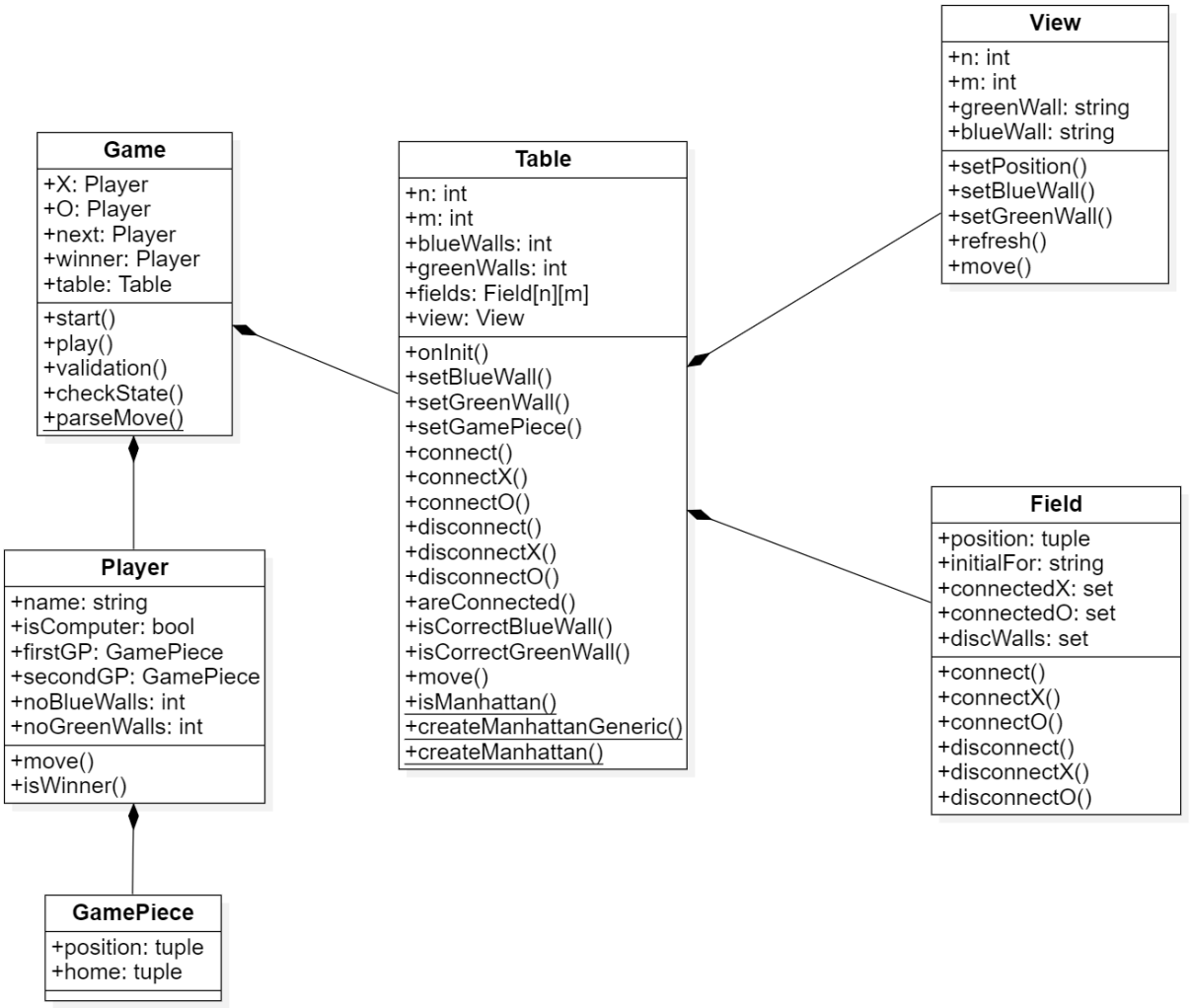
Извештај пројекта „Blockade”

Теодора Коцић 17190
Стефан Алексић 16995

На слици 1 приказан је класни дијаграм у којем су наведене све до сада формиране класе у сврху решавања проблема.



Слика 1 Класни дијаграм

1. Фаза – Формулација проблема и интерфејс

За потребе дефинисања стања проблема игре користи се класа *Game* која садржи инстанцу објекта класе *Table*, као и две инстанце класе *Player*.

