Principles of Software Engineering

2805ICT: System and Software Design

MilestoneTwo

(INDIVIDUAL SUBMISSION)

Weight 10%

Ethel Beckett | S5125717

Sept 18, 2020

1 OBJECTIVES

1. Practice reporting the progress on a software development project.
2. Describe a testing plan and testing performed within the current progress.
3. Use the UML as a tool to describe and specify software designs and be able to attach semantic meaning to UML constructs by establishing relationships with the code the artifact models.
4. Discuss software design patterns meaningfully.
5. Review the potential evolutions that a software project may have regarding software architecture and software design paradigms.
6. Model dynamic behaviour of software systems using suitable UML notation.
7. Describe the progress concerning user interface design.
8. Review metrics regarding progress by reviewing historical date on version control system.
9. Discuss the properties of good software design including the nature and the role of associated documentation.
10. Create associated descriptions to document UML diagrams.

2 PROGRESS REPORT

I. Requirements

During the first milestone, a list of functional requirements has been achieved such as the player is able to initiate a game, the maze allows the presence of static and moving objects, the system allows Pacman to be controlled using the arrow keys on the keyboard, the system allows pelettes and energizer to be collected and disappear when eaten by Pacman, they system should display the current highest score and the 1up score of the player and the 1up score should return to 0 when a new set of game start. The milestone1 process was focused on implementing the foundation of Pacman application and milestone 2 will be focusing on applying other high priority functions that were not implemented in milestone one.

Below is the list of functional requirements for the milestone 2 processes that have been completed according to their priority. It also includes other functional requirements that have only been partially completed which is stated in the description.

Table 1 Functional Requirements

Identifier	Priority	Requirement	Description
BA-RE-2	5	Completed	Exit game
PA-RE-13	5	Completed	Game over (dies 3x)
GH-RE-17	5	Completed	ghost disappear when eaten
PA-RE-12	5	Completed	Pacman respawn
GH-RE-18	5	Completed	Ghost respawn
LE-RE-28	5	Completed	Navigate main menu
LI-RE-24	5	Completed	Pacman lives (3)
LI-RE-25	5	Completed	Removes life when pacman dies

GH-RE-15	5	Incomplete	Partially implemented. Only 2 states have been implemented such as chase Pacman and scared ghost
BA-RE-2	4	Completed	Pacman dies on ghost collision
PA-RE-11	4	Completed	Pacman eats ghost
LE-RE-28	4	Completed	Navigate menu using mouse
SC-RE-23	4	Completed	Realtime score
BA-RE-4	3	Completed	Implemented but altered mute sound to volume slider
BA-RE-3	3	Completed	Background music
BA-RE-5	3	Completed	Play sound effects
SC-RE-22	2	Incomplete	This has been partially implemented where 1up score is working correctly then add it to the array however the highest score can only store temporary objects and clears the new data when the game is restarted.
SC-RE-21	2	completed	Reset score
SC-RE-22	2	Incomplete	This has been partially implemented. It allows to display the static (fruit) object on first level however there is no consecutive level implemented at this time.
LE-RE-26	2	Incomplete	Partially implemented, only one level is on display because there is no other level applied.

II. Use Cases

The use cases have been implemented in milestone 2 however few use cases was incomplete or currently is still in the development process such as:

a. Applying scatter state where each ghost has their own pattern by going into each corner of the puzzle when Pacman collects the energizer;

b. Recording the highscore into JSON file was initially implemented this function records the data it triggers it to also download a copy to the user. In order to update the JSON file, this downloaded data is required to be injected back to the project file to be read again and get updated. One way of addressing this issue is to learn or see if p5.js can be saved without having the user to download the file.

c. The third use case is new stages for the user when they complete the first stage. This has not been implemented and it is still in the development process.

III. Software Architecture and tools

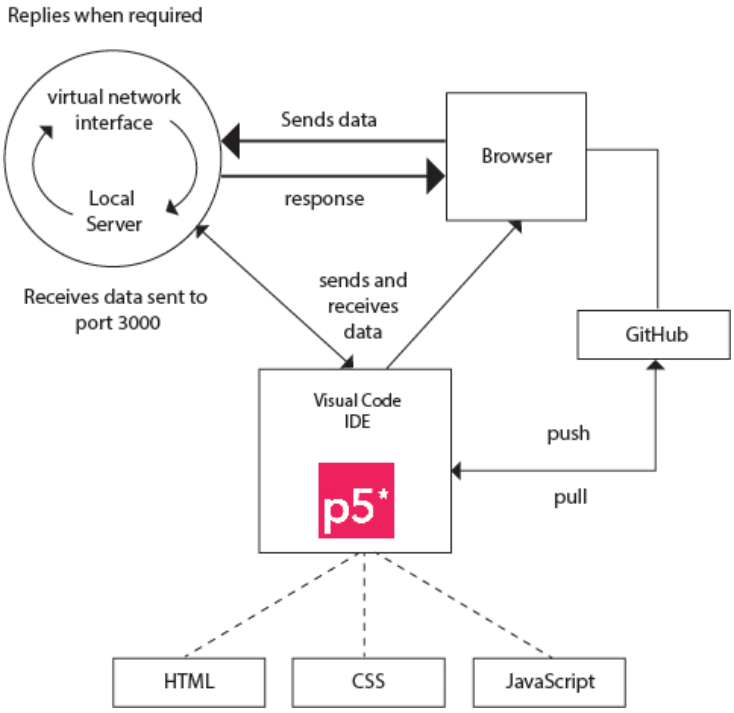


Figure 1 illustrates the software architecture of the application.

P5.js utilises a local server to be able to connect and display results to the browser to allow the cross-origin requests. The p5.js is utilized via the IDE which also connects to the Github repository where I can push and pull my commits. This can be accessed via browser to clone or to share my repository. The connections between the browser, server, and IDE are asynchronous. A function needs to be completed before the next one is processed. When a request is triggered it sends the data to the port then connects to the virtual network and displays the

response to the browser. When there is an error in the function, the application will stop executing until it is fixed.

The tools used in the application are the following:

- HTML

  Allows to read p5.js to access and be displayed in the browser.

- CSS

  CSS is not particularly used in the application however this incorporated from the milestone one for documentation processes.

- JavaScript

  The application is purely based on p5.js which is part of the JavaScript library that has drawing capabilities. It comprises classes, objects, and functions. The classes are used to create objects, for example, field, Pacman, and ghosts' classes. The objects consist of state and behavior which can be passed between functions and functions are a set of instructions that provide input and output.

- Github (Version Control)

  This tool was used to back-up my files. It allows me to create multiple branches for each task I have created where I can pull anytime should I need another copy or revert back to the older version. It also helps to share my repository to anyone should I need to collaborate with a team.

- Visual Studio Code

  Is a source-code editor preference because it makes writing and debugging codes easier. It also allows me to utilise the terminal within the IDE instead of having a separate terminal window such as brackets. It makes it easier to run my node server without opening a new window.

- Express Node

  I used express node to simply connect with my local server to prevent cross-origin restrictions. CORS has restrictions on requesting resources when a request is coming from a different server for unauthorized access. By

configuring my server, it allows me to send and receive a request to the browser.

- Papyrus

  Papyrus used for all UML diagrams.

IV. Summary of Design

Purpose and Goal

The goal of the design is to re-create Pacman as close I can and implement the same functionality as the original to provide a nostalgic feel to the user. The design is aimed to be friendly, exciting, and good color coordination.

Wireframe

Figure 2 shows the prototype I created using Illustrator. I created the prototype by creating small boxes to represent the brick and recreate the maze. It helped structure my maze by knowing the size of canvas I need to recreate the maze in p5. I used this prototype as my guide when creating each brick. The dots represent the placement of the ghosts, the energizer, and Pacman. The maze is constructed using the constructor and using alphanumeric to represent each object.

