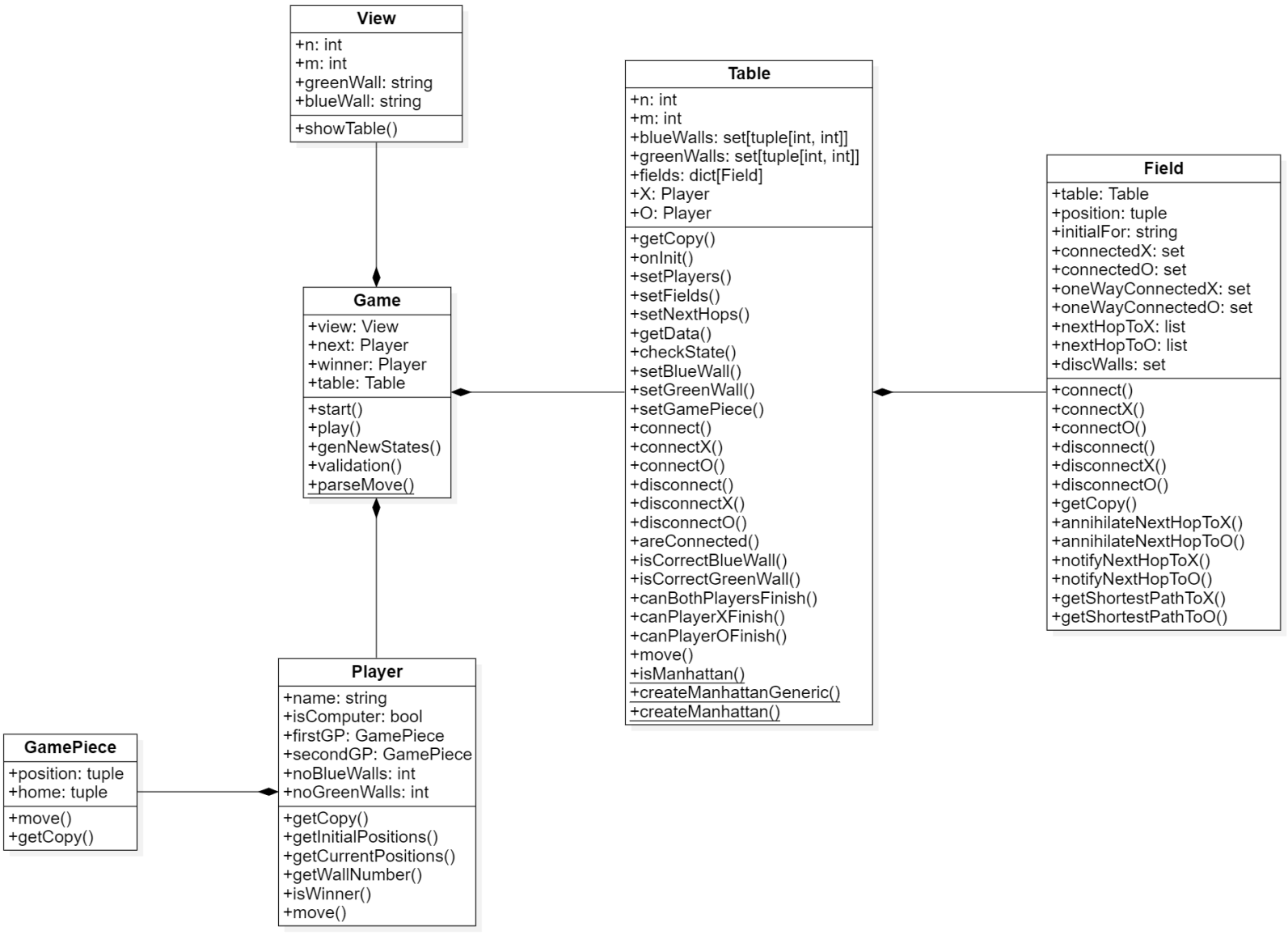
Извештај пројекта „Blockade”

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. С обзиром да су поједине класе претрпеле промене у односу на прву фазу, истаћи ћемо их уколико сматрамо да су битне за нагласити.