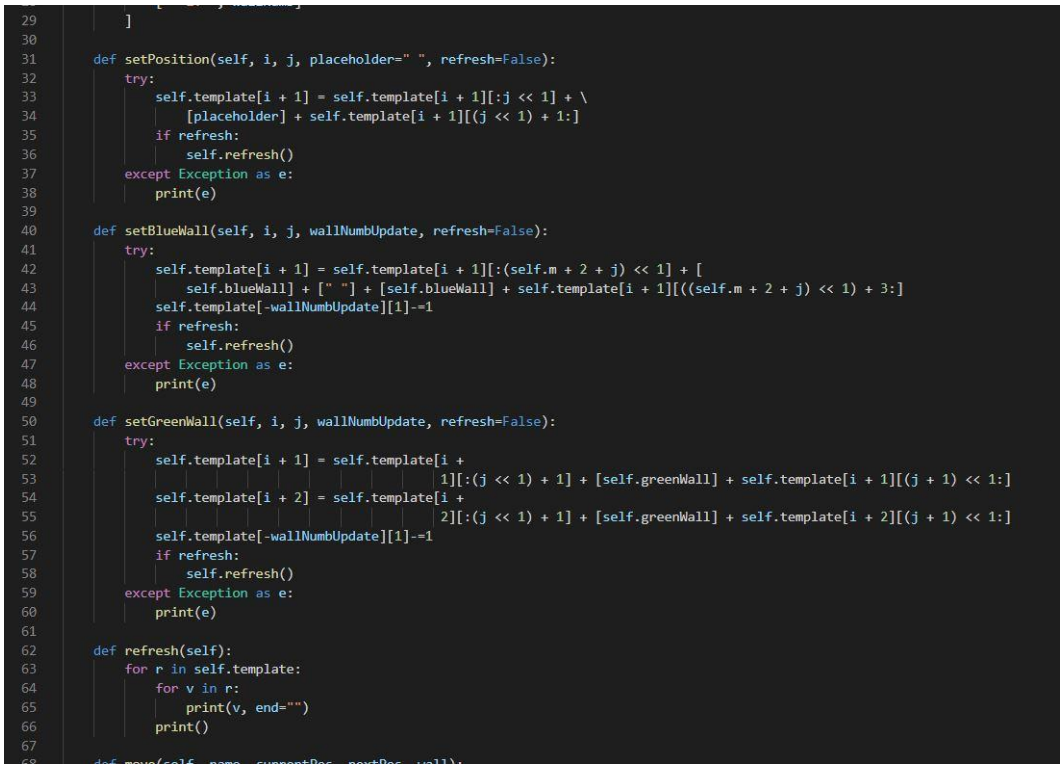
```
5  class Game:
6      def __init__(self, n=11, m=14, initial={(4, 4): 'X', (8, 4): 'X', (4, 11): 'O', (8, 11): 'O'}, wallNumb=9, greenWall="\u01c1", blueWall="\u2550", rowSep="\u23AF"):
7          self.table = Table(n, m, initial, wallNumb, greenWall, blueWall, rowSep)
8          self.human = None
9          self.computer = None
10         self.next = None
11         self.winner = None
```

Слика 2 Конструктор класе Game

У оквиру класе *Table*, као што се може видети у 7. линији кода на слици 2, врши се иницијализација табле која се користи у игри. Параметрима *n* и *m* дефинишемо димензије табле, атрибут типа речник *initial* садржи вредност почетних позиција сваког од четири играча на табли. У променљивој *wallNumb* памти се информација о броју зидова једне боје коју поседује сваки од играча (дакле број зидова играча X, односно играча O је 2**wallNumb*). Остали параметри конструктора служе за приказ табле у конзоли апликације.