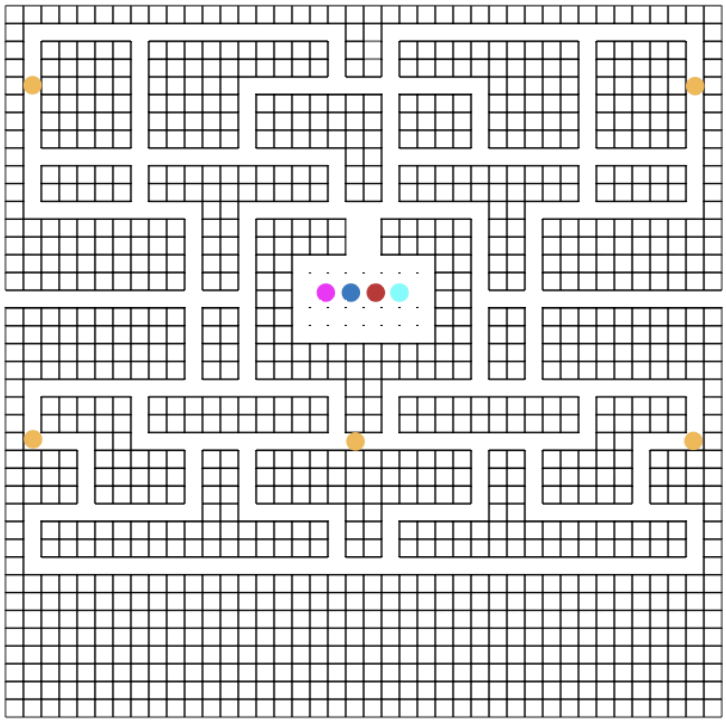


Figure 2 Prototype

Stakeholders / Audience

- Users / Gamers

  People who will use the application
- Developers

  The developer of the application
- Tester

  People who will test the application
- Company stakeholders

  “Hypothetically” the company stakeholders

Classes

There are multiple classes in the application and these classes are used to create my objects and within the class, it consists of the method. These methods can be called on the class itself. For example, Class Pacman --- within the class Pacman there are methods such as “this.show”, “this.move”, “this.collission”, and so on. These methods can be called by calling the class itself, for example, “pacman.show” or “pacman.move”. This way, it prevents me from creating new codes whenever I need them because it allows for the code to be reused multiple times.

Dynamic State

The arrow keys on the keyboard serve as the main control for Pacman. Initially, Pacman is chased by the ghosts (Main State). If Pacman collides with a pelette or fruit, the pelettes or fruit is removed with a sound effect.
If Pacman eats the energizer, the ghost changes its state to blue and moves slower and when Pacman collides with the ghost, the ghost disappears with sound effect then respawns to their starting position otherwise, pacman dies with sound effect, with less life and then respawn to its starting position.
When Pacman dies three (3) times consecutively, it is game over and the screen reloads but if Pacman is able to collect all the pelettes then Pacman wins. The state changes when a function is triggered, and they work asynchronously where

a set of instructions needs to be completed before the next function is processed.

Subsystem Decomposition

Figure 3 illustrates the subsystem decomposition of Pacman application to a manageable subsystem by grouping its classes together to maximize the use of cohesion and minimize the coupling. The application frameworks house the whole application, the control serves as the input and the settings for the user interface. The model is the set of codes that are executed when it is initialized. The view sets the location for the simulation, visualization, and display as a user interface. The process allows the subsystem to be developed individually and improves maintainability of the application.

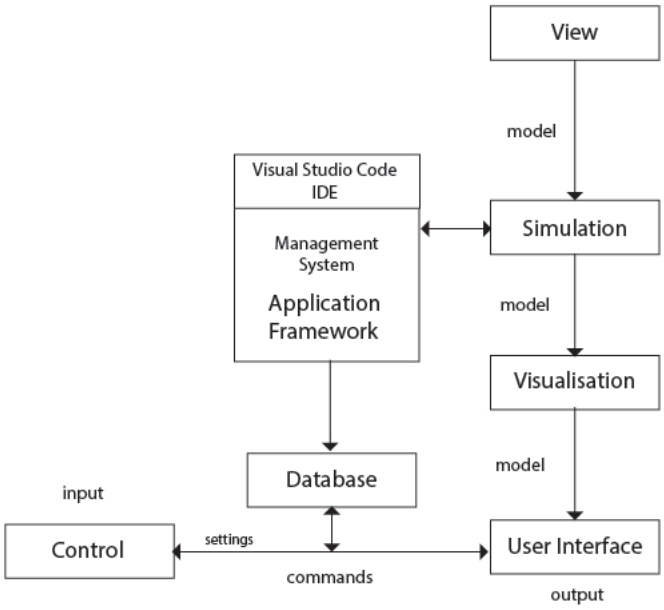


Figure 3 System Decomposition

V. Persistent Data Management, access control and security and user interface

The Pacman application is relatively small, the images and sound are also stored and converted to a smaller file, so it won't take too much memory. Models is composed of instruction algorithms that can be accessed multiple times; this improves the efficiency in the memory. When the game is initialized the application preloads all the assets for quick access so that when these assets are called later on the user does not have to wait for it to load because it has been preloaded. Once the game ends, everything is reset back to empty until it is reloaded again. When it comes to security, the application does not store

permanent data on the system nor collect any private information to play the game. Eventually to secure the codes a server/client model can synchronize packages that will allow the user to download the latest available packages for its dependency. The interface varies in different frame rates depending on Pacman's state. In chase mode the ghosts move faster and when on the scared mode the ghost's movement goes slower. Since we can reuse our code, we can call a function to turn on and turn off when it is needed.

VI. Testing

Component testing

Each component of the application has been developed in smaller tasks to ensure it runs and tested. For example, "class Pacman" a construction was created and its methods.

this.show method was initially created and run until the Pacman image is called in the "drawplayscreen function()". This.frame has also been set to play the animation sprites.

```
this.show = function() {
     image(pacmanImg, this.x , this.y , 32 , 32 , 32*this.frame++ ,
32*this.direction  ,32  , 32);
     this.frame = (this.frame === 6)?0:this.frame;
}
```

Once Pacman is displayed on the screen, I moved on to the direct method and control function. If an arrow key is pressed then the pacman.move class method will be called to initiate the instructions. For example, when the right key arrow is pressed then move to this direction === 0 which is the right key. This is similar to other keys or directions except the direction is different (0 right, 1 down, 2 left, 3 up directions).

```
if(this.direction === 0) {
     this.x += 32;
}
```

Each component has been tested the same way as the above. By using classes, it allows me to set codes that are related to each other and reuse them multiple times and maximizing the use of high cohesion.

Integration testing

The integration testing is testing the whole application from initiating the game and ending the game. It is tested how all the functions and classes work together as a whole. The test has passed satisfactorily as the application allows initiating a game, navigate between the menu, Pacman moves, eats the pelettes, the fruit, the state changes to scared when Pacman eats the energizer and capable of eating the ghost, the ghost respawns to its original position, Pacman dies when collided with a ghost on normal states and when Pacman dies three (3) times consecutively the game over screen appears with the new high score or does not record new score if it is less than existing high score then reloads the window to go back to the main menu. Overall, the functions are all working well together except there are functions that are still lacking and still in development, for example, expiry time when it is on scared state and different levels of Pacman. In addition, the node express server is also working.

Use Case for testing:

Project	Pacman Game
Module:	Field Environment
Requirement:	BA-RE-1 – BA-RE-5, MA-RE-6, MA-RE-7, PA-RE9 – PA-RE-13, GH-RE-15, GH-RE-17,GH-RE-18, LI-RE-24, LI-RE-25, LI-RE-26, LE-RE-28
Test case ID:	Test 1
Test Objective:	GUI testing and play pacman
Test Date and Time:	16 September 2020, 9:15pm

No	Steps	Data	Expected Results	Actual Result

1	Launch Game and End game	Click buttons	Launch and ability to end the game	As expected
2	Launch game and move pacman	Controls	Pacman moves	As expected
3	Launch game, sounds, turns on, ghost moves	Chase state on, sounds plays once	Ghost moves, sound plays	As expected
4	Launch game, collect pelettes, energizer, eat ghost and sound effects	Control on keypressed	Sound effects plays. Pelettes, energizer and ghost disappear when eaten by pacman,	As expected
5	Launch game, collide pacman with ghost and sound effects	Control on keypressed	Pacman dies on collision with ghost and back to original position with sound effects	As expected
6	Lauch game and collide pacman with ghost three	Control on key pressed	Pacman died on third try and game over screen displayed	As expected

	times and game over			

VII. Version Control

Using version control was very helpful to create back up for my work because I can create new branches to my repository and can revert back anytime to a particular commit if I need to. I can also share my repository anytime to anyone to collaborate if it's required. Github also helps me tracks the changes to my files.

My repository consists of 6 branches

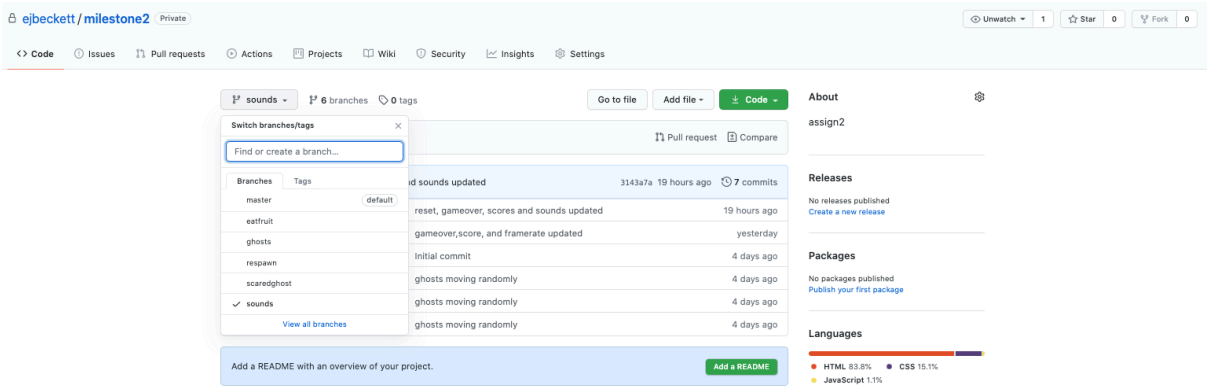


Figure 4 Repository and branches

- **Master**

  Is the starting point of my application (milestone1). From here I added new branches. This way I know what the functions that I have built from my milestone1 where I can revert back to my original codes if I need to.

