CSC2045 Software Engineering

Report 2

Group 51

Student name	Student number	QUB email address
Aoife Brown	40152941	abrown72@qub.ac.uk
Chloe McAteer	40148890	cmcateer31@qub.ac.uk
Chloe Mullan	40152290	cmullan53@qub.ac.uk
Leanne McGuinness	40156078	lmcguinness09@qub.ac.uk
James Vint	40152348	jvint01@qub.ac.uk

Contents

Interface Design: 3
Class Relationship Diagram: 10
Sequence Diagrams: 11
Test Plan: 16
Adherence to Process 19
Identifying Secure System Features 19
Appendix 20

Interface Design:

Splash Screen:

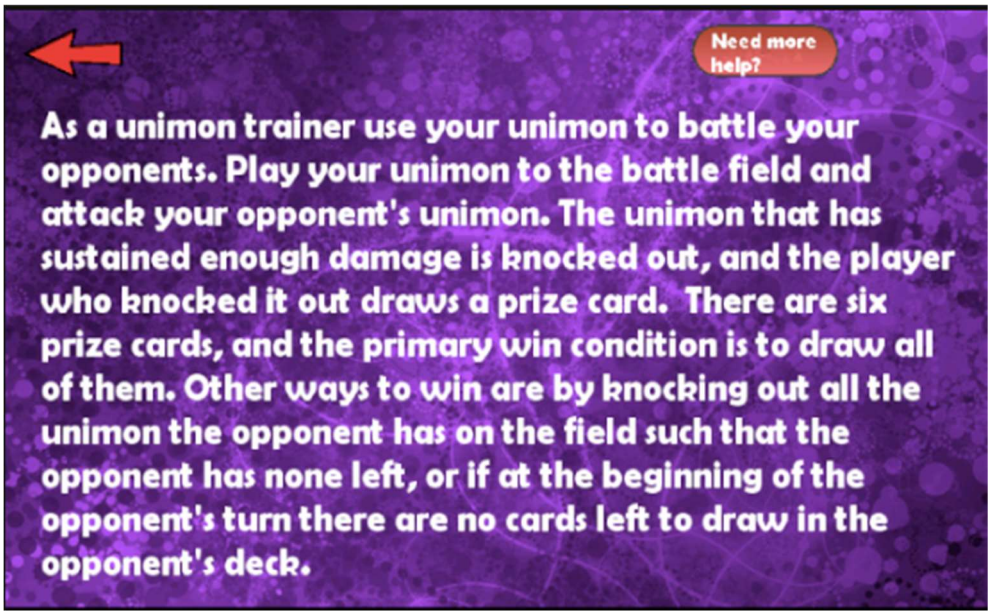
This is the splash screen that is displayed once the game is opened

Start Screen:

This is the games start screen that the user is taken to when the splash screen disappears

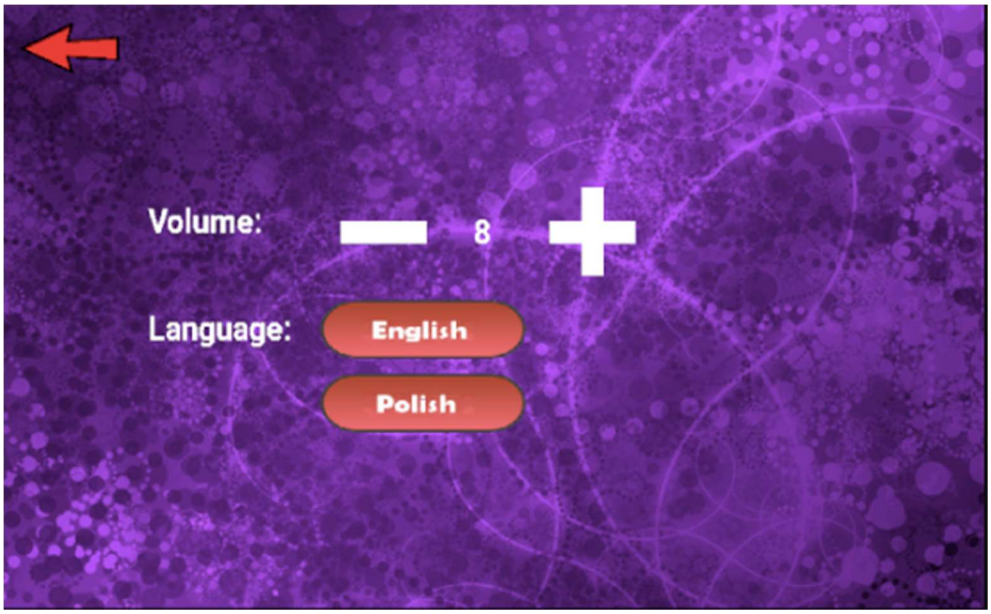
Instructions:

When the user presses the “How to play” button on the start screen or the “Instructions” button on the pause menu, they are shown these instructions. If they still require more assistance they can click the “need more help” button and can email their question.