Класу *View* користимо само у сврхе представљања изгледа читаве игре у конзоли. Поред конструктора у оквиру којег се врши исцртавање табле. Класа садржи и функције којима се заправо креира шаблон који се даље користи у игри за приказивање тренутног стања табле. Приликом покретања апликације поред табле у конзоли приказане су и све четири фигуре на својим почетним позицијама и то је одрађено у функцији *onInit(...)* класе *Table* и овај део кода приказан је на слици 6.

```
4  class View:
5      def __init__(self, n=11, m=14, wallNumb=9, greenWall="\u01c1", blueWall="\u2550", rowSep="\u23AF"):
6          self.n = n
7          self.m = m
8          self.greenWall = greenWall
9          self.blueWall = blueWall
10
11         self.template = [
12             [" ",  *[(" {0:x}".format(i).upper())
13                      for i in range(1, m + 1)], "  "],
14             [" ", *(" " + self.blueWall) * m, "  "],
15             *[list("{0:x}".format(j).upper() + self.greenWall + (" |") * (m - 1) + " " + self.greenWall +
16                    "{0:x}".format(j).upper() + "\n" + " " + (" " + rowSep) * m + "  ") for j in range(1, n)],
17             ["{0:x}".format(n).upper(), self.greenWall, *(" |") *
18              (m - 1), " ", self.greenWall, "{0:x}".format(n).upper()],
19             [" ", *(" " + self.blueWall) * m, "  "],
20             [" ",  *[(" {0:x}".format(i).upper())
21                      for i in range(1, m + 1)], "  "],
22             ["Number of walls:"],
23             ["*X:"],
24             [" -P: ", wallNumb],
25             [" -Z: ", wallNumb],
26             ["*O:"],
27             [" -P: ", wallNumb],
28             [" -Z: ", wallNumb]
29         ]
```



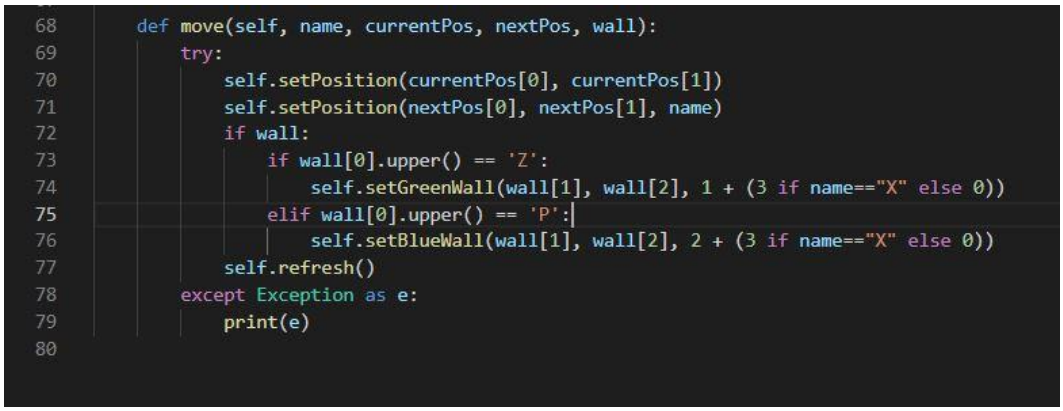
```
        ]

    def setPosition(self, i, j, placeholder=" ", refresh=False):
        try:
            self.template[i + 1] = self.template[i + 1][:j << 1] + \
                [placeholder] + self.template[i + 1][(j << 1) + 1:]
            if refresh:
                self.refresh()
        except Exception as e:
            print(e)

    def setBlueWall(self, i, j, wallNumbUpdate, refresh=False):
        try:
            self.template[i + 1] = self.template[i + 1][:(self.m + 2 + j) << 1] + [
                self.blueWall] + [" "] + [self.blueWall] + self.template[i + 1][((self.m + 2 + j) << 1) + 3:]
            self.template[-wallNumbUpdate][1]-=1
            if refresh:
                self.refresh()
        except Exception as e:
            print(e)

    def setGreenWall(self, i, j, wallNumbUpdate, refresh=False):
        try:
            self.template[i + 1] = self.template[i +
                                                 1][:(j << 1) + 1] + [self.greenWall] + self.template[i + 1][(j + 1) << 1:]
            self.template[i + 2] = self.template[i +
                                                 2][:(j << 1) + 1] + [self.greenWall] + self.template[i + 2][(j + 1) << 1:]
            self.template[-wallNumbUpdate][1]-=1
            if refresh:
                self.refresh()
        except Exception as e:
            print(e)

    def refresh(self):
        for r in self.template:
            for v in r:
                print(v, end="")
            print()
```



```
    def move(self, name, currentPos, nextPos, wall):
        try:
            self.setPosition(currentPos[0], currentPos[1])
            self.setPosition(nextPos[0], nextPos[1], name)
            if wall:
                if wall[0].upper() == 'Z':
                    self.setGreenWall(wall[1], wall[2], 1 + (3 if name=="X" else 0))
                elif wall[0].upper() == 'P':
                    self.setBlueWall(wall[1], wall[2], 2 + (3 if name=="X" else 0))
            self.refresh()
        except Exception as e:
            print(e)