- **Ghosts branch**

  The ghost branch consists of updates that are relevant to the ghosts.

- **Scaredghost branch**

  New branch for change of state of the ghost, re-spawning and updating the score

- **Respawn branch**

  Updating the ghost respawn, pacman dying and respawn back to its original position.

- **Eatfruit**

  Updates for pacman eating the fruit, change of framerates, game over and updating score.

- **Sounds**

  Updates on applying the music background before pacman starts and the sound effects.

The languages used are displayed for HTML, CSS, and JavaScript. The HTML and CSS are relatively small but because I have also uploaded my milestone 1 which has my html documentation it also calculates how much HTML and CSS were used.

My repository is currently set to private and it requires an invitation to access my repository. I have also set up a .gitignore to ignore nodes when I commit since this can be rebuild using npm install, it does not require to be uploaded to my repository and takes up unnecessary space.

Repository logs:

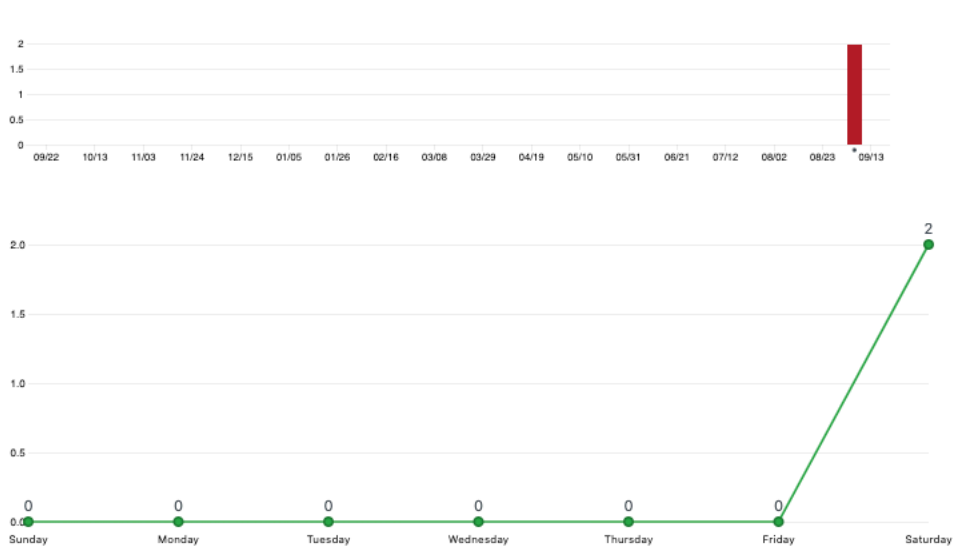


Figure 6 Commit Size

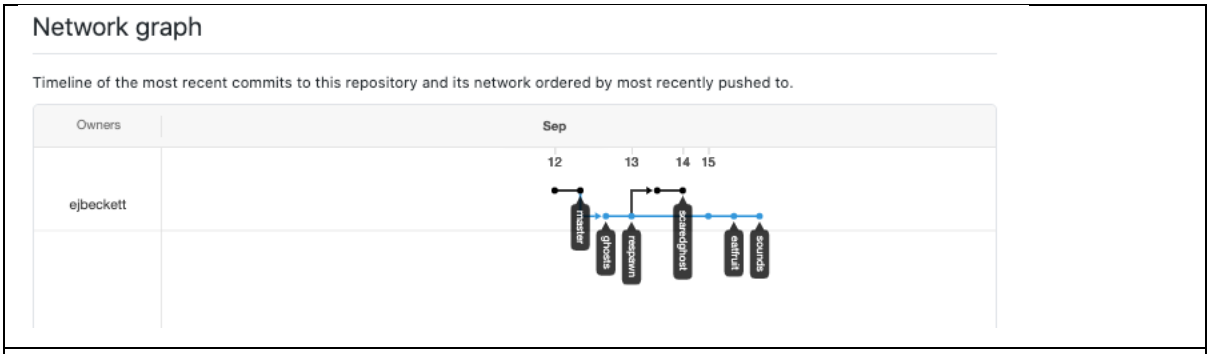


Figure 7 Network graph

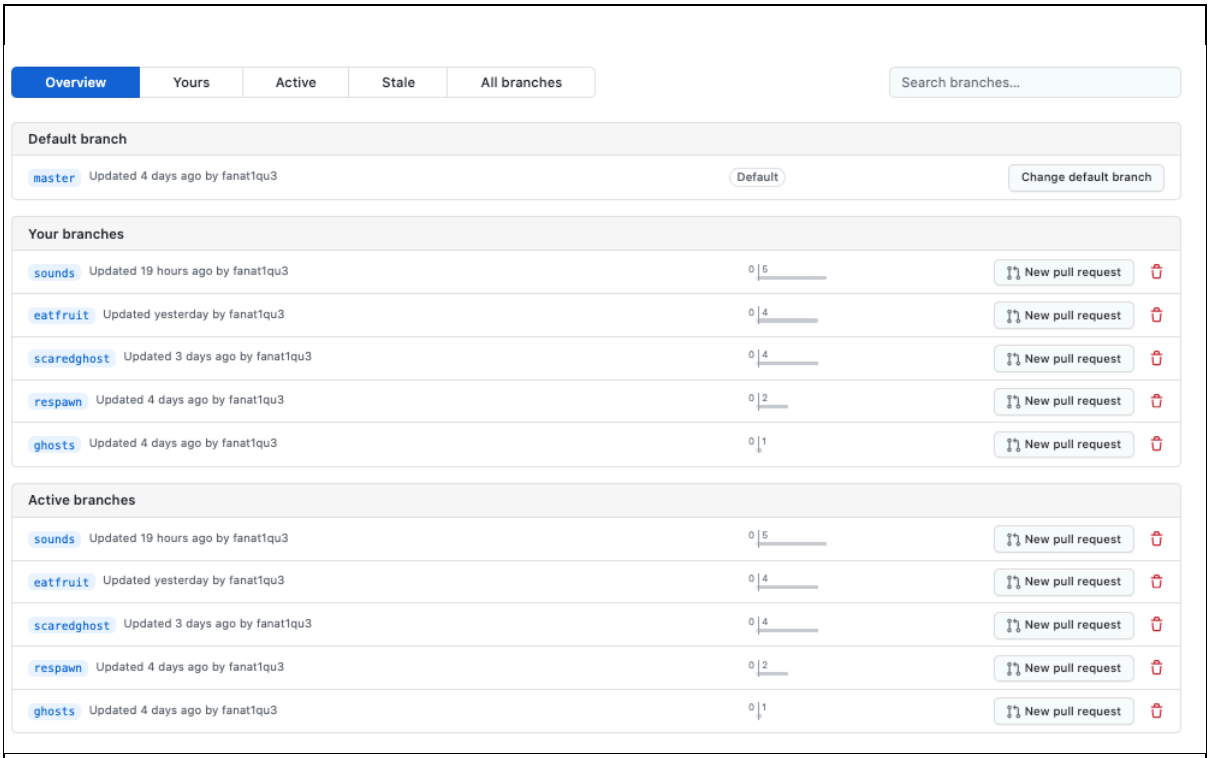


Figure 8 Overview

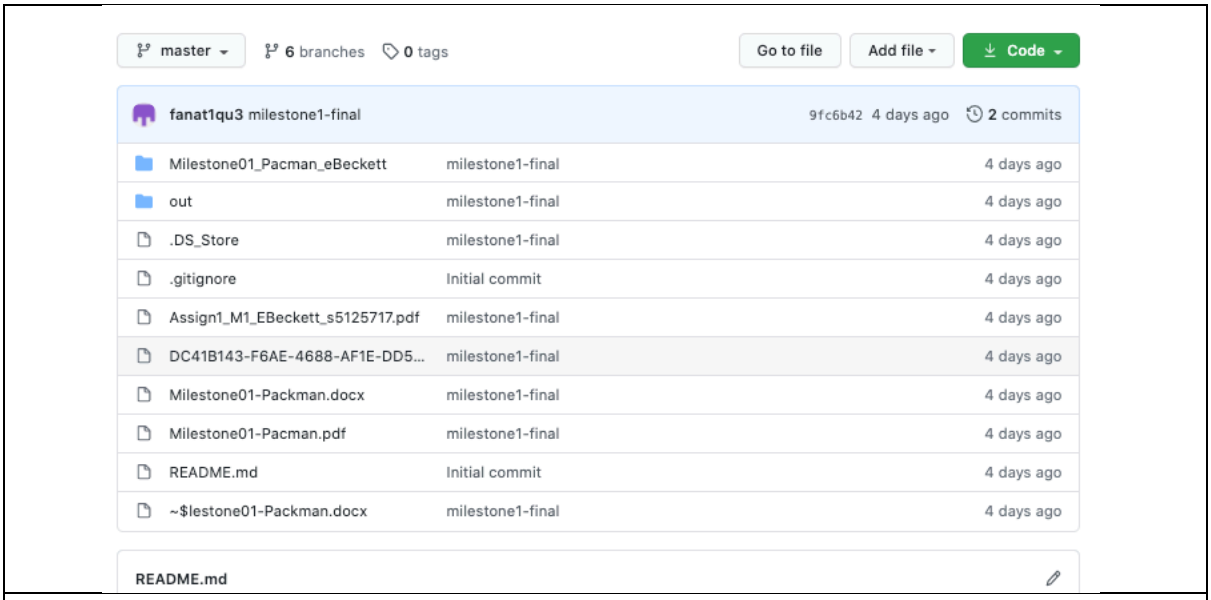


Figure 9 Master

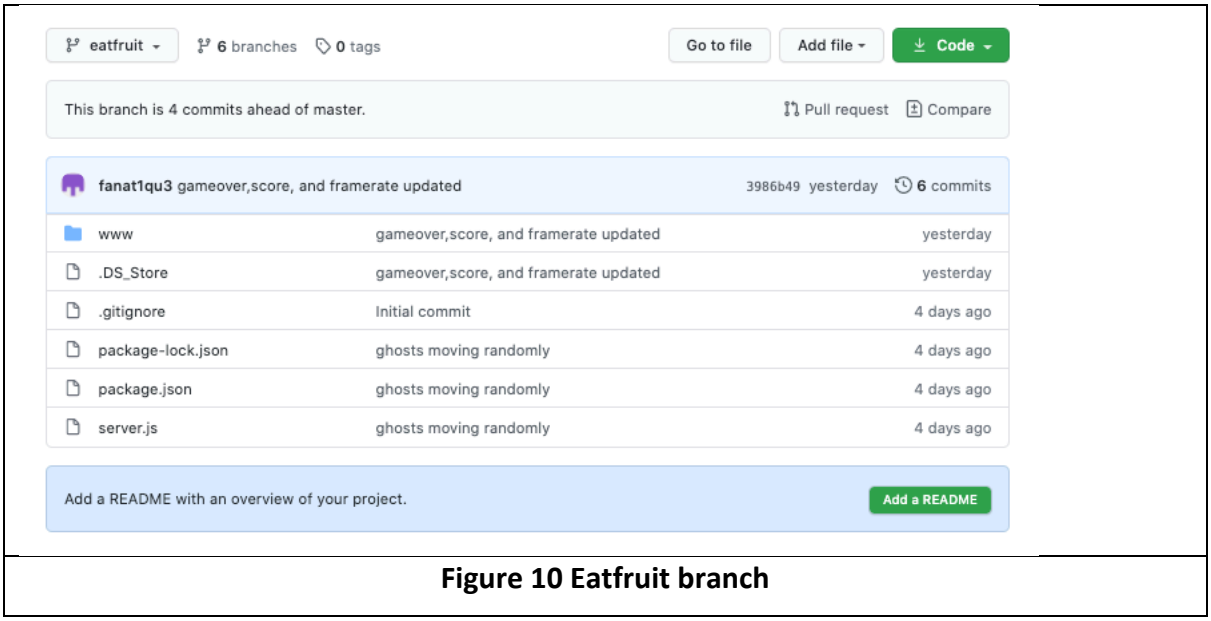


