CA314 Scrabble - Assignment 2

Gareth Hogan 20379616 *Detutu Adebisi 20330263*
Rosa Duessmann 20495376 *Peadar McHenry 20490894*
Eimear McKevitt 20500199 *Paul McKee 20414956*

Contents:

Work Distribution:

Paul - Object Diagram
Peadar - Communication Diagrams
Eimear - Revising Class Diagram, Network Ideas & Deployment Diagram
Rosa - Prototyping, Class Skeletons
Detutu - UI Mock-Ups
Gareth - Sequence Diagrams, State Diagrams, Report

Introduction:

Now in the second submission of the project, we are transitioning from design to prototyping and starting to implement our Scrabble game. In this assignment we revised and finalised important diagrams like the class diagram which are vital to the overall structure of our implementation, we also did some new diagrams showing how individual objects and classes would work together to achieve a functioning game, this helped us to pin down and iterate on our plans for the implementation.

In this submission, you will find diagrams and designs which show how we plan our code to work together and interact between classes and methods. And towards the end, you will see the beginning of our implementation based upon our design.

Meetings & Agile Method - Update:

In the first submission, we mentioned how we were trialling using Jira to document and track our agile method, here is an update on how that is going. We are still keeping up with it, normally hoping to update it at least once a week. We do still find that keeping it updated very regularly is a challenge because of how busy we are with other assignments and work as individuals, this also is reflected in how we haven't had meetings as frequently as we did earlier in the semester. But even though we know we haven't been able to utilise the Jira tool like an agile team in the industry would, how we are using it has helped some of us between our meetings as a place to check up on what must be done, if someone can start something we haven't assigned work to yet, or help review diagrams completed by someone else *(you can find pictures of the board evolving in the appendix).* Alongside that we have tried to keep in regular contact via messages with the team, to review people's work, and give feedback and help if needed in between meetings.

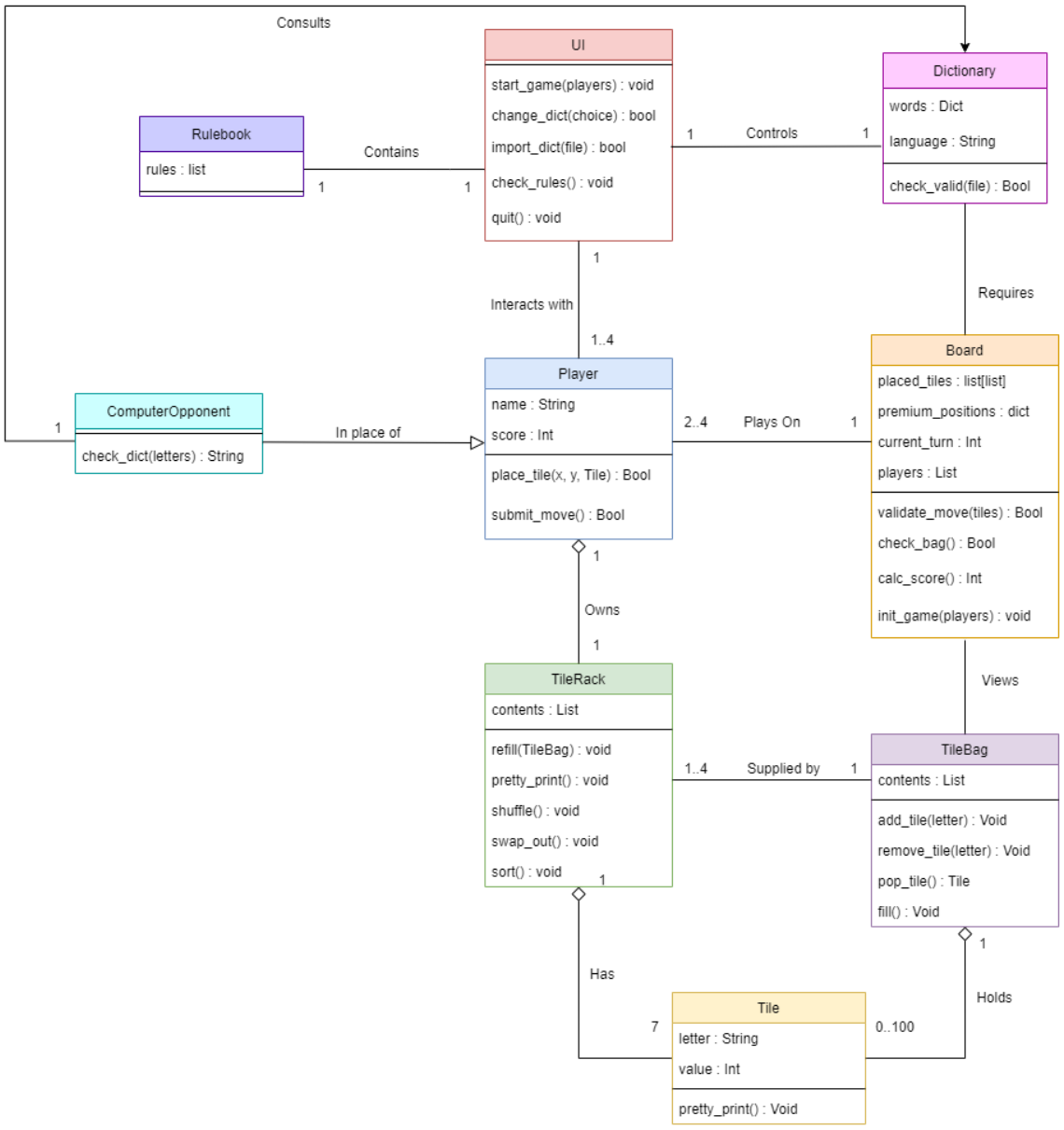
continue to the next page

Revised Class Diagram:

The class diagram is one of the most integral and useful diagrams to use when designing an object-oriented program. It gives an overview of the system as a whole, while also giving good detail about important implementation details such as methods and attributes.

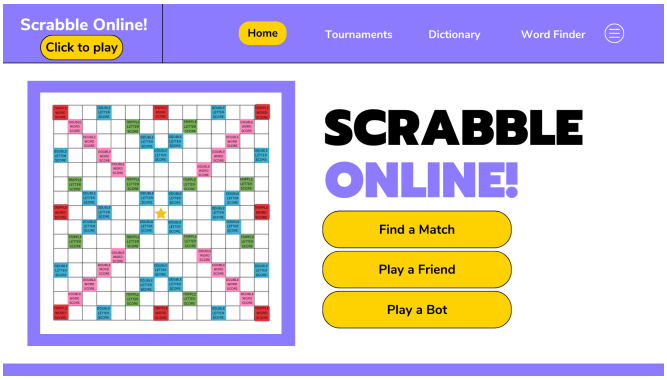
After the first submission, we updated the class diagram to better reflect the improved design as we move towards implementation. We have better named the associations between all our classes, adding any that were missing along with cardinality for all those associations. We have also gone through all the classes' renaming methods and attributes to better show how they will appear in the final code, also adding return types and data types where we knew them. Some things are still not finalised and will probably change as we find the solutions during implementation, especially methods around the Player-UI-Board relationship as that is a complex area of the final implementation. We have also left out a few methods arguments, as we were not sure yet exactly how the implementation would work, and would not know what it needed yet.