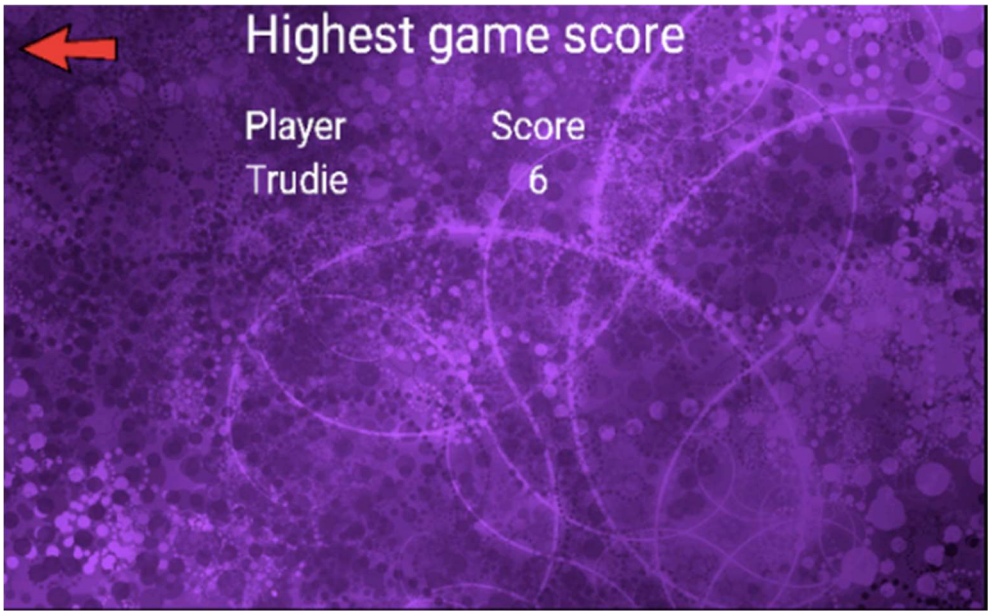
Settings:

The settings can be accessed by the “Settings” button on the start screen. Here the player can change the volume of the game and they can also choose which language they would prefer the game to be in.



High Score:

The highest game score can be accessed by pressing the “High score” button on the start screen. This screen displays the player who has received the highest score in the game, once another player scores higher than them it will then change



One Player/Two Player:

Once the player selects “Start” on the start screen it then asks them to select “One Player” or “Two Player”

Select game level:

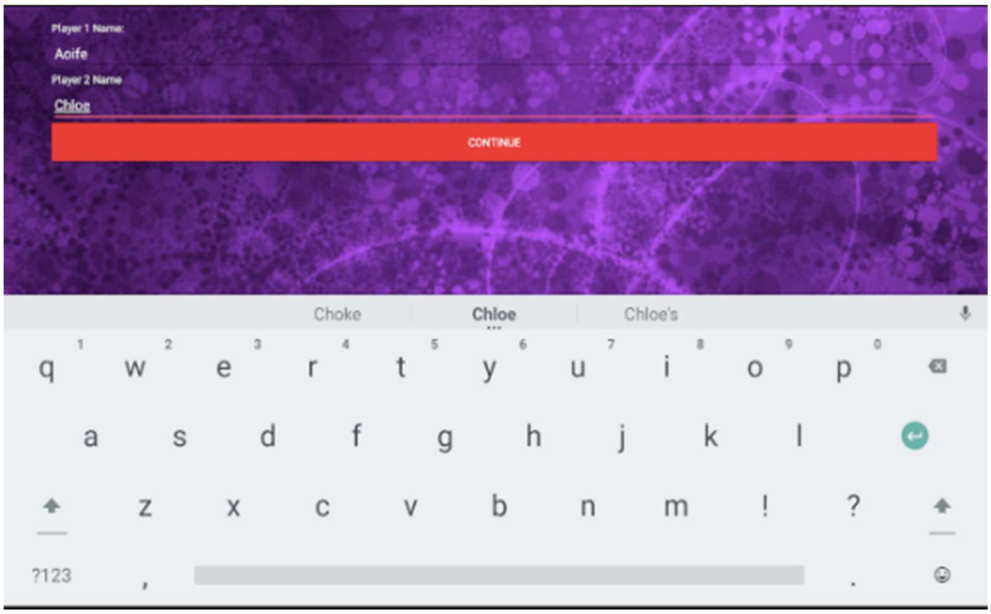
If the players chooses to play a one player game, they can select the level of difficulty of the game (easy/hard).

One player board:

This is the game board for a one player game

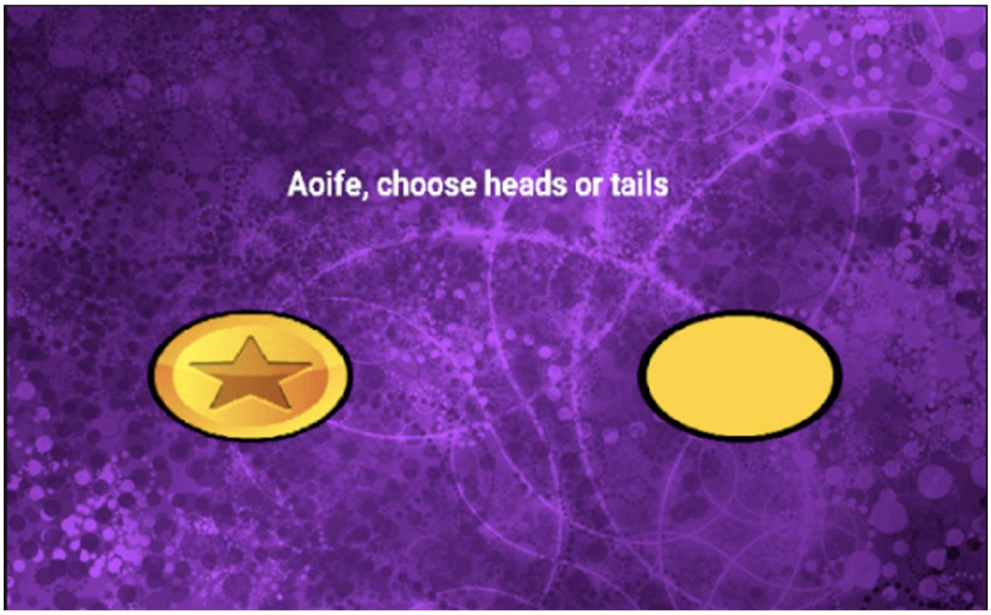
Enter player names:

If the player chooses to play a two player game, then they enter the names of both players



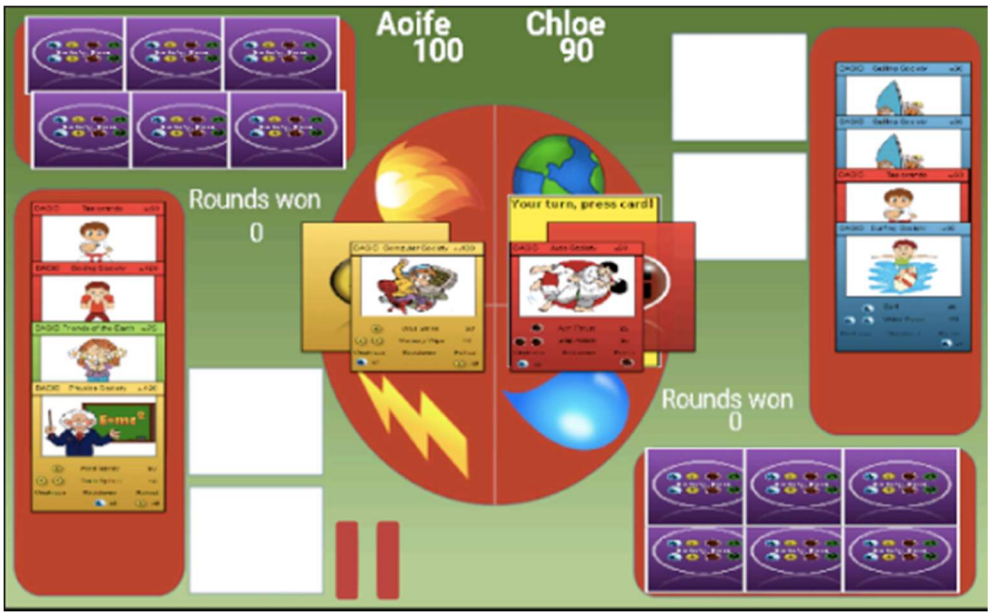
Heads or tails:

Once both player's names have been entered, a coin is flipped to see who plays first



Two player board:

This is the board of a two player game



Pause menu:

This menu is displayed when a player presses the pause symbol on the board, this menu allows them to resume playing the game, restart the game, view the game instructions, save the current game or quit the game.

Card Menu:

This menu is displayed when a player clicks on their active card. It displays what the player can do with their card.

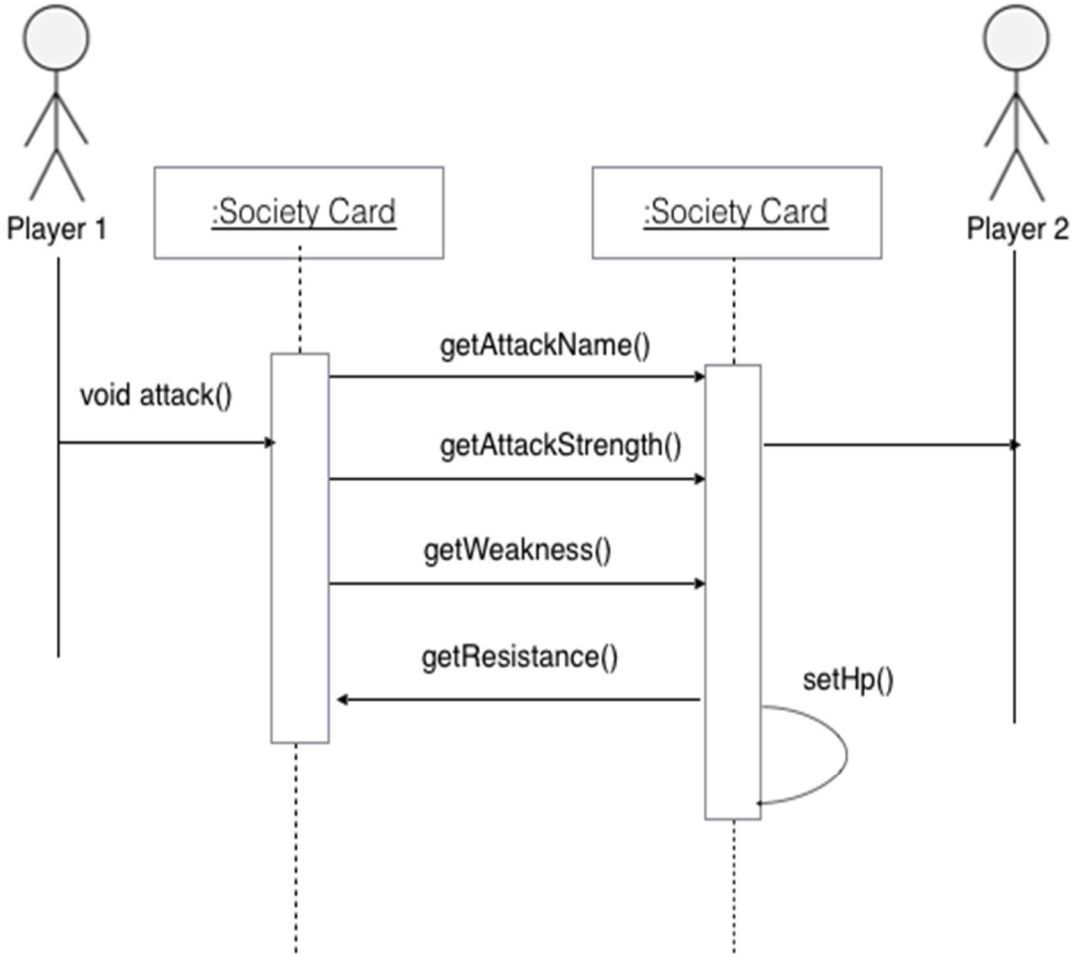
Choose a new card:

If the player retreats a card and wants to play another one, they can select a card from their bench and then press “use card!”

Class Relationship Diagram:

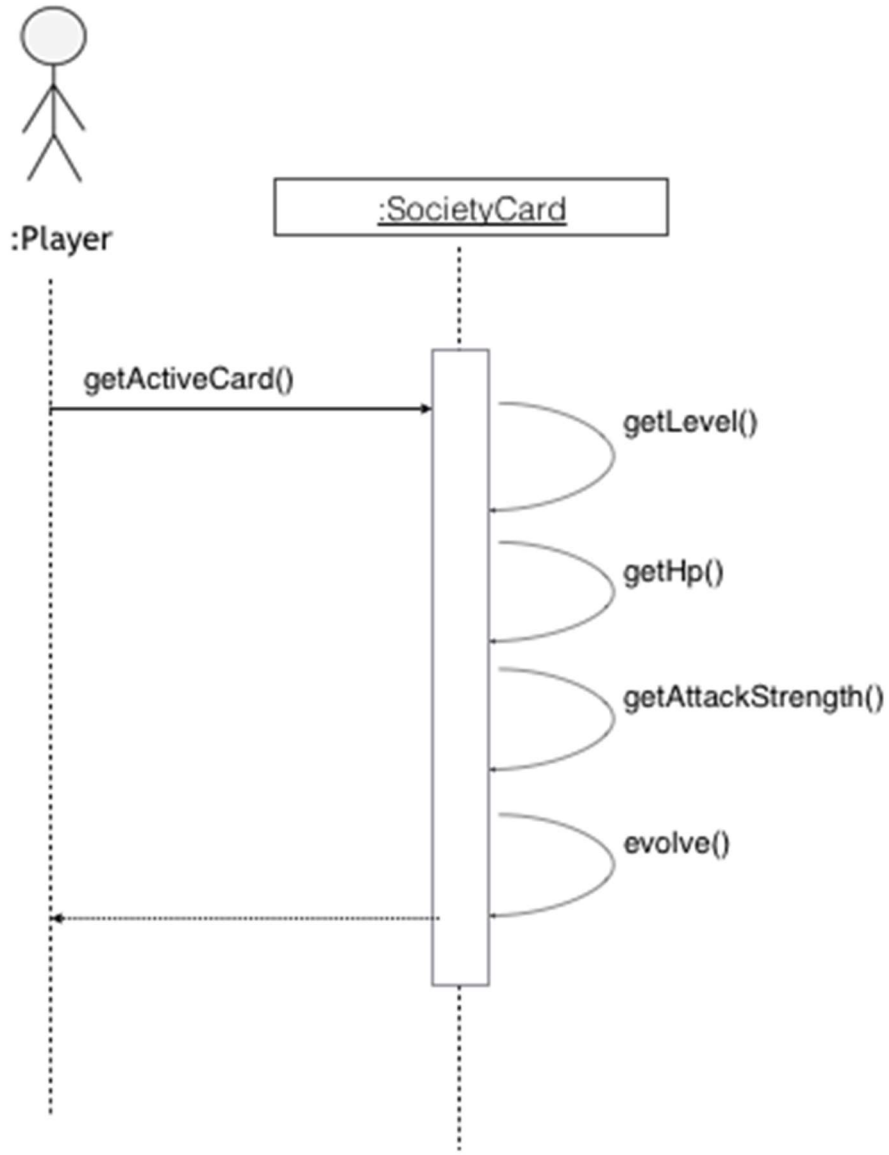
Sequence Diagrams:

Attack:



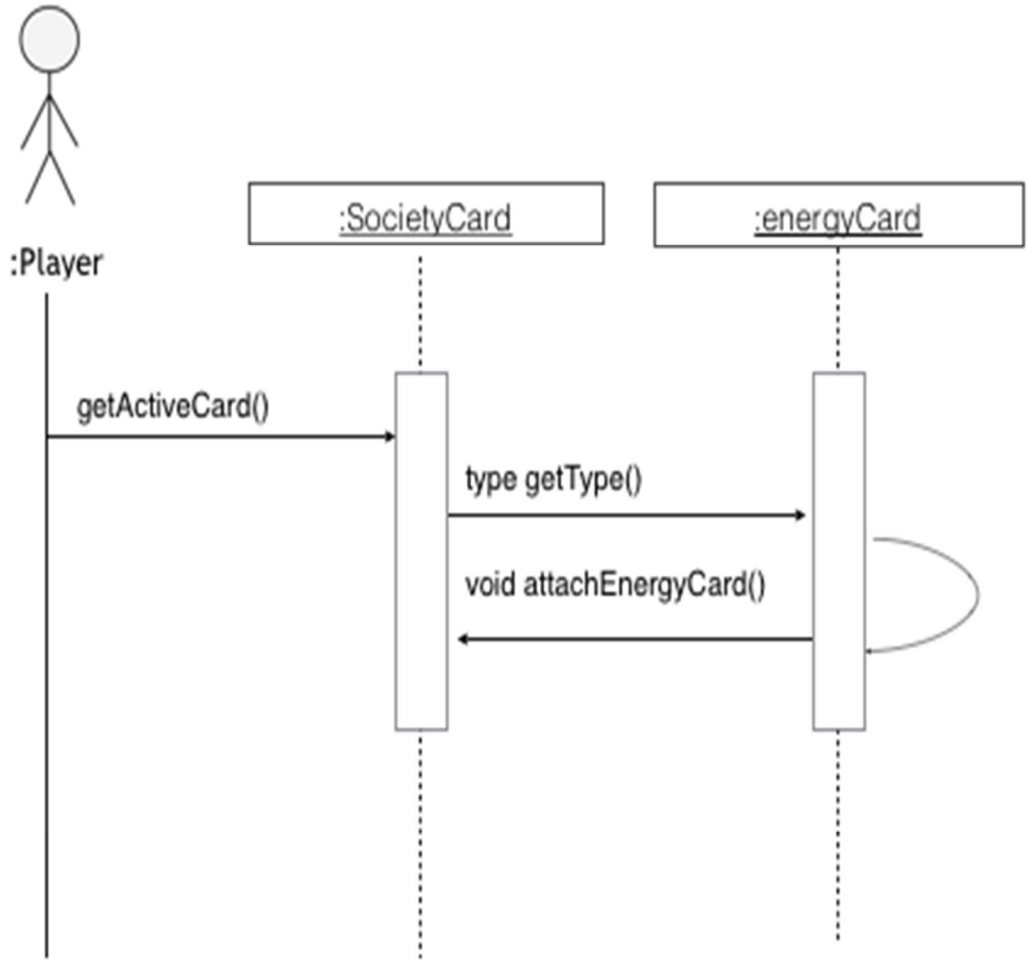
The above sequence diagram is for the attack method. The player uses this method to lower the HP on their opponent's active card, by the attack strength of their active card. If the opponent has a weakness to the type of card the player is using they will lose twice as much HP. However if they have a resistance to the type of card the player is using, they will only lose half the attack strength of the opposing player's active card.

Evolve:



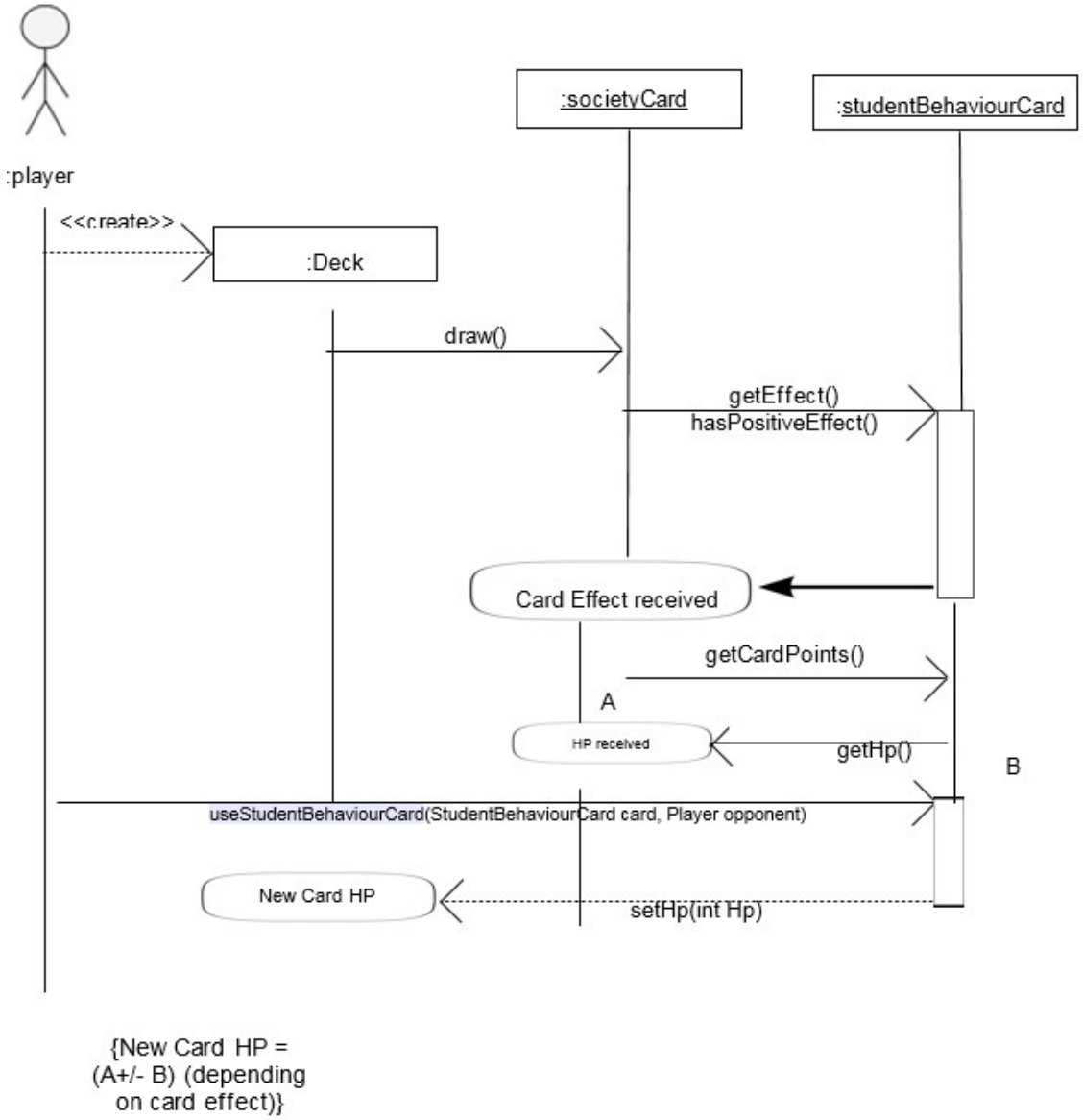
Above is the sequence diagram for the evolve method. The player would use this method on their active card. The evolve method requires the level and HP of the card it is to be used on. If the card is level one five points are taken from the card's HP, and ten points are added to the card's attack strength. If the card is level two ten points are taken from the card's HP, and twenty points are added to the card's attack strength. If the card is level three it cannot be evolved.