```

Слика 3, 4, 5 Конструктор класе View
Функције класе View

Функцијом *setPosition(...)*, *setBlueWall(...)* и *setGreenWall(...)* врши се промена приказа тренутног стања табеле. Функцијом *refresh* освежава се приказ табле, док се позив функције *move* врши унутар класе *Table*, о којој ће убрзо бити више речи у тексту.

```
14
15      def onInit(self, initial):
16          connectInitialX = []
17          connectInitialO = []
18          for i in range(0, self.n):
19              self.fields.append([])
20              for j in range(0, self.m):
21                  connectedX = set(Table.createManhattan(
22                      (i, j), 0, self.n-1, self.m-1, 2))
23                  connectedO = set(Table.createManhattan(
24                      (i, j), 0, self.n-1, self.m-1, 2))
25
26                  self.fields[i].append(
27                      Field((i, j), connectedX, connectedO, initial.get((i+1, j+1), None)))
28
29                  match initial.get((i+1, j+1), None):
30                      case "X":
31                          manGen = Table.createManhattanGeneric(
32                              (i, j), 1, self.n, self.m, 1)
33                          connectInitialO += list(zip([(i, j)]
34                                                      * len(manGen), manGen))
35                      case "O":
36                          manGen = Table.createManhattanGeneric(
37                              (i, j), 1, self.n, self.m, 1)
38                          connectInitialX += list(zip([(i, j)]
39                                                      * len(manGen), manGen))
40
41          self.connectO(connectInitialO)
42          self.connectX(connectInitialX)
43
44          for pos in initial.keys():
45              self.view.setPosition(pos[0], pos[1], initial[pos])
46          self.view.refresh()
47
```

Слика 6 Постављање фигура на одговарајуће почетне позиције

```
12
13      def start(self, wallNumb, initial):
14          while self.next is None:
15              try:
16                  xPos = [x for x in initial.keys() if initial[x] == "X"]
17                  oPos = [o for o in initial.keys() if initial[o] == "O"]
18                  match input("X/o?\n"):
19                      case ("X" | "x"):
20                          self.X = Player(
21                              "X", False, wallNumb, xPos[0], xPos[1])
22                          self.O = Player(
23                              "O", True, wallNumb, oPos[0], oPos[1])
24                      case ("O" | "o"):
25                          self.O = Player(
26                              "O", True, wallNumb, oPos[0], oPos[1])
27                          self.X = Player(
28                              "X", False, wallNumb, xPos[0], xPos[1])
29                      case _:
30                          raise Exception("Invalid player selection input!")
31              except Exception as e:
32                  print(e)
33          self.next = self.X
34          self.play()
35
```

Слика 7 Функција за постављање почетног стања игре

За представљање почетног стања игре користи се функција *start*, чланица класе *Game*, приказана на слици 7.

У функцији *start* постављају се вредности за почетне позиције играча, играч X увек ће бити први на потезу када игра отпочне, док се кориснику пружа могућност одабира фигуре са којом ће играти игру (унос са тастатуре у 18. линија кода). Корисник уносом карактера "X"/"x" бира опцију да игра први, односно други уносом карактера "O"/"o".

Конструктор класе *Player* за први аргумент има *name* којим се води рачуна о томе ко је први на потезу, док се другим аргументом поставља флег који води рачуна о томе да ли је играч човек или рачунар (због потреба израде каснијих фаза пројекта), затим се редом као аргументи прослеђују број зидова за сваку од две боје зидова које добијају играчи на почетку игре и иницијалне позиције обе фигуре датог играча, као што се може видети на слици 7. Такође класа *Player* садржи две инстанце класе *GamePiece* која служи за дефинисање једне фигуре и један од атрибута ове класе је атрибут *home* који чува податак о почетној позицији фигуре датог играча. Због тога се фигуре (две фигуре по играчу) креирају као нови објекти типа *GamePiece* којима се прослеђују иницијалне позиције које се памте у класи *Player* (15. и 16. линија кода на слици 8). Други атрибут *position* памти информацију о тренутној позицији фигуре на табли.

```
4   class Player:
5       def __init__(self, name, isComputer=False, wallNumb=9, initialPos1st=None, initialPos2nd=None):
6           if name == "X":
7               initial1st = initialPos1st if initialPos1st else (4, 4)
8               initial2nd = initialPos2nd if initialPos2nd else (8, 4)
9           else:
10              initial1st = initialPos1st if initialPos1st else (4, 11)
11              initial2nd = initialPos2nd if initialPos2nd else (8, 11)
12          self.name = name
13
14          self.firstGP = GamePiece(initial1st)
15          self.secondGP = GamePiece(initial2nd)
16
17          self.noBlueWalls = wallNumb
18          self.noGreenWalls = wallNumb
19
20          self.isComputer = isComputer
21
```