Below you can see the most updated class diagram we have so far in our project:



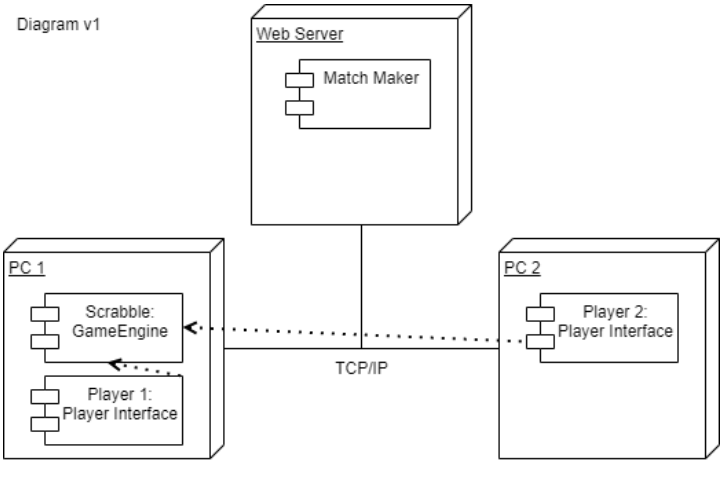
UI Mockup:

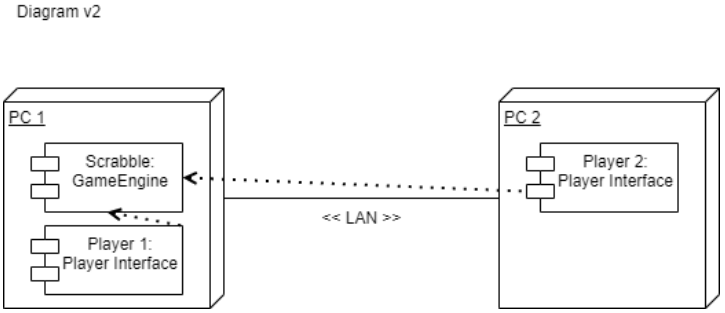
Ultimately we do not know how our game will look at this early stage, we are still working on prototypes and mostly the backend side of the game that a user will never see. But a simple UI Mockup is good to give us a target to aim for and remind us of the importance of getting what the user sees working, otherwise, the work behind the scenes won't work if the user can't play the game. For our design we looked at the classic scrabble board and other online scrabble sites like playscrabble.com, also looking at what methods we are planning to implement and then designing how the end user might eventually activate those methods.



Network Ideas - Deployment Diagram:

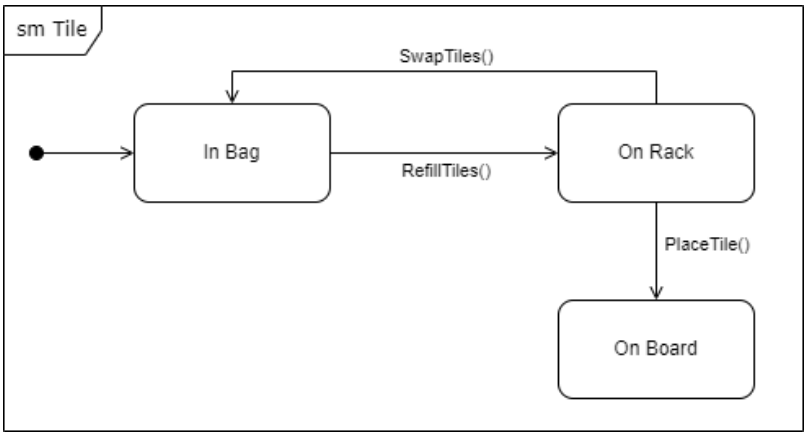
We had two main Network ideas for our project. Our first idea (Diagram v1) was to have an online matchmaker, the game would be hosted through one of the clients but players would be able to connect to them using our online service. The other idea (Diagram v2) we have is not to use a matchmaker and to have the game hosted and joined locally via a LAN connection. The latter idea is a lot more achievable and suited to our needs, as we would not need an external server to host the matchmaker, although it isn't totally in line with our original vision. In Python, we can implement a TCP/IP connection and LAN connection with relative ease.

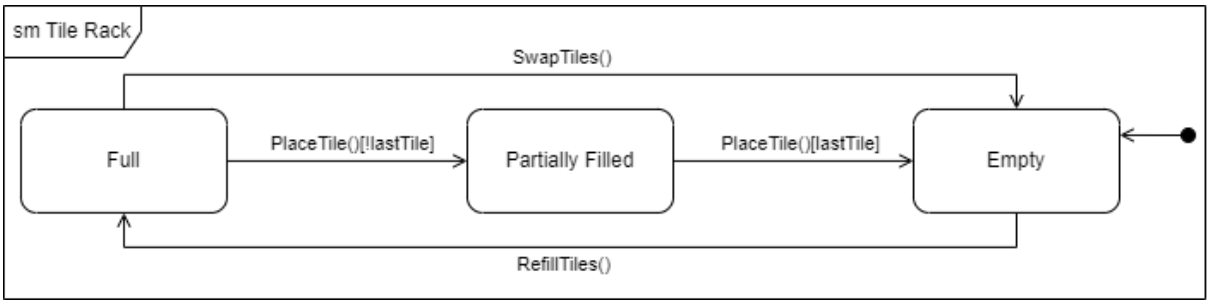




State Diagrams:

State diagrams are used to model the behaviour of one object, modelled independently of others. It shows what “states” the object can be in, and also helps visualise what are the triggers and events that cause that object to change state. We found it very helpful to model the states of our objects, mostly finding it useful to think about and design the trigger events, and what exactly is powering each change of state. You can see our state diagrams for the *Tile* and the *Tile Rack* below.