Figure 10 Eatfruit branch

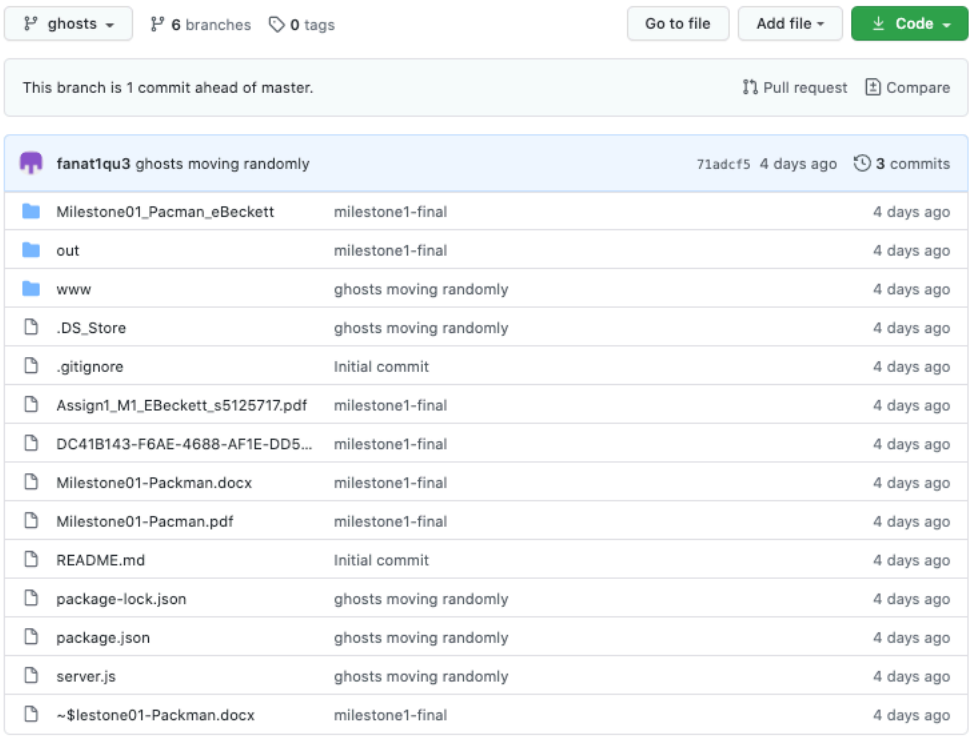


Figure 11 Ghost branch

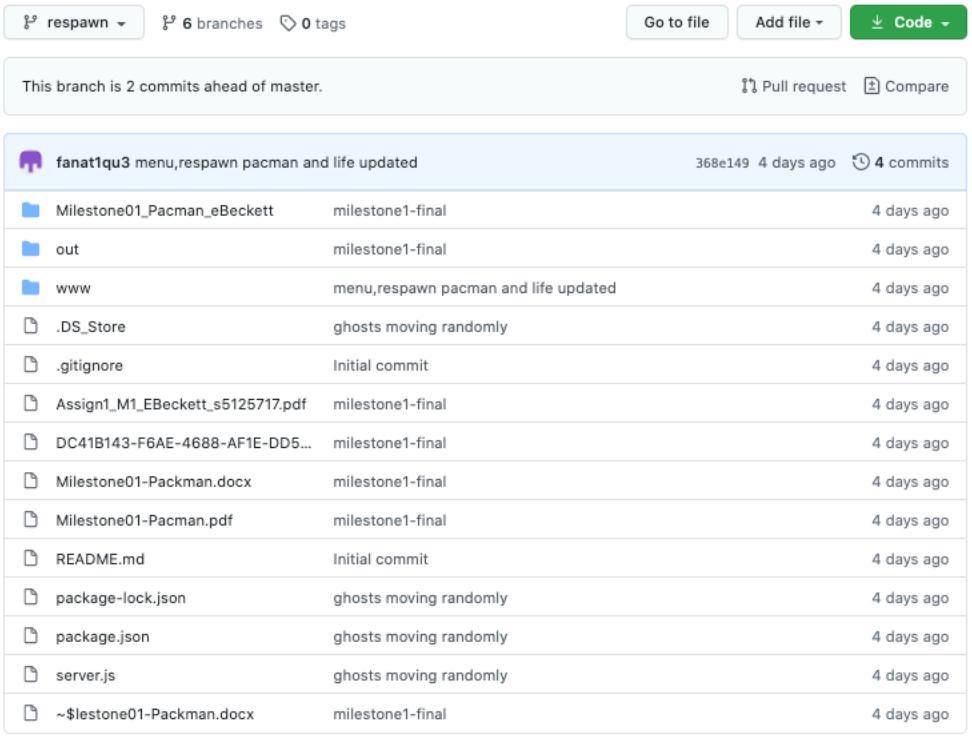


Figure 12 Respawn branch

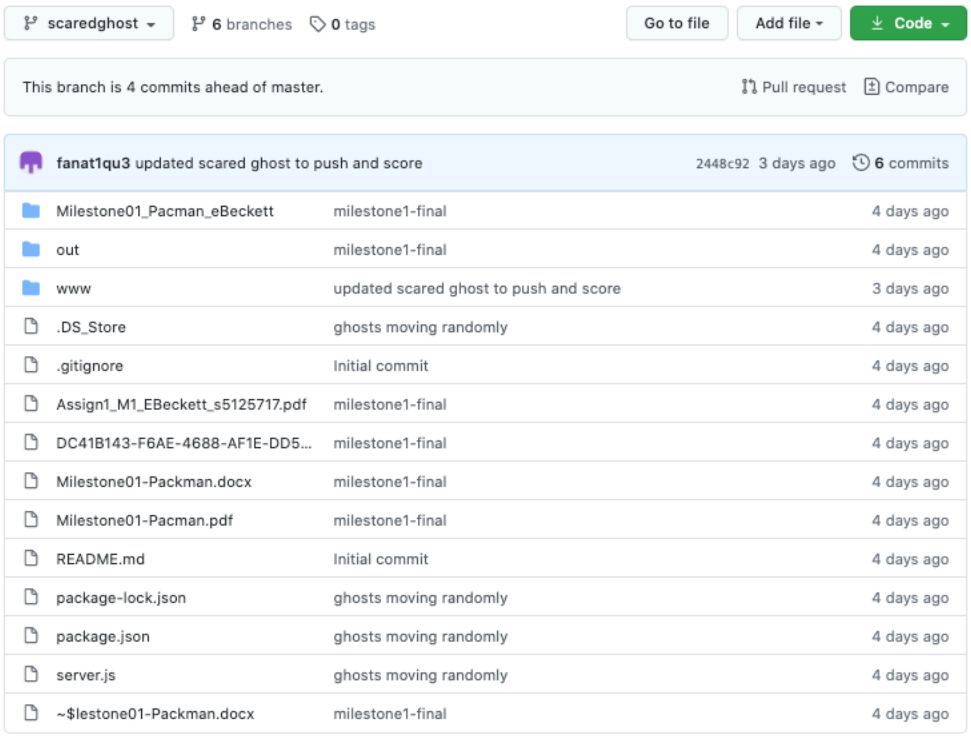


Figure 13 Scaredghost branch

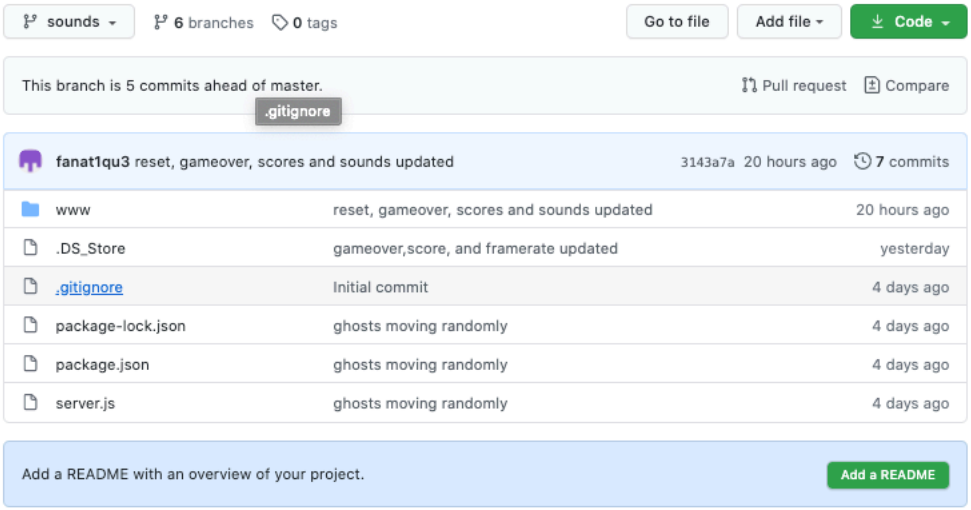


Figure 14 Sounds branch

3 Putting UML to use

I. Dynamic Modelling

a. Interaction diagram

i. Sequence Diagram

Figure 15 illustrates the sequence diagram when the player initiates a game the playScreen function is called to construct all objects and will be displayed. The chaseMode state turns on where ghosts start to move. When Pacman dies, it checks if Pacman have enough lives and if not then it is game over. The score gets recorded.

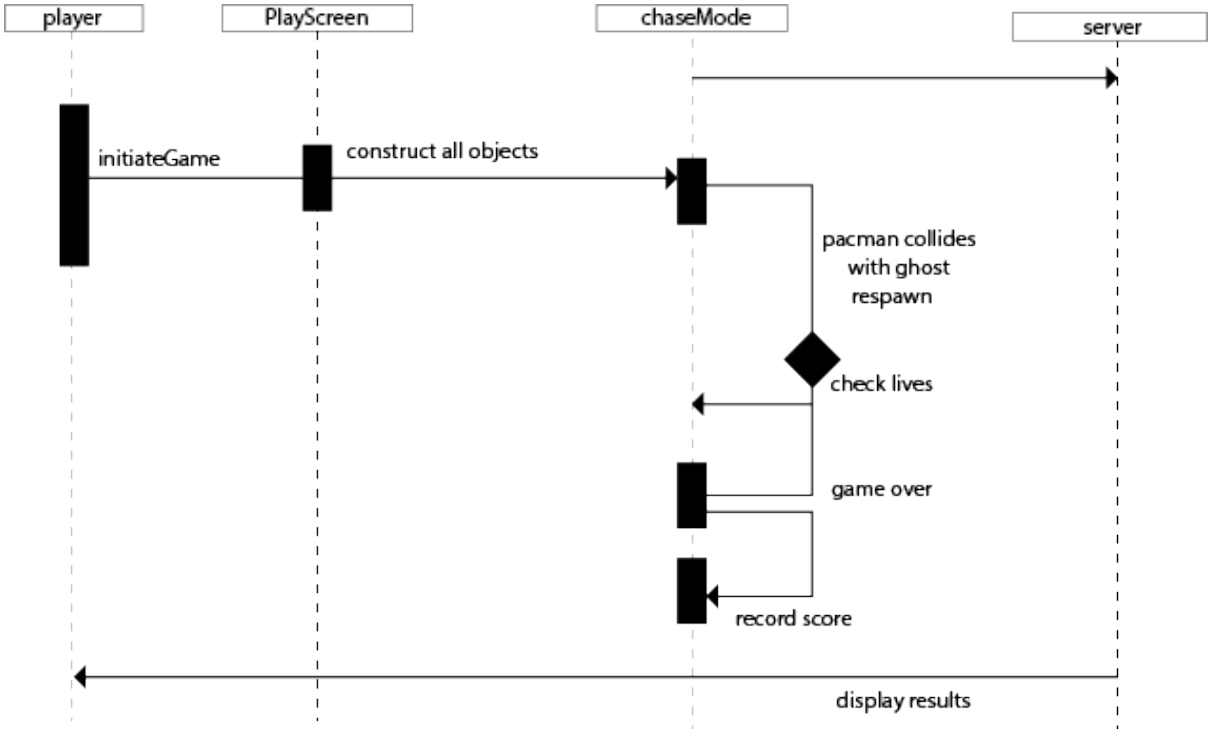


Figure 15 Sequence Diagram

Codes relative to the sequence diagram:

```
this.colission = function(ghost) {
            var distance = dist(this.x, this.y, ghost.gX, ghost.gY);
                if(distance < this.radius + ghost.radius)
                    return true;
                return false;
        }
for(var i=0; i < ghosts.length; i++) {
```

```
        if(pacman.colission(ghosts[i])) {
            if(ghosts[i].isScared === true) {
                ghosts[i].isScared = false;
                eatGhostSound.play();
                ghosts.splice(i,1);
                score = score + 200;
                ghosts.push(new Ghost(32*12,32*10, ghosts[i].img));
            }
        for(var j=0; j < lives.length; j++) {
            if(pacman.colission(ghosts[i])) {
                deathSound.play();
                lives.splice(j,1);
                pacman.respawn();
                for(var k=0; k < ghosts.length; k++) {
                    ghosts[k].ghostRespawn();
                }
                if(lives.length < 1) {
                    saveScore();
                }
            }
        }

        }
    }
```

ii. Collaboration diagram

Figure 16 displays the collaboration diagram in a number of sequences according to the flow of operation when the button is clicked. From Main Menu, users can switch screen by clicking the button. When the start game button is clicked it calls the drawPlayScreen() function to display all the objects and calls the other functions associated such as the instance and methods within each class like Pacman, ghosts, pelettes, lives, and energizer. If user clicked the highscore button, the highscoreList is displayed on the screen and also allows them to return back to the main menu. The exit game button terminates the program and returns the user back to the main menu. Each button that is clicked, it hides the other button and shows the relevant buttons relative to the screen.