Слика 8 Конструктор класе Player

На крају сваког потеза врши се испитивање да ли је игра окончана датим потезом позивом функције *checkState()* чија се имплементација може видети на слици 10. Испитује се да ли је управо одиграним потезом играч стао на иницијално поље противника, чиме се прекида игра, јер је он уједно и победник. То се проверава кроз позив функције *isWinner(...)* чланице класе *Player*. У оквиру функције *isWinner(...)* испитује се да ли се тренутна позиција играча налази у одговарајућој листи позиција. У позиву наведене функције унутар функције *checkState(...)* као

аргумент који представља описану листу прослеђују се иницијалне позиције играча O за играча X и обратно, позиције играча X за играча O.

```
    def play(self):
        while not self.winner:
            try:
                move = Game.parseMove(input(
                    f"{self.next.name} is on the move!\n"))
                if move and self.validation(move):
                    self.table.move(self.next.name, self.next.move(
                        move[0][1], move[1], move[2]), move[1], move[2])
                    self.next = self.X if self.next.name == self.O.name else self.O
                    self.checkState()
            except Exception as e:
                print(e)
        print(f"{self.winner.name} won! Congrats!")

    @staticmethod
    def parseMove(stream):
        try:
            ret = []
            m = stream.replace('[', '').replace(']', '').upper().split(' ')
            if m[0] not in ["X", "O"]:
                raise Exception("Invalid player ID!")
            if m[1] not in ['1', '2']:
                raise Exception("Invalid piece ID!")
            ret += [[m[0], int(m[1])]]
            if len(m) < 4:
                raise Exception("Missing positional coordinates!")
            ret += [tuple([ord(x)-55 if x>='A' else ord(x)-48 for x in m[2:4]])]
            if len(m) > 4:
                if m[4] not in ["Z", "P"]:
                    raise Exception("Invalid wall ID!")
                ret += [[m[4], *[ord(x)-55 if x>='A' else ord(x)-48 for x in m[5:7]]]]
            else:
                ret += [None]
            return ret
        except Exception as e:
            print(e)
            return []
```

Слика 9 Функција која обезбеђује приказ произвољног стања игре

```
    def checkState(self):
        if self.O.isWinner((self.X.firstGP.home, self.X.secondGP.home)):
            self.winner = self.O
        elif self.X.isWinner((self.O.firstGP.home, self.O.secondGP.home)):
            self.winner = self.X
```

Слика 10 Провера да ли је игра окончана

Функција којом се обезбеђује приказ произвољног стања игре је *play*, док функција *parseMove* служи да из уноса потеза са тастатуре очита вредности за одговарајуће параметре и уколико је дошло до некоректног уноса обавести корисника о томе, у супротном врати изглед самог потеза који је корисник одиграо.
У функцији *play* позива се функција *move* из класе *Table* којом се постављају зидови уколико их играч поседује и врши се промена места саме фигуре, позивом функције *move* класе *Player*. Такође у функцији обезбеђује се и промена редоследа

играња потеза, тј. уколико је на потезу последњи био играч X сада се поставља да је на потезу играч O и обратно.