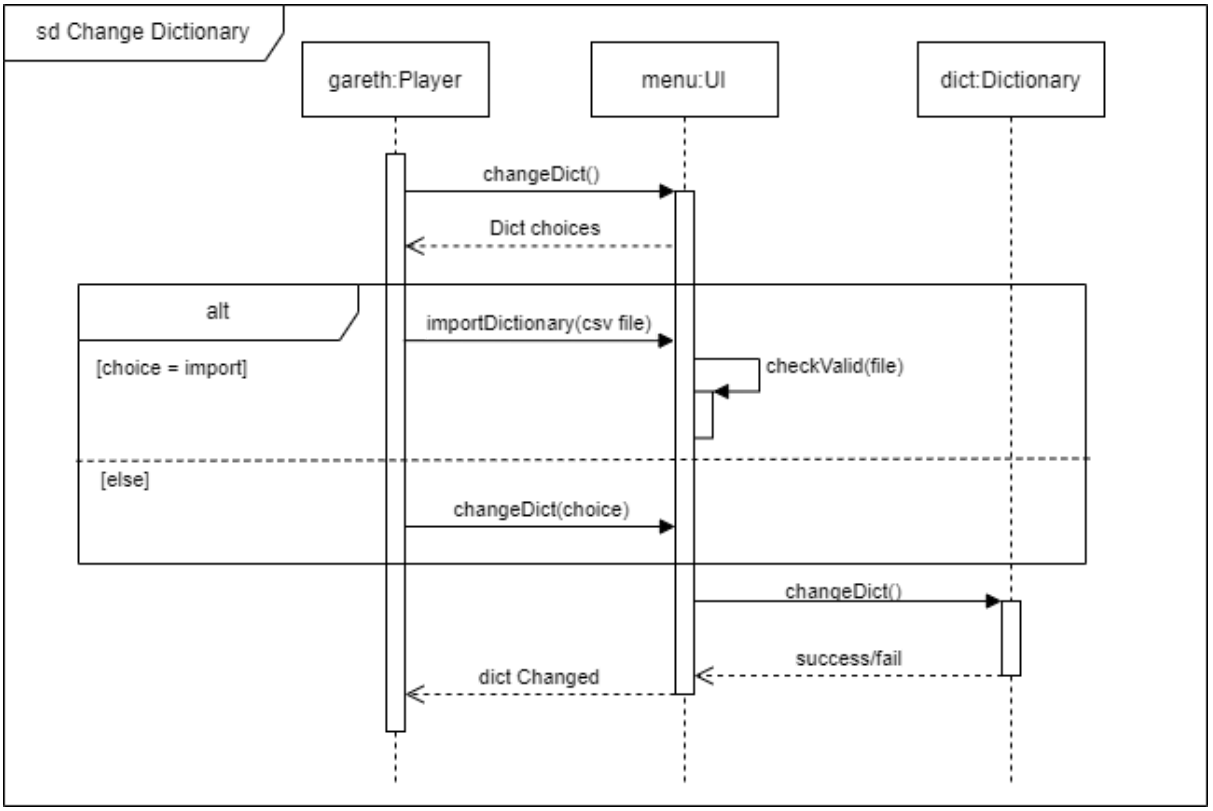


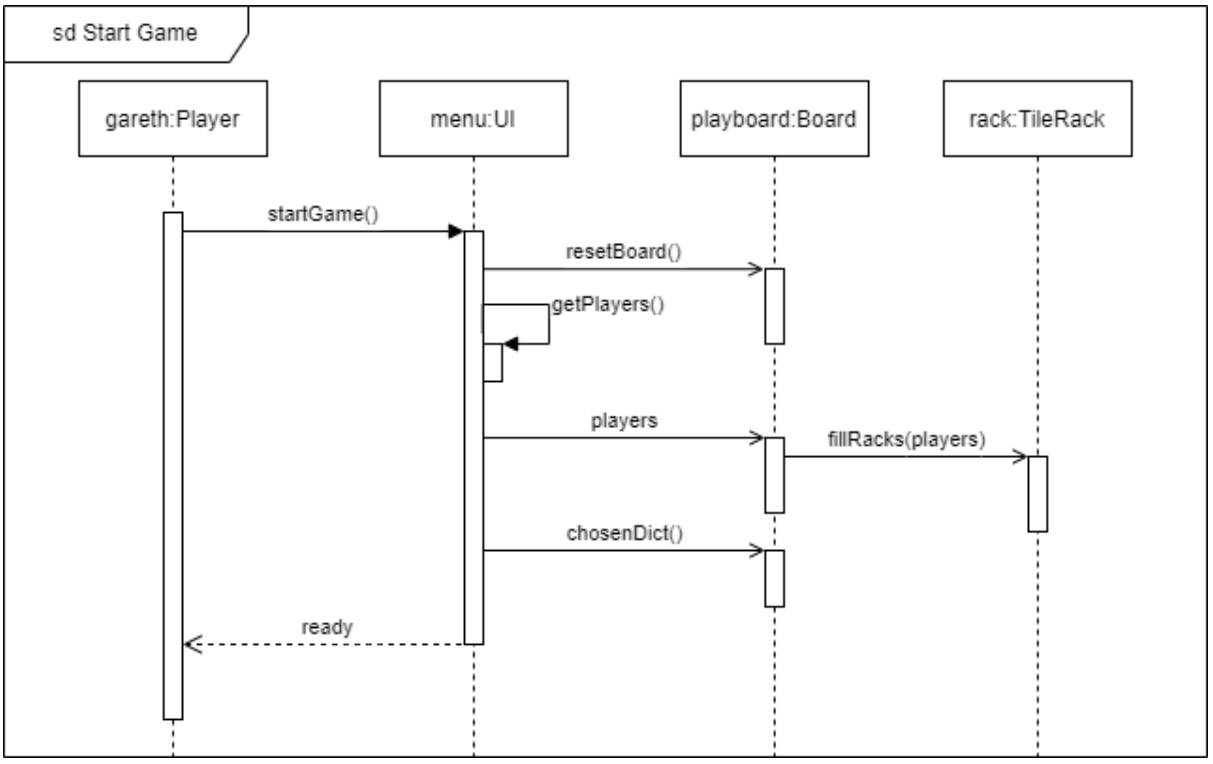


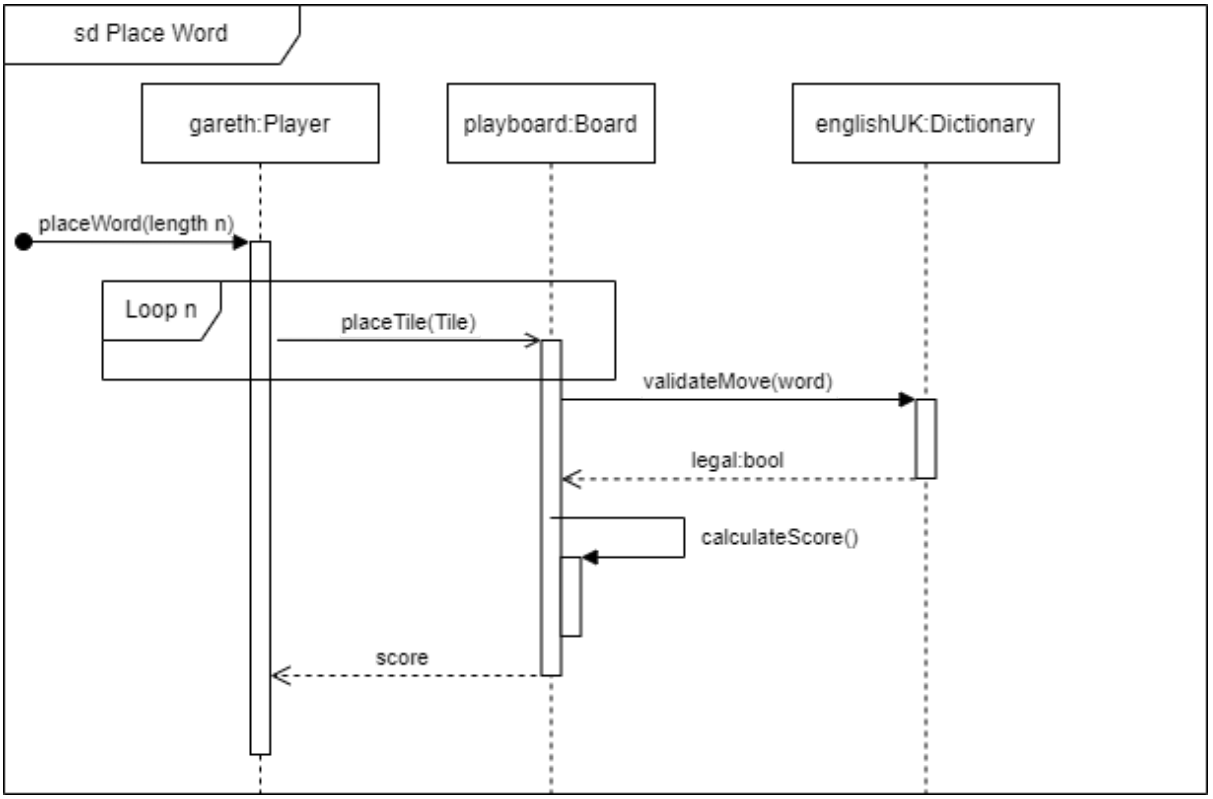
Sequence Diagrams:

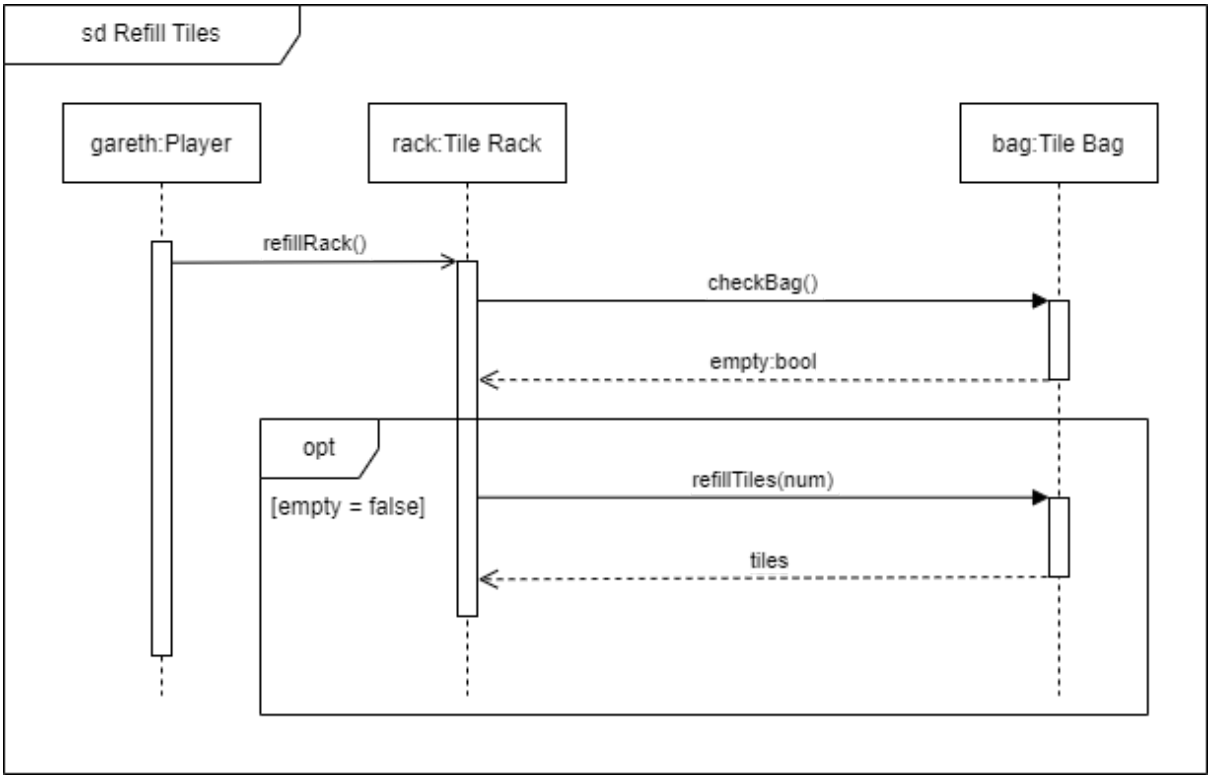
As opposed to state diagrams, sequence diagrams show how multiple objects interact with each other over time to allow a task to be completed. They can help us visualise how a user can accomplish a certain task which requires multiple objects to interact and pass data between them. They help translate some of the more complex use cases from our previous submission into useful plans for us to use in designing the methods and attributes that each object will hold, and to know exactly what each object passes and returns between each other to complete the task. Below you can see the sequence diagrams for *Changing Dictionary, Starting Game, Place Word* and *Refill Tiles.*

continue to the next page


sd Change Dictionary
gareth:Player
menu:UI
dict:Dictionary
changeDict()
Dict choices
alt
importDictionary(csv file)
checkValid(file)
[choice = import]
[else]
changeDict(choice)
changeDict()
success/fail
dict Changed

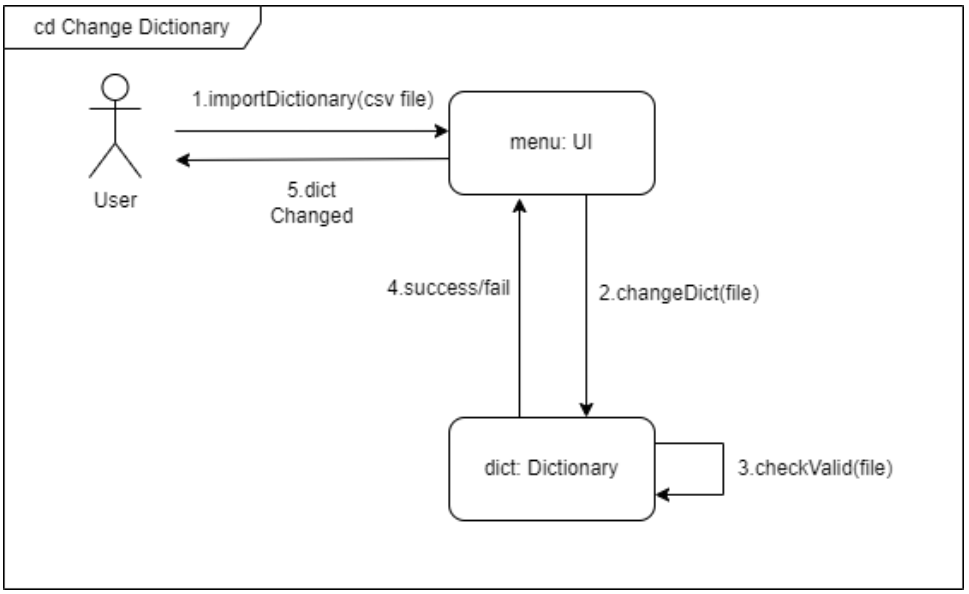

sd Start Game
gareth:Player
menu:UI
playboard:Board
rack:TileRack
startGame()
resetBoard()
getPlayers()
players
fillRacks(players)
chosenDict()
ready

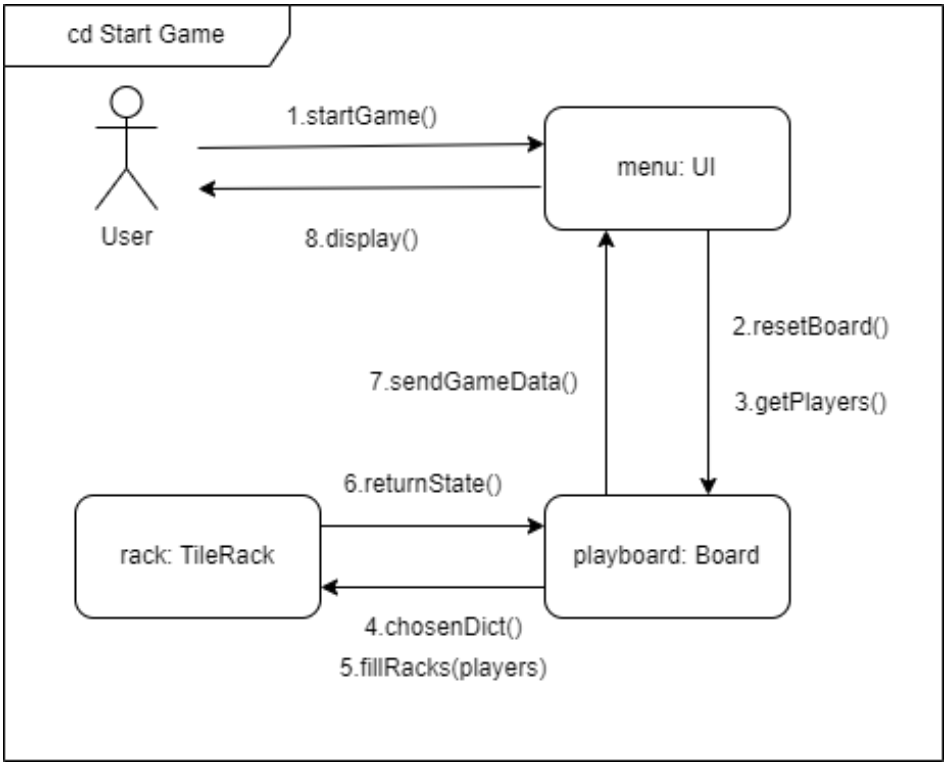

sd Place Word
gareth:Player
playboard:Board
englishUK:Dictionary
placeWord(length n)
Loop n
placeTile(Tile)
validateMove(word)
legal:bool
calculateScore()
score