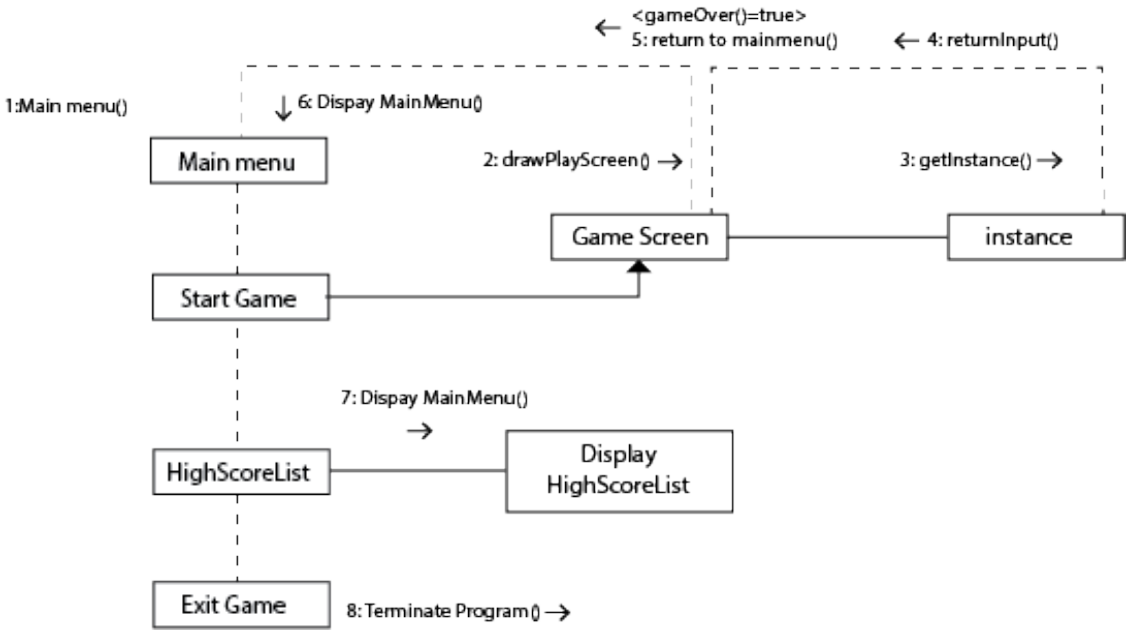


Figure 16 Collaboration diagram

Codes relative to the collaboration diagram

```
//button variables CONST applied as variables are
const MAIN_MENU = 0;
const PLAY = 1;
const HIGH_SCORE = 2;
const END_GAME = 3;
//button variable functions  for different screens events
let currentScreen = MAIN_MENU;
let playButton1, playButton2, playButton3, playButton4;

function setup() {
//creates buttons to navigate through different screens
     playButton1 = createButton('MAIN MENU');
     playButton1.position(400, height/2 - playButton1.size(100, 40).width);
     playButton1.mouseClicked(playMenuButton);

     playButton2 = createButton('PLAY');
     playButton2.position(400, height/2 + 50 - playButton2.size(100, 40).width);
     playButton2.mouseClicked(playButtonGame);


     playButton3 = createButton('HIGHSCORE');
     playButton3.position(400, height/2 + 100 - playButton3.size(100, 40).width);
     playButton3.mouseClicked(playpHighScore);
```

```
     playButton4 = createButton('END GAME');
     playButton4.position(400, height/2 + 470 - playButton4.size(100, 40).width);
     playButton4.mouseClicked(endGame);

//functions used to show and hide buttons as per the screen events
    function playMenuButton() {
        currentScreen = MAIN_MENU;
        playButton1.show();
        playButton2.show();
        playButton3.show();
        playButton4.hide();
    }
    function playButtonGame(){
        currentScreen = PLAY;
        playButton1.hide();
        playButton2.hide();
        playButton3.hide();
        playButton4.show();
    }
    function playpHighScore() {
        currentScreen = HIGH_SCORE;
        playButton1.show();
        playButton1.position(400, height/2 + 400 - playButton1.size(100,
40).width);
        playButton2.hide();
        playButton3.hide();
        playButton4.hide();
    }
    function endGame() {
        playButton4.mouseClicked(window.location.reload());
    }

switch(currentScreen) {
            case MAIN_MENU:
                drawMainMenuScreen();
                break;
            case PLAY:
                drawPlayScreen();
                break;
            case HIGH_SCORE:
                drawHighScore();
                break;
        }
```

b. State-based diagrams

i. The state machine that describes the flow of states for a given class in response to the outside stimuli. This diagram illustrates how the button is initiated by showing and hiding each button when they are inactive. When the drawplayScreen() button is clicked it switches to the screen then hides the buttons from main menu and when highscore button is clicked the switch is initiated and changes screen then shows the relative button and hides the start game button.

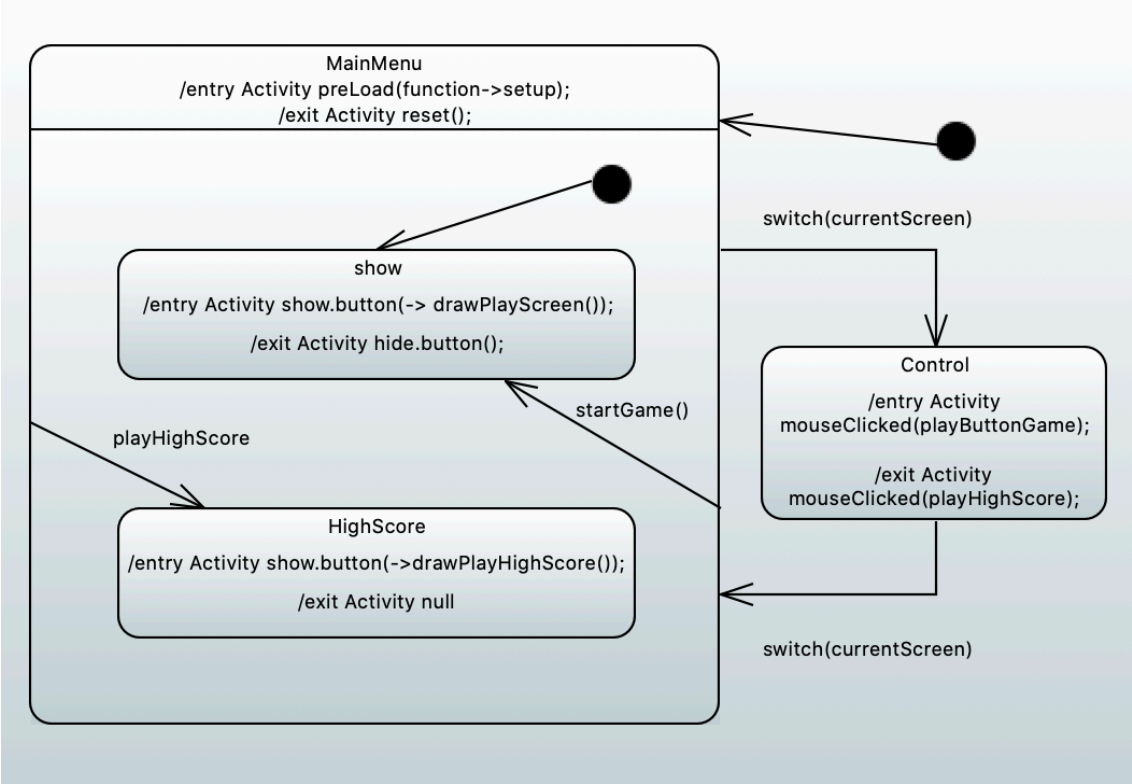


Figure 17: State Machine that describes the flow of states

Codes relative to the state machine that describe the flow of states

```
const MAIN_MENU = 0;
const PLAY = 1;
const HIGH_SCORE = 2;

    playButton1 = createButton('MAIN MENU');
    playButton1.position(400, height/2 - playButton1.size(100, 40).width);
    playButton1.mouseClicked(playMenuButton);

    playButton2 = createButton('PLAY');
    playButton2.position(400, height/2 + 50 - playButton2.size(100, 40).width);
```

```
     playButton2.mouseClicked(playButtonGame);


     playButton3 = createButton('HIGHSCORE');
     playButton3.position(400, height/2 + 100 - playButton3.size(100, 40).width);
     playButton3.mouseClicked(playpHighScore);

function playButtonGame(){
        currentScreen = PLAY;
        playButton1.hide();
        playButton2.hide();
        playButton3.hide();
        playButton4.show();

    }
    function playpHighScore() {
        currentScreen = HIGH_SCORE;
        playButton1.show();
        playButton1.position(400, height/2 + 400 - playButton1.size(100,
40).width);
        playButton2.hide();
        playButton3.hide();
        playButton4.hide();
    }

        switch(currentScreen) {
            case MAIN_MENU:
                drawMainMenuScreen();
                break;
            case PLAY:
                drawPlayScreen();
                break;
            case HIGH_SCORE:
                drawHighScore();
                break;
        }
```

ii. Activity State Diagram

Figure 18 demonstrates the activity state diagram of changing the state of the ghosts. Initially the states start as the ghost in chaseState() with conditions that when Pacman eats the energizer the scaredState() is turned on to True and allows pacman to eat the ghosts and when timer is expired the scaredState() turns off as False and returns to chaseState(). Please note that the timer was one of the requirements that was not applied and is still in current development process, hence does not reflect on the code.