Функција *move* класе *Table* позива остале функције чланице ове класе: *setBlueWall(...), setGreenWall(...)* и *setGamePiece(...)*. На слици 11 приказана је имплементација функције *move,* функције којом се мења тренутно стање на табли након сваког потеза играча постављањем одговарајућег зида (функције *setBlueWall(...)* и *setGreenWall(...)*) на прослеђену позицију као и померањем жељене фигуре. Функцијом *setGamePiece(...)* врши се постављање конекција за дату фигуру у зависности од тренутне и наредне позиције на табли. Већи део кода из имплементације функције је релевантан за касније фазе израде пројекта, међутим оно што је овде важно јесте да се за прослеђену фигуру остварују конекције, тј. овиме се ограничава њено кретање по табли. Тако да фигура може да пређе на поља чије је растојање једнако два (уколико то поље није заузето), при чему се израчунавање врши по Manhattan Pattern-у. Manhattan Pattern израчунава растојање између два вектора без коришћења функција квадрирања и кореновања.

У функцији приказаној на слици 12, такође се кроз конекције обезбеђује да уколико се на позицији која јесте правилно одиграна као нови потез, налази нека друга фигура, померање тренутне фигуре на суседну позицију (позиција која се налази између тренутне и жељене) буде могуће. Ова функција у суштини прави нове и укида неке од постојећих веза противнику због услова да играч може да се креће по хоризонтали и вертикали и за по једно поље у случају да је жељено поље заузето, међутим ово ће бити објашњено са више детаља касније у тексту.

```
184
185     def move(self, name, currentPos, nextPos, wall=None):
186         if wall:
187             if wall[0] == "Z":
188                 self.setGreenWall((wall[1], wall[2]))
189             if wall[0] == "P":
190                 self.setBlueWall((wall[1], wall[2]))
191         self.setGamePiece(currentPos, nextPos, name)
192         self.view.move(name, currentPos, nextPos, wall)
```

Слика 11 Функција која садржи логику одигравања потеза на табли

```
114
115     def setGamePiece(self, prevPos, position, name="X"):
116         forConnect = list(map(lambda x: (x, (x[0] - position[0], x[1] - position[1])),
117                               Table.createManhattan(position, 1, self.n, self.m, 1)))
118         forConnect = list(filter(lambda x: x[0][0] + x[1][0] > 0 and x[0][0] + x[1][0] <=
119                          self.n and x[0][1] + x[1][1] > 0 and x[0][1] + x[1][1] <= self.m, forConnect))
120         forPrevConnect = list(map(lambda x: (x, (x[0] - prevPos[0], x[1] - prevPos[1])),
121                                   Table.createManhattan(prevPos, 1, self.n, self.m, 1)))
122         forPrevConnect = list(filter(lambda x: x[0][0] + x[1][0] > 0 and x[0][0] + x[1][0] <=
123                               self.n and x[0][1] + x[1][1] > 0 and x[0][1] + x[1][1] <= self.m, forPrevConnect))
124         forDisconnect = Table.createManhattanGeneric(
125             position, 1, self.n, self.m, 2)
126         forDisconnect = list(zip([position]*len(forDisconnect), forDisconnect))
127         forPrevDisconnect = Table.createManhattanGeneric(
128             prevPos, 1, self.n, self.m, 2)
129         forPrevDisconnect = list(
130             zip([prevPos]*len(forPrevDisconnect), forPrevDisconnect))
131
132         if name == "X":
133             self.disconnectO(forDisconnect + forPrevConnect)
134             self.connectO(forConnect + forPrevDisconnect)
135         else:
136             self.disconnectX(forDisconnect + forPrevConnect)
137             self.connectX(forConnect + forPrevDisconnect)
```

Слика 12 Функција која ограничава кретање пешака

На слици 13 може се видети изглед функције *move* класе *Player*. Повратна вредност функције је стање у којем се фигура налазила пре одигравања новог потеза играча. Ако играч остане без зидова потез може да садржи само одабир фигуре и нову позицију фигуре, што се дефинише променљивом *wall*. У случају да играч још увек поседује зидове у зависности од прослеђене боје зида укупан број зидова те боје се декрементира за један.

```
22
23     def move(self, pieceNum, positon, wall=None):
24         prevPos = None
25         if pieceNum == 1:
26             prevPos = self.firstGP.position
27             self.firstGP.position = positon
28         else:
29             prevPos = self.secondGP.position
30             self.secondGP.position = positon
31
32         if self.noGreenWalls + self.noBlueWalls < 0:
33             wall = None
34
35         if wall != None:
36             if wall[0].upper() == "Z":
37                 self.noGreenWalls -= 1
38             elif wall[0].upper() == "P":
39                 self.noBlueWalls -= 1
40         return prevPos
```