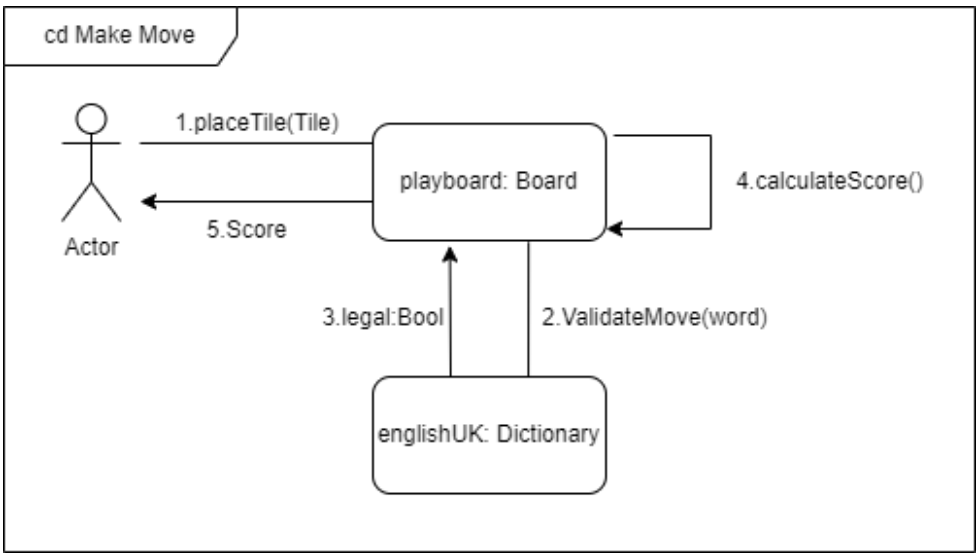

sd Refill Tiles
gareth:Player
rack:Tile Rack
bag:Tile Bag
refillRack()
checkBag()
empty:bool
opt
[empty = false]
refillTiles(num)
tiles

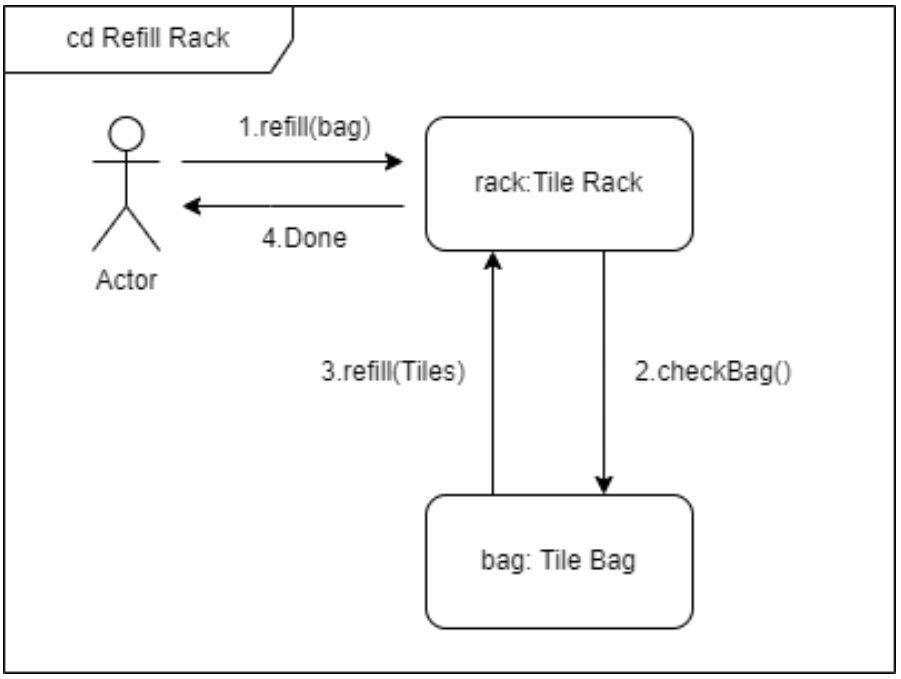
Communication Diagrams:

Communication diagrams are closely related to the sequence diagrams we made above, they model the same type of communication between objects but are more focused on the relationship between objects and not the sequence of operations that occurs. You can see in the diagrams the messages passed between each object as the actor tries to accomplish a specific task. The diagram is laid out to better understand the relationship and association between classes as opposed to the sequence and timing of events that the SD shows. Below you can see four communication diagrams, each modelled from a sequence diagram above.



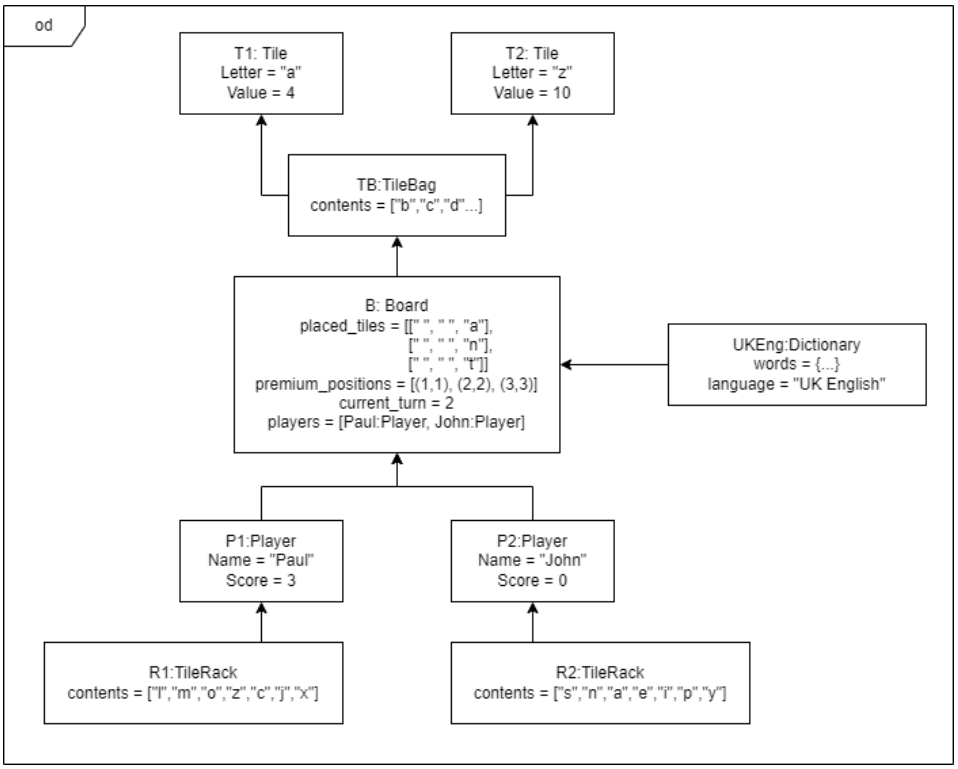



cd Make Move
1.placeTile(Tile)
playboard: Board
4.calculateScore()
5.Score
Actor
3.legal:Bool
2.ValidateMove(word)
englishUK: Dictionary


cd Refill Rack
1.refill(bag)
rack:Tile Rack
4.Done
Actor
3.refill(Tiles)
2.checkBag()
bag: Tile Bag