Attach Energy card:



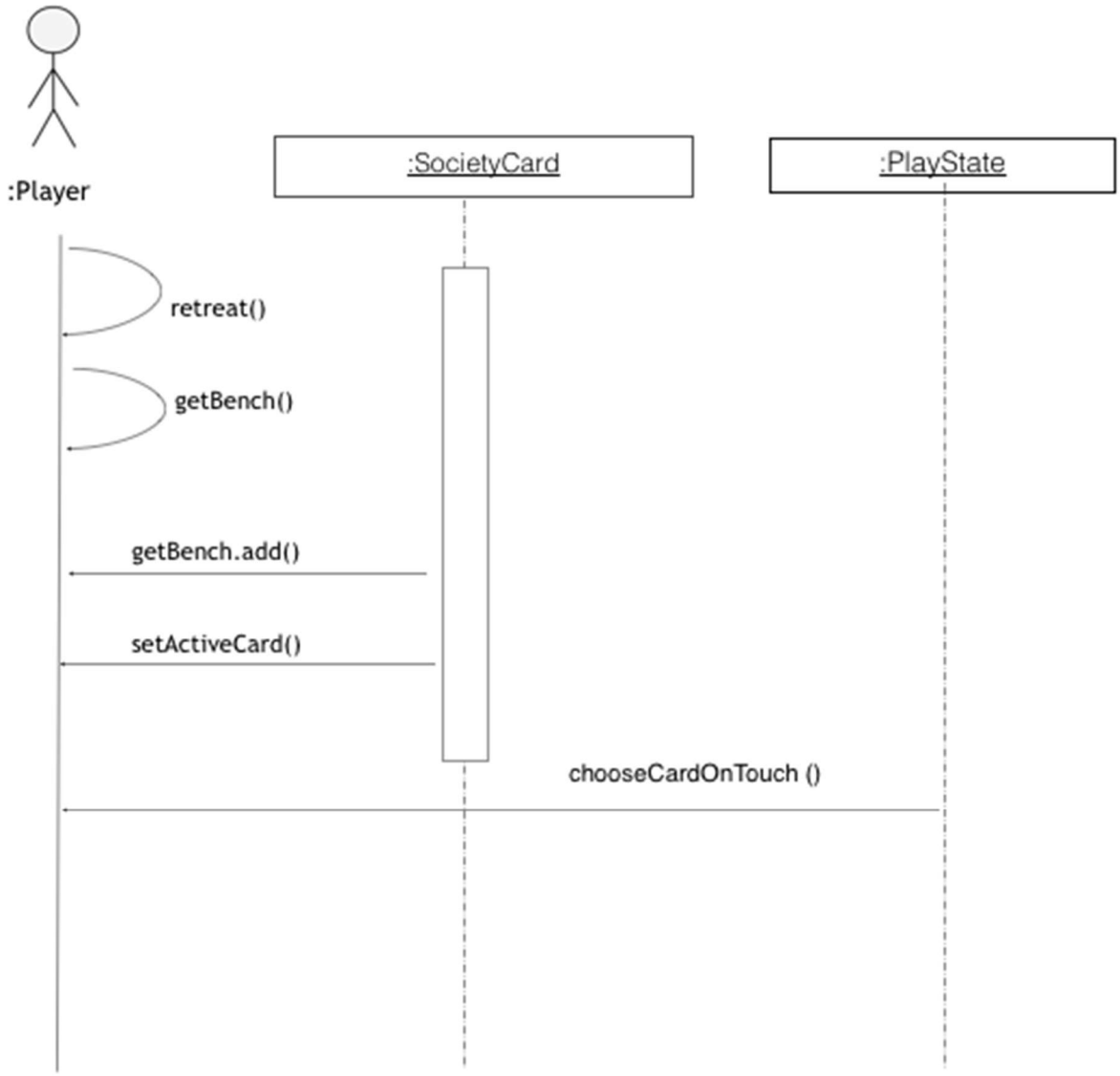
This is the sequence diagram for attaching an energy card. The player would use this method on their active society card. The attachEnergyCard() method requires the type of energy of the card it is to be used on. Whether the active card is electric, water, fighting or earth energy, the matching energy card will automatically be attached to it. In order for the Unimon to attack and use their special abilities at least one energy card in needed.