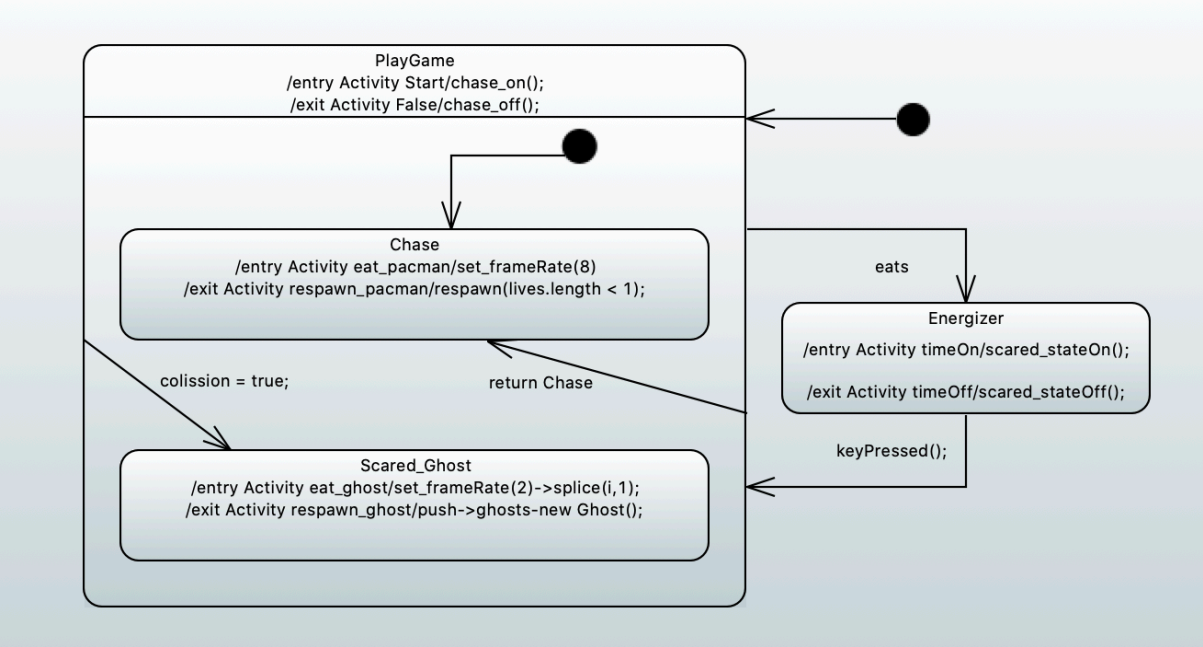


Figure 18 Activity Diagram

Code relative to the activity diagram

```
for(var i=0; i < ghosts.length; i++) {
        if(pacman.colission(ghosts[i])) {
            if(ghosts[i].isScared === true) {
                ghosts[i].isScared = false;
                eatGhostSound.play();
                ghosts.splice(i,1);
                score = score + 200;
                ghosts.push(new Ghost(32*12,32*10, ghosts[i].img));
            }
        for(var j=0; j < lives.length; j++) {
            if(pacman.colission(ghosts[i])) {
```

```
                deathSound.play();
                lives.splice(j,1);
                pacman.respawn();
                for(var k=0; k < ghosts.length; k++) {
                    ghosts[k].ghostRespawn();
                }
                if(lives.length < 1) {
                    saveScore();
                }
            }
        }

        }
    }

for(var i = 0; i < energizers.length; i++) {
        energizers[i].show();
        if(pacman.energize(energizers[i])) {
            eatFruitSound.play();
            energizers.splice(i,1)
            score = score + 50;
            ghostScared();
        }
    }

function ghostScared() {
        for(var i = 0; i < ghosts.length; i++)
            ghosts[i].isScared = true;
    }
function keyPressed() {
    if(keyCode === RIGHT_ARROW) {
        if(field.theField[pacman.y / 32][pacman.x / 32 + 1] !== '*')
        pacman.move(0);
    }
    else if (keyCode === DOWN_ARROW) {
        if(field.theField[pacman.y / 32 + 1][pacman.x / 32] !== '*')
        pacman.move(1);
    }
    else if (keyCode === LEFT_ARROW) {
        if(field.theField[pacman.y / 32][pacman.x / 32 - 1] !== '*')
        pacman.move(2);
    }
    else if (keyCode === UP_ARROW) {
        if(field.theField[pacman.y/ 32 - 1][pacman.x/ 32] !== '*')
```

```
            pacman.move(3);
    }
}

class Pacman {
    constructor(x,y) {
        this.x = x;
        this.y = y;
        this.frame = 0;
        this.direction = 0;
        this.radius = 5;
        this.xspeed = 1;
        this.yspeed = 0;

        this.show = function() {
            image(pacmanImg, this.x , this.y , 32 , 32 , 32*this.frame++ ,
32*this.direction  ,32  , 32);
            this.frame = (this.frame === 6)?0:this.frame;
        }
        /** setting control direction movement
        * - ----- x ----- +
        *  |       3     |
        *  | 2     1   0 |
        *   ------------
        *         +
        */
        this.move = function(m) {
            this.direction = m;
            //right
            if(this.direction === 0) {
                this.x += 32;

            }
            //down
            if(this.direction === 1) {
                this.y += 32;

            }
            //left
            if(this.direction === 2) {
                this.x -= 32;

            //up
            }
```

```
        if(this.direction === 3) {
            this.y -= 32;

        }

        if(this.x < 0) {
            this.x = width -32;
        }
        if(this.x >= width) {
            this.x =0;
        }
    }

    /**Pacman-pelette distance check in 10 radius */
    this.eat = function(pelette) {
        var distance = dist(this.x, this.y, pelette.pX, pelette.pY);
            if(distance < this.radius + pelette.radius)
                return true;
            return false;
    }
    /**Pacman-energizer distance check in 10 radius */
    this.energize = function(energizer) {
        var distance = dist(this.x, this.y, energizer.eX, energizer.eY);
            if(distance < this.radius + energizer.radius)
                return true;
            return false;
    }

    this.colission = function(ghost) {
        var distance = dist(this.x, this.y, ghost.gX, ghost.gY);
            if(distance < this.radius + ghost.radius)
                return true;
            return false;
    }

    this.eatFruit = function(fruit) {
        var distance = dist(this.x, this.y, fruit.fX, fruit.fY);
            if(distance < this.radius + fruit.radius)
                return true;
            return false;
    }

    this.respawn = function() {
        this.x = 448;
```

```
            this.y = 576;
        }
    }
}

function Ghost(x,y,img,theField) {
    this.gX = x;
    this.gY = y;
    console.log(this.gX,this.gY);
    this.img = img;
    this.frame = 0;
    this.direction = 0;
    this.radius = 5;
    this.movement= true;
    this.isScared = false;


    this.show = function() {
        if(this.isScared === false)
            image(img, this.gX, this.gY, 32, 32, 0,0,32,32)
        else {
            image(weakImg, this.gX, this.gY, 32, 32, 0,0,32,32)
        }
        // this.direction = (this.direction === 3)?0:this.direction;

    }
    /**d = position */
    this.ghostMove = function(tiles) {
        if(this.movement === false) {
            var d = floor(random(4));
            this.direction = d;
        }
        var lastx = this.gX;
        var lasty = this.gY;
        if(this.direction === 0) {
            this.gX += 32;
        }
        if(this.direction === 1) {
            this.gY += 32;
        }
        if(this.direction === 2) {
            this.gX -= 32;
        }
        if(this.direction === 3) {
```