Слика 13 Функција којом се мења стање играча

```
    def setBlueWall(self, pos):
        if self.isCorrectBlueWall(pos):
            self.blueWalls.add(pos)
            forDisconnect = []
            up1 = pos[0] - 1 > 0
            down1 = pos[0] + 1 <= self.n
            down2 = pos[0] + 2 <= self.n
            left1 = pos[1] - 1 > 0
            right1 = pos[1] + 1 <= self.m
            right2 = pos[1] + 2 <= self.m

            if down1:
                forDisconnect += [(pos, (1, 0))]
                if right1:
                    forDisconnect += [(pos, (1, 1))]
                    forDisconnect += [((pos[0], pos[1] + 1), (1, -1)),
                                      ((pos[0], pos[1] + 1), (1, 0))]
                    if down2:
                        forDisconnect += [((pos[0], pos[1] + 1), (2, 0))]
                    if up1:

                        forDisconnect += [((pos[0] + 1, pos[1] + 1), (-2, 0))]
                if up1:
                    forDisconnect += [((pos[0] + 1, pos[1]), (-2, 0))]
                if down2:
                    forDisconnect += [(pos, (2, 0))]
                if left1 and ((pos[0], pos[1]-2) in self.blueWalls or (pos[0]-1, pos[1]-1) in self.greenWalls):
                    forDisconnect += [(pos, (1, -1)),
                                      ((pos[0]+1, pos[1]), (-1, -1))]
                if right2 and ((pos[0], pos[1]+2) in self.blueWalls or (pos[0]-1, pos[1]+1) in self.greenWalls):
                    forDisconnect += [((pos[0], pos[1]+1), (1, 1)),
                                      ((pos[0]+1, pos[1]+1), (-1, 1))]

            self.disconnect(forDisconnect, "P")
```

Слика 14 Функција за постављање плавих зидова

```
    def setGreenWall(self, pos):
        if self.isCorrectGreenWall(pos):
            self.greenWalls.add(pos)
            forDisconnect = []
            up1 = pos[0] - 1 > 0
            down1 = pos[0] + 1 <= self.n
            down2 = pos[0] + 2 <= self.n
            left1 = pos[1] - 1 > 0
            right1 = pos[1] + 1 <= self.m
            right2 = pos[1] + 2 <= self.m

            if right1:
                forDisconnect += [(pos, (0, 1))]
                if down1:
                    forDisconnect += [(pos, (1, 1))]
                    forDisconnect += [((pos[0] + 1, pos[1]), (-1, 1)),
                                      ((pos[0] + 1, pos[1]), (0, 1))]
                    if left1:
                        forDisconnect += [((pos[0] + 1, pos[1] + 1), (0, -2))]
                    if down2 and ((pos[0]+2, pos[1]) in self.greenWalls or (pos[0]+1, pos[1]-1) in self.blueWalls):
                        forDisconnect += [((pos[0]+1, pos[1]), (1, 1)),
                                          ((pos[0] + 1, pos[1] + 1), (1, -1))]
                    forDisconnect += [((pos[0], pos[1] + 1), (0, -2))]
                if up1 and ((pos[0]-2, pos[1]) in self.greenWalls or (pos[0]-1, pos[1]-1) in self.blueWalls):
                    forDisconnect += [(pos, (-1, 1)),
                                      ((pos[0], pos[1] + 1), (-1, -1))]
                if right2:
                    forDisconnect += [(pos, (0, 2))]
                    if down1:
                        forDisconnect += [((pos[0] + 1, pos[1]), (0, 2))]
            self.disconnect(forDisconnect, "Z")
```

Слика 15 Функција за постављање зелених зидова

Уколико је валидна позиција на коју корисник жели да постави зид функцијама *setBlueWall(...)* и *setGreenWall(...)* је то могуће и одрадити. Испитивање валидности жељене позиције на коју би ишао зид врши се функцијама класе *Table*, приказаним на слици 14. Остатак код коришћеног у поменутим функцијама биће описан у некој од каснијих фаза израде пројекта, због тога што није релевантан за дефинисање проблема постављених фазом 1.

```
172     def isCorrectBlueWall(self, pos):
173         return not (pos in self.greenWalls or [x for x in [(pos[0], pos[1] - 1), pos, (pos[0], pos[1] + 1)] if x in self.blueWalls])
174
175     def isCorrectGreenWall(self, pos):
176         return not (pos in self.blueWalls or [x for x in [(pos[0] - 1, pos[1]), pos, (pos[0] + 1, pos[1])] if x in self.greenWalls])
177
```