Use Student Behaviour card:



Once a user plays a Student Behaviour card, the Student Behaviour card effect will be received. If positive it will add the card points to the current users score, or if negative it will take away points from the current users opponents score. The current user or their opponents HP will then be up-dated depending on the hasPositiveEffect() method.

Retreat:



The above sequence diagram is for the retreat method. The player would use this method when they want to change their card. It moves the player's active card back to the bench and sets their active card to be null

Test Plan:

Below are some of the tests that where carried out in different class in the game, evidence of the tests working is in the appendix.

The following tests were carried out on the Player class:

1.This test ensures that after the player retreats a card their active card is set to null

```
public void retreatTest(){
mockPlayer1.retreat();
Assert.assertEquals(null, mockPlayer1.getActiveCard());
}
```

2. This test checks that once the player is not a winner it is set to false

```
public void checkIfWinnerTest(){
   Assert.assertEquals(false, mockPlayer1.checkIfWinner());
}
```

3. This tests that when a player attacks, their opponents active cards HP is reduced by their attackStrength.

```
public void attackTest(){
   mockPlayer1.setActiveCard(mockCard);
   mockPlayer2.setActiveCard(mockCard2);
   mockPlayer1.attack(mockPlayer2);
   Assert.assertEquals(50, mockPlayer2.getActiveCard().getHp());
}
```

4. This test checks that when the prize card is not flipped that it is set to false

```
public void checkPlayerFlippedPrizeCardTest(){
   boolean isFlipped = mockPlayer1.checkIfPlayerHasFlippedPrizeCards(mockPlayer1);
   Assert.assertEquals(false, isFlipped);
}
```

The following are tests that were carried out on the SocietyCard class:

1. This test ensures that the setHp method correctly sets the cards HP

```
public void setHpTest() {
   int mockHp = 75;
   mockCard.setHp(mockHp);
   Assert.assertEquals(mockHp, mockCard.getHp());
}
```

2. This test ensures that the setAttackStrength() method sets the attackStrength correctly

```
public void setAttackStrengthTest(){
   int mockAttackStrength = 30;
   mockCard.setAttackStrength(mockAttackStrength);
   Assert.assertEquals(mockAttackStrength, mockCard.getAttackStrength());
}
```

3. This test makes sure that once a player uses an evolve card that their active cards level moves up one level

```
public void evolveTest() {
   mockCard.evolve();
   Assert.assertEquals(Level.Level1, mockCard.getLevel());
}
```

The following are some of the tests that were carried out of the PlayState:

1.Test to ensure that the setPlayer1Wnner method, actually equals true when it is set to true:

```
public void setPlayer1Winner_SetPlayer1WinnerToTrue() {
   PlayState playState = new PlayState();
   playState.setPlayer1Winner(true);
   Assert.assertEquals(true, PlayState.player1Winner);
}
```

2.Test to ensure that the setPlayer2Winner method, actually equals true when it is set to true:

```
public void setPlayer2Winner_SetPlayer2WinnerToTrue() {
   PlayState playState = new PlayState();
   playState.setPlayer2Winner(true);
   Assert.assertEquals(true, PlayState.player2Winner);
}
```

3. Test to make sure that isPause is true, when it is set to true:

```
public void setPause_SetPauseToTrue() {
   PlayState playState = new PlayState();
   playState.setPause(true);
   Assert.assertEquals(true, playState.isPause());
}
```

4. Test to make sure that isMenu is true, when it is set to true:

```
public void setMenu_SetMenuToTrue() {
   PlayState playState = new PlayState();
   playState.setMenu(true);
   Assert.assertEquals(true, playState.isMenu());
}
```

5. Test to make sure that isPrizeCardError is true, when it is set to true:

```
public void setPrizeCardError_SetPrizeCardErrorToTrue() {
   PlayState playState = new PlayState();
   playState.setPrizeCardError(true);
   Assert.assertEquals(true, playState.isPrizeCardError());
}
```

6. Test to make sure that isAttackPlayer1 is true, when it is set to true:

```
public void setAttackPlayer1_SetAttackPlayer1ToTrue() {
   PlayState playState = new PlayState();
   playState.setAttackPlayer1(true);
   Assert.assertEquals(true, playState.isAttackPlayer1());
}
```

7. Test to make sure that isAttackPlayer2 is true, when it is set to true:

```
public void setAttackPlayer2_SetAttackPlayer2ToTrue() {
   PlayState playState = new PlayState();
   playState.setAttackPlayer2(true);
   Assert.assertEquals(true, playState.isAttackPlayer2());
}
```

8. Test to make sure that isChooseCard is true, when it is set to true:

```
public void setChooseCard_setChooseCardToTrue()  {
   PlayState playstate = new PlayState();
   playstate.setChooseCard(true);
   Assert.assertEquals(true, playstate.isChooseCard());
}
```

9. Test to make sure that isEvolvePlayer1 is true, when it is set to true:

```
public void setEvolvePlayer1_setEvolvePlayer1ToTrue()  {
```

```
        PlayState playstate = new PlayState();
        playstate.setEvolvePlayer1(true);
        Assert.assertEquals(true, playstate.isEvolvePlayer1());
    }
```