Object Diagram:

An object diagram can be considered a special subset of the class diagram, it helps to show what the class diagram would look like when the classes are instantiated as objects, with example attributes. It further helps to visualise how our design will work when implemented, as it serves as an example game almost, we can see players' names, their tiles, the board and more. Below you can see the object diagram we made, it covers a subset of our class diagram based on the board and players.



Revisions & Iterations:

Throughout the second submission, we have been constantly updating the diagrams as we flesh out the design and also as we prototype and test it in implementations. We presented the final diagrams above but we also have some examples of iterations we went through in the past before we got the final version. In the appendix you can find:

- One of the first iterations of the object diagram
- Old communication diagrams
- Pre-prototyping class diagram

A lot of these iterations were early designs that we got done, and then when we came together as a team to compare and review, we shared feedback on how to improve them and to change parts to keep a consistent design across all diagrams. For example, the class diagram you can see in the appendix is made before we compared it without prototyping in python. After prototyping, we could see methods we had missed in design before, and also after starting implementation in python we could decide on format choices such as Class names being in Pascal Case `TileRack` and method names being in snake case `pretty_print`, which we then replicated across the final class diagram.

Class Skeletons & Code Prototyping:

- Find the full code we have so far at our Git Repo here

Throughout the second submission, we began prototyping the Tile-related classes to get a better sense of what the finished project would require to function on the backend. We also created a short testing script to debug and test the interactions between the classes, as laid out in the Class Diagram. As a result, we have most of the methods for the Tile-related classes in a functional state.

This is our Tile class, with some comparison overrides to enable sorting. You can view TileRack and TileBag in our GitLab or in in the Appendix

```
class Tile():

    def __init__(self, letter):
        self.letter = letter
        self.value = tile_info[letter][0]

    def __str__(self):
        return f'|{self.letter.upper()} {self.value}|'

    def __eq__(self, other):
        return self.letter == other
    def __gt__(self, other):
        return self.letter > other
    def __lt__(self, other):
        return self.letter < other

    def pretty_print(self):
        print(f'+-+\n|{self.letter.upper()}|\n+-{self.value}')
```

Our `TileBag.fill()`, is a helper function to fill the Tile Bag with tiles in the default distribution.

```
    def fill(self):

        for letter in tile_info.keys():
            for j in range(tile_info[letter][1]): # value
tuple index 1: the amount of tiles of this letter in default
distribution
                self.add_tile(letter)

        print("Filled Tile Bag")
```

Our `TileBag.push_tile(tile)` and `TileBag.pop_tile()`, are used to transfer tiles to and from the Tile Rack.

```
    def push_tile(self, tile):
        #placeholder - probably required for
TileRack.swap_out()
        return

    def pop_tile(self):
        return self.contents.pop(random.randint(0,
len(self.contents) - 1))
```

Showcase of TileRack functions

```
Tiles Refilled
 +-+  +-+  +-+  +-+  +-+  +-+  +-+
 |C|  |N|  |T|  |G|  |L|  |A|  |L|
 +-3  +-1  +-1  +-2  +-1  +-1  +-1
Tiles Shuffled
 +-+  +-+  +-+  +-+  +-+  +-+  +-+
 |L|  |N|  |T|  |G|  |A|  |L|  |C|
 +-1  +-1  +-1  +-2  +-1  +-1  +-3
Tiles Sorted
 +-+  +-+  +-+  +-+  +-+  +-+  +-+
 |A|  |C|  |G|  |L|  |L|  |N|  |T|
 +-1  +-3  +-2  +-1  +-1  +-1  +-1
```

Debug print of TileBag

```
TileBag object, Tile count: 93
Tiles:
|  0|, |  0|, |A 1|, |A 1|, |A 1|, |A 1|, |A 1|, |A 1|, |A 1|, |B 3|, |B 3|, |C 3|, |D 2|, |D 2|, |D 2|, |D 2|, |E 1|, |E 1|, |E 1|, |E 1|
, |E 1|, |E 1|, |E 1|, |E 1|, |E 1|, |E 1|, |E 1|, |E 1|, |F 4|, |F 4|, |G 2|, |G 2|, |H 4|, |H 4|, |I 1|, |I 1|, |I 1|, |I 1|, |I 1|, |I 1|, |I
1|, |I 1|, |I 1|, |J 8|, |K 5|, |L 1|, |L 1|, |M 3|, |M 3|, |N 1|, |N 1|, |N 1|, |N 1|, |N 1|, |O 1|, |O 1|, |O 1|, |O 1|, |O 1|, |O 1|, |O 1|, |
O 1|, |P 3|, |P 3|, |Q 10|, |R 1|, |R 1|, |R 1|, |R 1|, |R 1|, |R 1|, |S 1|, |S 1|, |S 1|, |S 1|, |T 1|, |T 1|, |T 1|, |T 1|, |T 1|, |U 1|, |U 1|
, |U 1|, |U 1|, |V 4|, |V 4|, |W 4|, |W 4|, |X 8|, |Y 4|, |Y 4|, |Z 10|
```

Team Minutes:

Wednesday 12/10, 15:25-15:40
Attendance: Paul, Peadar, Rosa, Eimear, Detutu, Gareth

- Met soon after the first submission to look at the second submission, analyse what needed to be done
- Delegated some tasks that could be done without feedback and clarifications from Renaat

Wednesday 19/10, 12.45-13.15
Attendance: Paul, Peadar, Rosa, Eimear, Detutu, Gareth

- Divided the rest of the work between members, to be done by next week then we can make the report
- Discussed what needed to be done for each piece of work that was a bit obscure
- Shared web pages and resources for people doing diagrams

Wednesday 02/11, 12.50-13.15
Attendance: Paul, Peadar, Detutu, Gareth, Eimear, Rosa

- Checked up on work to be finished
- Divided small bits of work left to be done
- Organised to have a meeting before submission Friday

Friday 04/11, 14.00-15.00
Attendance: Rosa, Eimear, Gareth

- A quick online call with some members to run through final checks of the report before submission, formatting, inserting images, etc.