Слика 16 Функције за испитивање валидности нових позиција зида

Дакле функција *isCorrectBlueWall* враћа вредност True уколико се прослеђена позиција (скуп (врста, колона)) зида не налази у листи позиција (листа садржи скупове(врста, колона) за сваку позицију на којој већ постоји зид на табли) постављених зидова зелене боје, као и плаве боје и уколико се позиција у жељеној врсти: (врста, колона + 1), односно (врста, колона - 1) не налази у листи постављених зидова плаве боје. Слично и за проверу постављања новог зеленог зида, разлика је у томе што се у овом случају испитује да ли позиције (врста -1, колона) и (врста + 1, колона) не припадају листи постављених зидова зелене боје.

```
38      def validation(self, move):
39          try:
40              if self.next.name != move[0][0]:
41                  raise Exception("Not your turn!")
42              if self.table.n < move[1][0] or move[1][0] < 1:
43                  raise Exception("Row index out of bounds!")
44              if self.table.m < move[1][1] or move[1][1] < 1:
45                  raise Exception("Column index out of bounds!")
46              if move[1] == (self.next.firstGP.position if move[0][1] == 1 else self.next.secondGP.position):
47                  raise Exception("You're already on that position!")
48              if move[1] == (self.next.firstGP.position if move[0][1] == 2 else self.next.secondGP.position):
49                  raise Exception("Can't step on your pieces!")
50              if not move[2] and self.next.noBlueWalls + self.next.noGreenWalls > 0:
51                  raise Exception("You didn't put up a wall!")
52              if move[2]:
53                  if move[2][1] > self.table.n-1 or move[2][1] < 1:
54                          raise Exception("Wall row index out of bounds!")
55                  if move[2][2] > self.table.m-1 or move[2][2] < 1:
56                      raise Exception("Wall column index out of bounds!")
57                  if move[2][0] == "Z":
58                      if self.next.noGreenWalls < 1:
59                          raise Exception(
60                              "You don't have any green walls left to place...")
61                      if not self.table.isCorrectGreenWall((move[2][1], move[2][2])):
62                          raise Exception(
63                              "Green wall cannot be set on the given position!")
64                  elif move[2][0] == "P":
65                      if self.next.noGreenWalls < 1:
66                          raise Exception(
67                              "You don't have any blue walls left to place...")
68                      if not self.table.isCorrectBlueWall((move[2][1], move[2][2])):
69                          raise Exception(
70                              "Blue wall cannot be set on the given position!")
71              if not self.table.areConnected(self.next.firstGP.position if move[0][1] == 1 else self.next.secondGP.position, move[1], move[0][0]):
72                  raise Exception("Invalid move!")
73          except Exception as e:
74              print(e)
75              return False
76          return True
```

Слика 17 Функција правилности потеза

Такође, испитивање валидности сваког потеза (померај пешака, позиције зидова, унос потеза, …) одрађено је унутар функције *validation(...)* и то се може видети на слици 14. Уколико је корисник одиграо потез у којем је за индекс врсте, односно колоне поставио вредност која је већа од димензија табле, односно мања од јединице, апликација ће пријавити грешку. Грешка ће се јавити у следећим случајевима: корисник жели да одигра потез у којем за нову позицију наводи тренутну позицију своје фигуре, или наводи иницијалну позицију дате фигуре; играч још увек има зидове, а у потезу није навео боју и позицију зида коју жели да постави на табли; уколико играч у потезу наводи зид који жели да постави на одређену позицију, а зидова тражене боје више нема или је за позицију зида унео параметре који излазе из опсега (о овоме је раније било речи кроз објашњења дата за функције *isCorrectBlueWall(...)* и *isCorrectGreenWall(...)*) и на самом крају испитује се да ли део потеза корисника који се односи на померај фигуре прати правила кретања по табли (по два поља лево односно десно гледано за врсту табле, по два поља доле односно горе гледано за колону табле, по једно поље гледано по дијагонали у односу на тренутну позицију или по једно поље уколико је потез валидан, а жељено поље које представља нову позицију заузето).