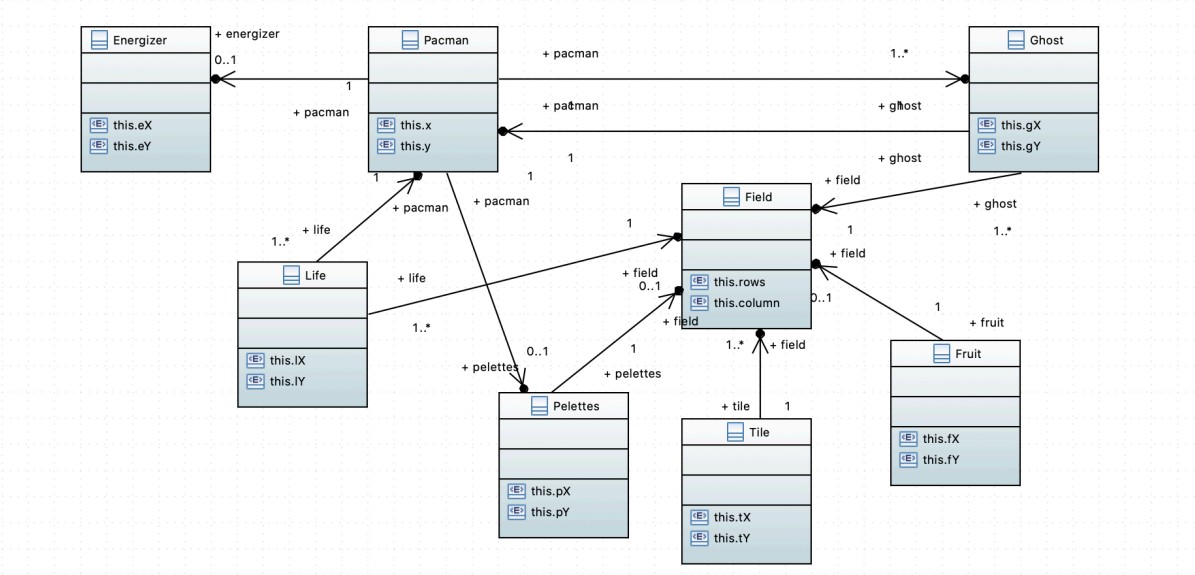
```
            this.gY -= 32;
        }
        for(var i = 0; i < tiles.length; i++) {
            if(this.colission(tiles[i])) {
                this.gX = lastx;
                this.gY = lasty;
                this.movement = false;
                this.ghostMove(tiles);
            }
            else {
                this.movement = true;
            }
        }
        if(this.gX < 0)
            this.gX = width -32;
        if(this.gX >= width)
            this.gX = 0;

    }
    this.colission = function(tile) {
        var distance = dist(this.gX, this.gY, tile.tX, tile.tY);
            if(distance < this.radius + tile.radius)
                return true;
            return false;
    }
    this.ghostRespawn = function() {
        // this.gX = 448;
        // this.gY = 320;
        this.gX = x;
        this.gY = y;
    }
}
```

II. Structural (static) Modelling

a. A class diagram

Figure 19 demonstrates the class diagram for Pacman classes on how they are related to each other. It also displays their cardinality and modality.

Figure 19 Class Diagram



III. For the use of Software Patterns

a. Figure 20 illustrates one of the software patterns illustrated by some existing code in the current version. The ghost movement both for chaseState and scaredState uses the same codes for each state.

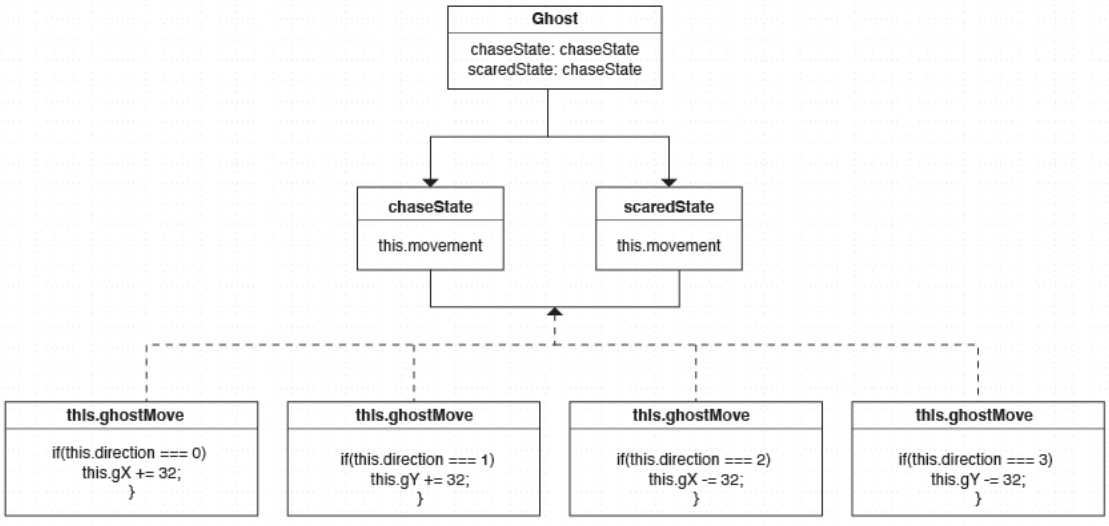


Figure 20 Software patterns

Codes relative to the Software patterns

```
/**d = position */
    this.ghostMove = function(tiles) {
        if(this.movement === false) {
            var d = floor(random(4));
            this.direction = d;
        }
        var lastx = this.gX;
        var lasty = this.gY;
        if(this.direction === 0) {
            this.gX += 32;
        }
        if(this.direction === 1) {
            this.gY += 32;
        }
        if(this.direction === 2) {
            this.gX -= 32;
        }
        if(this.direction === 3) {
            this.gY -= 32;
        }
        for(var i = 0; i < tiles.length; i++) {
            if(this.colission(tiles[i])) {
                this.gX = lastx;
                this.gY = lasty;
                this.movement = false;
                this.ghostMove(tiles);
            }
            else {
                this.movement = true;
            }
        }
        if(this.gX < 0)
            this.gX = width -32;
        if(this.gX >= width)
            this.gX = 0;

    }
```

Documentation generated via jsDoc showing the Global View of its members, classes and methods.

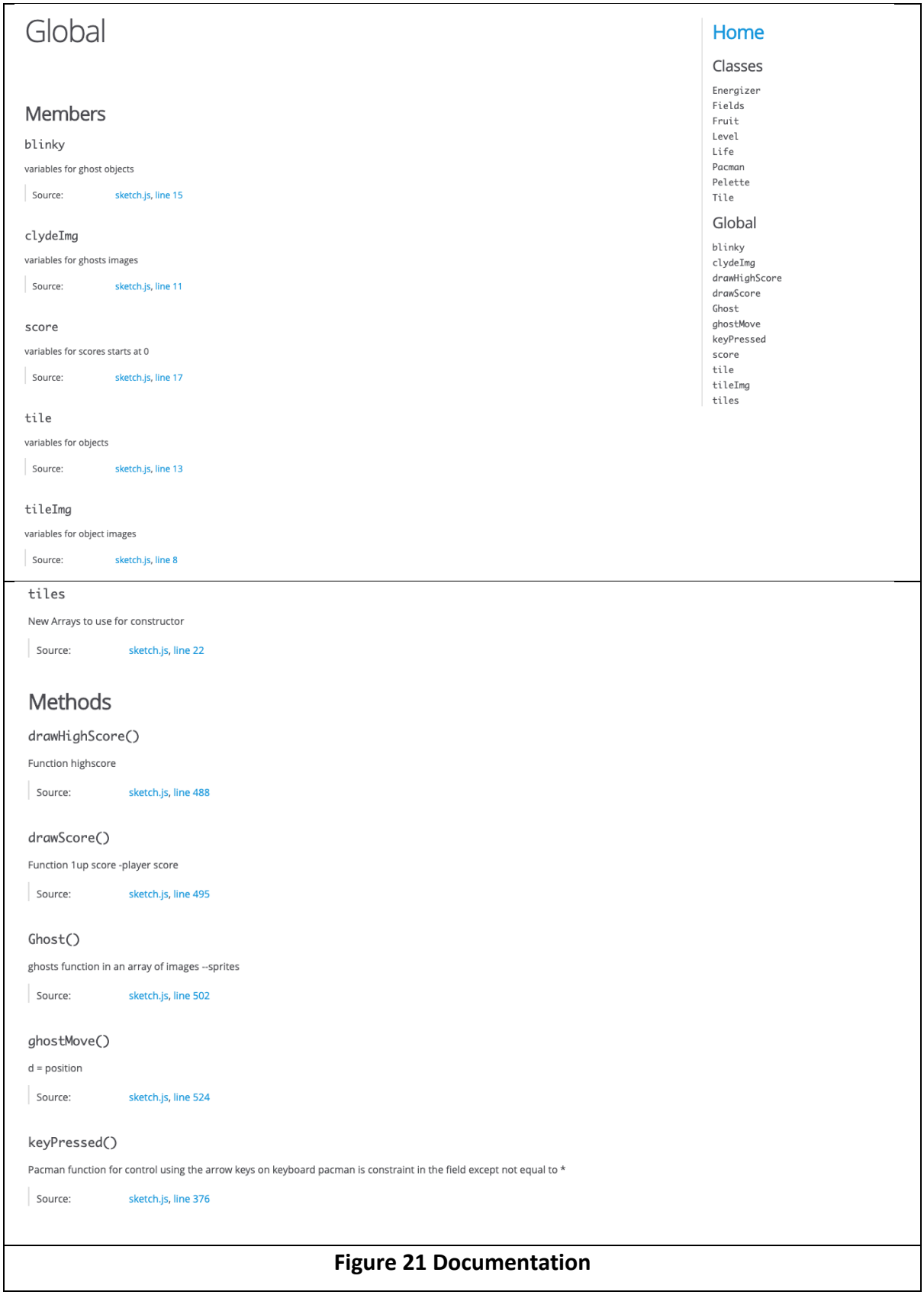


Figure 21 Documentation