject:

**21. Context** GameObject:: getImage() : BufferedImage

**post:** return image

Returns to that GameObject's image after the function call.

**22. Context** GameObject:: getPosition(): Vector2D

**post:** return position

Returns to that GameObject's position after the function call.

**23. Context** GameObject:: doesCollide(GameObject g ): boolean

**post:** if(position.distance(g.getPosition())<collideRadius+g.getCollideRadius()){

return true;

}

return false;

Returns true if a collision has occurred between two GameObject's else return false.

Weapon:

**24. Context** Weapon:: updatePosition(float timePassed ) : void

**post:** position=position.sum(new Vector2D(0,0.5f).scale(timePassed))

Updates the current position of the weapon.

GameDynamicsEngine:

**25. Context** GameDynamicsEngine:: run () : void

**post:** rp.repaint()

It runs the game after the function call. It sets the game objects, updates their positions and repaint.

**26. Context** GameDynamicsEngine:: keyPressed(KeyEvent e): void

**post:** if (e.getKeyCode()== input)

do smt

It obtains the data that pushed button by the user. If it is 27, game will go to menu. If it is 32, shoot() function is called. If it is 39 or 68, go right. If it is 37 or 65 go left.

**27. Context** GameDynamicsEngine:: keyReleased(KeyEvent e): void

**post:** if(e.getKeyCode()==32

shootBlock=false;

If the keyPressed is space it will use the shoot() method.

**28. Context** GameDynamicsEngine:: shoot(): void

**post:** objects.add(new Bullet(beast.getPosition().getX() east.getPosition().getY()+50))

After this func

ion call, bullet class will shoot to the upwards at beast's current position.

**29. Context** GameDynamicsEngine::createEnemy(float probability):void

**post:** if(probability>Math.random()){

if(Math.random()<0.33f){

objects.add(new DavidDavenport((float)Math.random()*800,500));

}

After this function call, an enemy will be created with random coordinates.

**30. Context** GameDynamicsEngine:: checkCollision ():void

**post:** if(objects.get(i).doesCollide)

It will check whether GameObject's collide with each other or not.

## 6. Conclusions and Lessons Learned

Our project, Hungry Beast is an arcade-style score based game. Our team is tried to develop something different from what's already done in this game genre. In order to achieve that, we focused on the graphical part of the game and we came up with an idea which includes Bilkent instructors. We used our 3 instructors, Okan Tekman, David Davenport and William Sawyer, as invulnerable creatures in the game. By doing that, we aimed the student of Bilkent University and encouraged them to play our game. We decided to use PC as platform and delivered our product only for this platform. Although we did a Greenfield engineering practice, nearly whole platform games have the same rules during the design process. We cloned these processes and implemented our game as creative as possible.

We had 3 big important parts during our project. These are analysis report, and design report and implementation part. In our analysis report we did something new. In CS102 course, we did the simple version of this reporting part but still it is more difficult. We used our knowledge from the previous years and constructed our project roots. We decided on our requirements, scenarios, interface, class diagrams and important state chart and sequence diagrams. By doing the analysis part, we had some solid idea about what are we going to do.

Next, we constructed our design report. Before starting to our design report, we corrected our mistakes from the analysis part and reconsidered our aims again. In design report we stated our goals, subsystem decomposition, architectural patterns, hardware/software mapping, persistent data management, access control and security, global software control and boundary conditions. We detailed our classes and separated them as subsystem in order to make implementation easier.

In the last part of our project, we faced with bigger problems comparing to our reports. Dividing a project into separate parts is hard. In addition to that, we also need to

know our functions and variable names, in order to use them correctly and efficiently. Therefore, we needed a time schedule to arrange this complication. The entities of the project is written first. Then we continued on the gameplay of the project. Lastly, we ended up with the GUI.

To sum up, project was a challenging and educating experience for our group. We learned project development cycle, task assignment, and how to group up individual progress into a group work. At this part, the documentation clearance and design patterns helped us a lot in order to group up our code. We traded some of our design goals from our design report and saw that almost nothing is went as we planned. We faced with unexpected difficulties but our communication as a group helped us to overcome these problems.