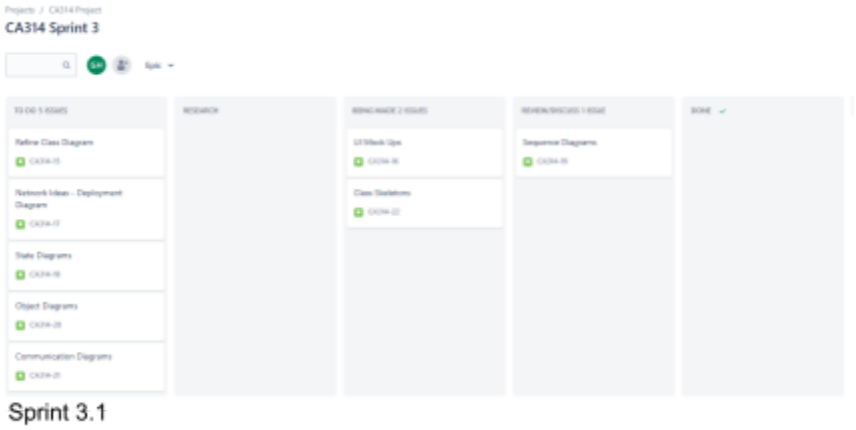
Conclusion:

In this submission, we refined and finalised a lot of the pre-implementation design. We created diagrams built on top of what we designed in the first submission, now showing how our classes and objects would work together to achieve a working Scrabble game. We did a lot of refinement on key parts of our design and our core class diagram which shows the overview of the whole system, bringing in more detail as we started to look toward the implementation.

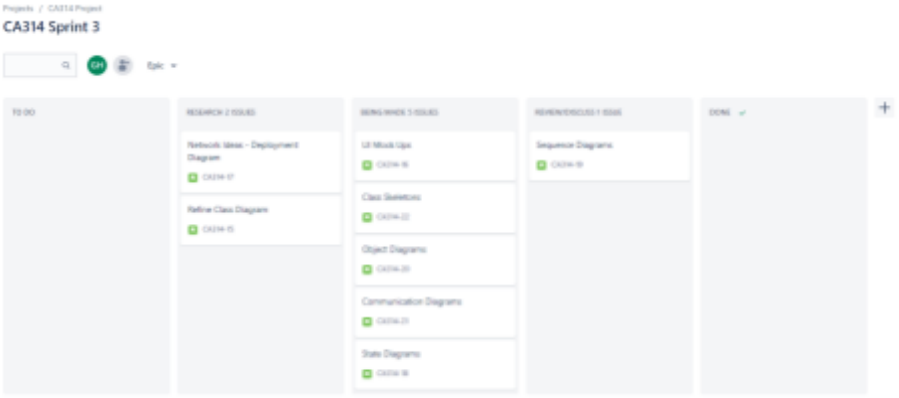
We have begun to transition now into the development and implementation phase of the project beginning with the prototyping of some of our classes and some proof of concept testing of how the classes and methods will be interacting in the background. As we finish this submission, we will be switching to full development and implementation of the game. We will begin to look more at our ideas for the networking and UI which we began thinking about in this submission.

Appendix:

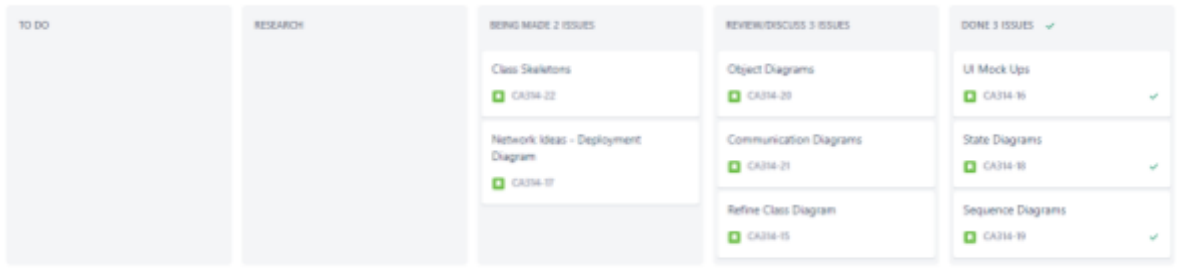
Agile Sprint Images



Sprint 3.1

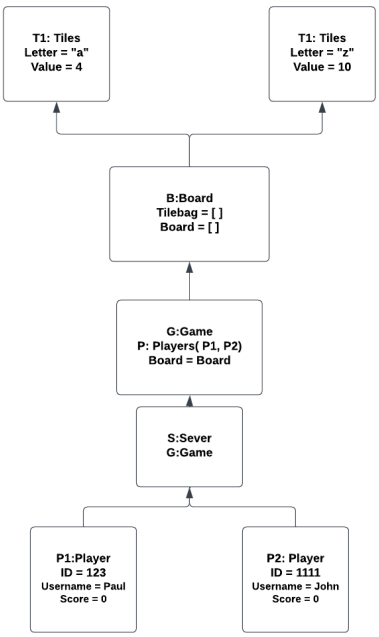


Sprint 3.2

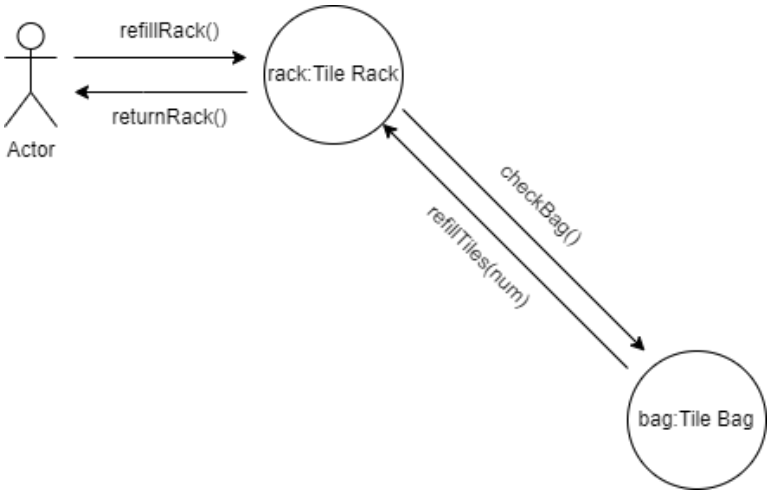


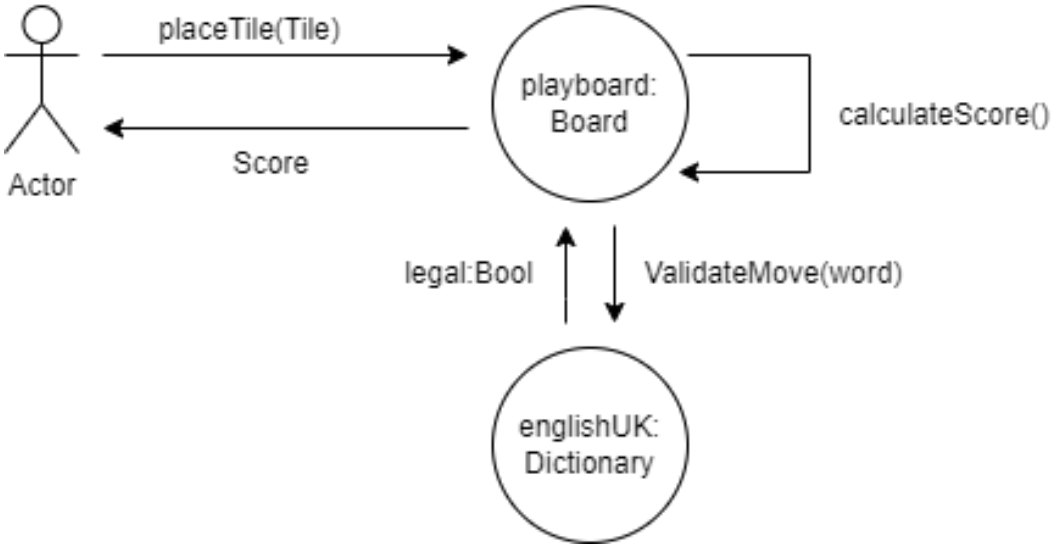
Sprint 3.3

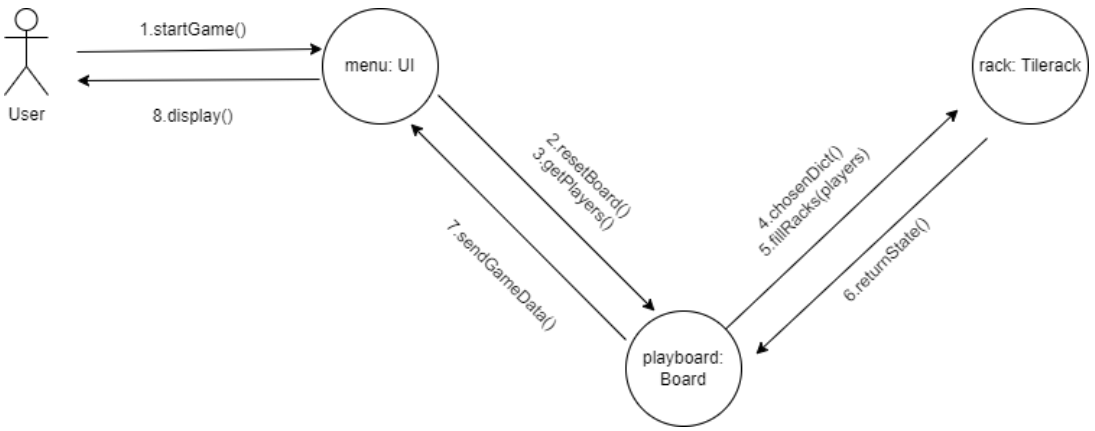
Iterations: Object Diagram



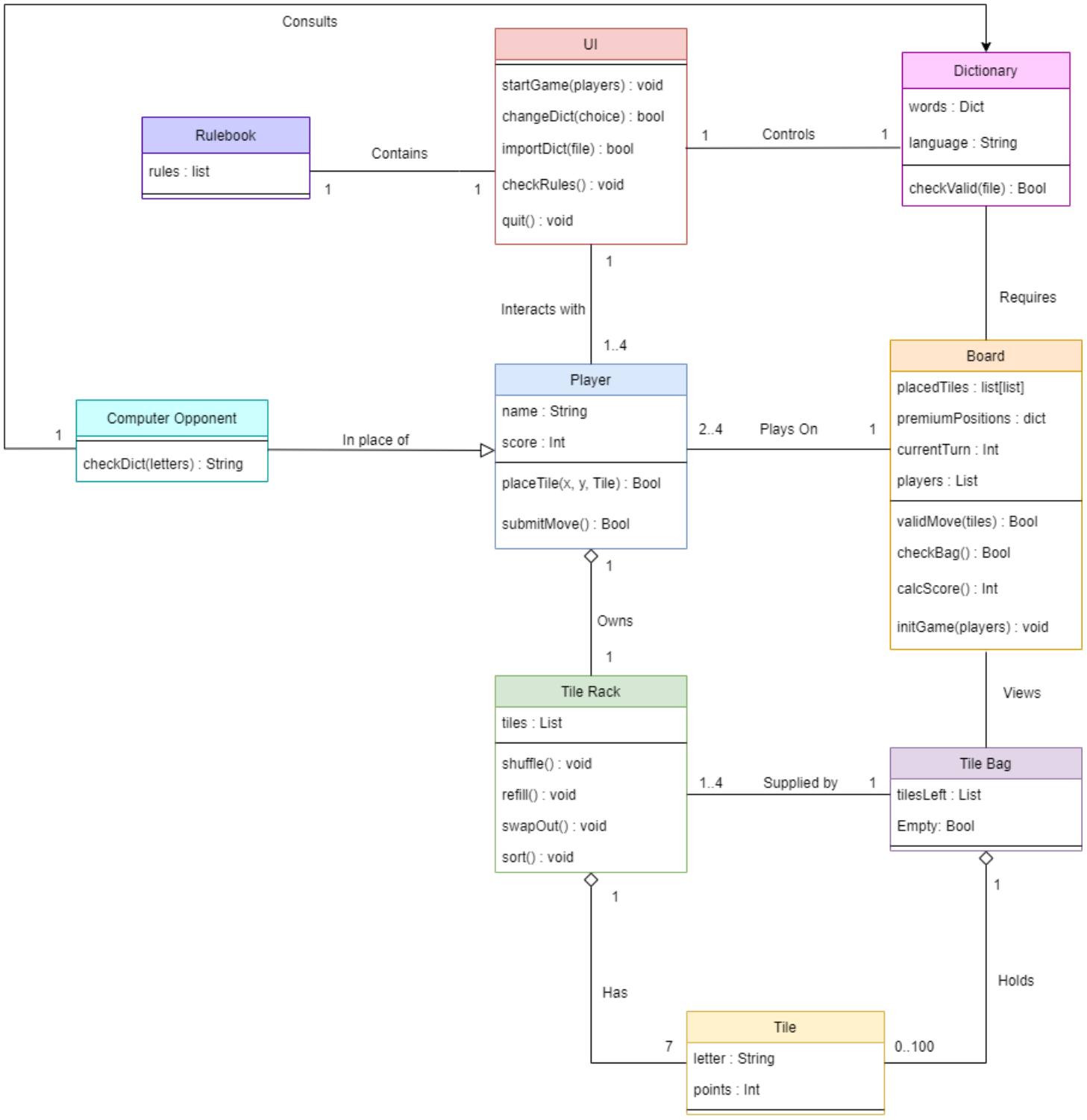
Communication Diagrams







Class Diagram



Class Skeletons for TileBag and TileRack

```
class TileBag():

    def __init__(self):

        self.contents = []

        self.fill()

    def __str__(self):

        return f'TileBag object, Tile count: {len(self.contents)}\nTiles:\n{", ".join(map(str, self.contents))}'

    def add_tile(self, letter):

        self.contents.append(Tile(letter))

    def remove_tile(self, letter):

        self.contents.remove(letter)

    def push_tile(self, tile):
        #placeholder - probably required for TileRack.swap_out()
        return

    def pop_tile(self):

        return self.contents.pop(random.randint(0, len(self.contents) - 1))

    def fill(self):

        for letter in tile_info.keys():
            for j in range(tile_info[letter][1]): # value tuple index 1: amount of tiles of this letter in default distribution
                self.add_tile(letter)

        print("Filled Tile Bag")
```

```
class TileRack():

    def __init__(self):

        self.contents = []

    def __str__(self):

        return f'TileBag object\nTiles: {", ".join(map(str, self.contents))}'

    def refill(self, bag):

        while len(self.contents) < 7:

            self.contents.append(bag.pop_tile())

        print("Tiles Refilled")

    def shuffle(self):
        random.shuffle(self.contents)

        print("Tiles Shuffled")

    def swap_out(self, bag):
        # placeholder
        return

    def sort(self):
        self.contents.sort()

        print("Tiles Sorted")

    def pretty_print(self):

        if len(self.contents) <= 0:
            print("Empty!")
            return

        outstr = (" +-+ " * len(self.contents)) + "\n"
        for t in self.contents:
            outstr += f' |{t.letter.upper()}| '
        outstr += "\n"
        for t in self.contents:
            outstr += f' +-{t.value} '

        print(outstr)
```