10. Test to make sure that isEvolvePlayer2 is true, when it is set to true:

```
    public void setEvolvePlayer2_setEvolvePlayer2ToTrue()  {
        PlayState playstate = new PlayState();
        playstate.setEvolvePlayer2(true);
        Assert.assertEquals(true, playstate.isEvolvePlayer2());
    }
```

11. Test to make sure that isCardRetreated is true, when it is set to true:

```
    public void setCardRetreated_setCardRetreatedToTrue() {
        PlayState playstate = new PlayState();
        playstate.setCardRetreated(true);
        Assert.assertEquals(true, playstate.isCardRetreated());
    }
```

12. Test to make sure that isRetreatError is true, when it is set to true:

```
    public void setRetreatError_setRetreatErrorToTrue() {
        PlayState playstate = new PlayState();
        playstate.setRetreatError(true);
        Assert.assertEquals(true, playstate.isRetreatError());
    }
```

13. Test to make sure that isRenderAnimation is true, when it is set to true:

```
    public void setRenderAnimation_setRenderAnimationToTrue()  {
        PlayState playstate = new PlayState();
        playstate.setRenderAnimation(true);
        Assert.assertEquals(true, playstate.isRenderAnimation());
    }
```

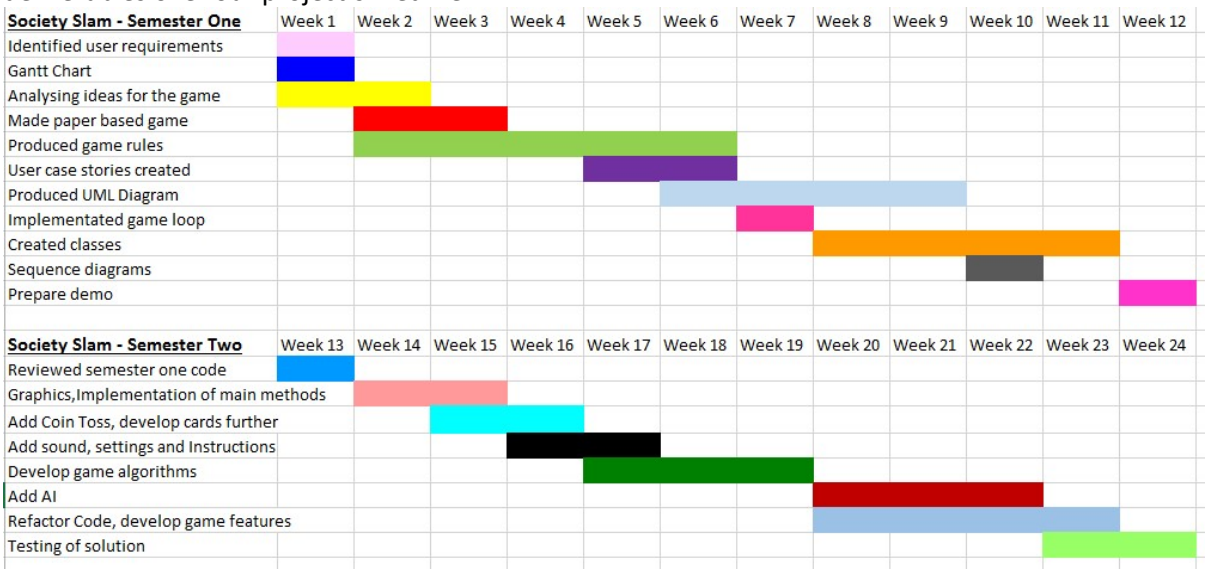
The following test was carried out in the coin class:

1. This tests that when the coin is flipped it is set to either heads or tails:

```
public void coinFlipTest(){
    mockCoin= mockCoin.flip();
    if(mockCoin == Coin.HEADS||mockCoin == Coin.TAILS) {
        Assert.assertTrue(true);
    }
}
```

Adherence to Process

When developing our working implementation of a trading card game we followed an Iterative and Incremental Development model of software development and used Agile Methods throughout. This meant it was easier for us to react to change between sprints and there was also a lower risk involved. Our lecturer, Phillip Hanna (CSC 2044), acted as the 'client'. In each sprint we sought to satisfy the client through early and frequent delivery of the system and welcomed any changes and improvements to the solution even late in the project. Throughout this project we promoted a sustainable development pace and paid continuous attention to technical excellence and good design. The team followed the principles of Agile in that it was self organising, it emphasised face to face communiction by reflectecting regularly on any improvements in weekly meetings and made use of techniques such as weekly stand-ups. The Gantt chart below represents the iterative development process which we followed, it indicates some of the main development strands and deliverables over our project's lifetime.



We have attached brief printed material representing evidence of adherence to process in the appendix of this report. Here you will find; documented evidence of appropriate unit testing with code coverage; version managed code that shows participation across the project team in the form of Git repository graphs, weekly team minutes. As well as this we have submitted our working solution on a USB drive.

Identifying Secure System Features

In the current system we have determined that the security objectives required are:

- Integrity – a user can only modify if authorised to do so.
- Authorisation –granting access to legitimate users and validating what a user is allowed to do (permissions).

In the future there will be opportunities to implement secure system features into our solution. This may come if an online multiplayer feature is added to the system. In this case we have determined that the software objectives required, in addition to above, are as follows:

- Privacy – ability to store and control a user's personal information.
- Anonymity- making users unidentifiable to others.

Appendix