Слика 1 Класни дијаграм

1. Фаза – Формулација проблема и интерфејс
2. Фаза – Оператори промене стања

За потребе дефинисања стања проблема игре користи се класа *Game* која садржи инстанцу класе *Table*, која је искоришћена за представљање стања игре, инстанцу класе *View*, која се користи за приказ произвољног стања игре као и две референце играча који је следећи на потезу, односно оног који ће евентуално бити победник игре, оба показивача су иницирана на *None*.

```
class Game:
    def __init__(self, n=11, m=14, greenWall="\u01c1", blueWall="\u2550", rowSep="\u23AF"):
        self.table = Table(n, m)
        self.view = View(n, m, greenWall, blueWall, rowSep)
        self.next = None
        self.winner = None
```

Слика 2 Конструктор класе Game

У оквиру класе *View*, као што се може видети на слици 3, врши се иницијализација атрибута *n* и *m*, који представљају димензију табеле, односно број врста и колона, *Unicode* вредности који ће се користити за приказ одговарајућих зидова и на крају шаблон табеле, који ће се попуњавати по потреби, како би се приказало произвољно стање игре.

```
class View:
    def __init__(self, n=11, m=14, greenWall="\u01c1", blueWall="\u2550", rowSep="\u23AF"):
        self.n = n
        self.m = m
        self.greenWall = greenWall
        self.blueWall = blueWall

        self.template = [
            [" ",  *[(" " + (chr(j+48) if j < 10 else chr(j+55)))
                     for j in range(1, m + 1)], "  "],
            [" ", *(" " + self.blueWall) * m, "  "],
            *[list((chr(i+48) if i < 10 else chr(i+55)) + self.greenWall + (" |") * (m - 1) + " " + self.greenWall +
                   (chr(i+48) if i < 10 else chr(i+55)) + "\n" + " " + (" " + rowSep) * m + "  ") for i in range(1, n)],
            [(chr(n+48) if n < 10 else chr(n+55)), self.greenWall, *(" |") *
             (m - 1), " ", self.greenWall, (chr(n+48) if n < 10 else chr(n+55))],
            [" ", *(" " + self.blueWall) * m, "  "],
            [" ",  *[(" " + (chr(j+48) if j < 10 else chr(j+55)))
                     for j in range(1, m + 1)], "  "]
        ]
```

Слика 3 Конструктор класе View

Класу *View* користимо само у сврхе представљања изгледа читаве игре у конзоли. Поред конструктора, класа садржи и функцију *showTable(...)* приказану на слици којом се уз помоћ шаблона приказује тренутно стање игре. Изглед функције је дат на слици 4.

```
def showTable(self, greenWalls, blueWalls, players, xWallNumb=(9, 9), oWallNumb=(9, 9)):
    table = list()
    for t in self.template:
        table.append(list(t))

    for gw in greenWalls:
        table[gw[0] + 1] = table[gw[0] + 1][:(gw[1] << 1) + 1] + [
            self.greenWall] + table[gw[0] + 1][(gw[1] + 1) << 1:]
        table[gw[0] + 2] = table[gw[0] + 2][:(gw[1] << 1) + 1] + [
            self.greenWall] + table[gw[0] + 2][(gw[1] + 1) << 1:]

    for bw in blueWalls:
        table[bw[0] + 1] = table[bw[0] + 1][:(self.m + 2 + bw[1]) << 1] + [self.blueWall] + [" "] + [self.blueWall] + table[bw[0] + 1][((self.m + 2 + bw[1]) << 1) + 3:]

    for player in players.keys():
        for i in range(2):
            table[players[player][i][0] + 1] = table[players[player][i][0] + 1][:players[player][i][1] << 1] + [player] + table[players[player][i][0] + 1][(players[player][i][1] << 1) + 1:]

    for r in table:
        for v in r:
            print(v, end="")
        print()
    print("\n\tWalls X\t\t|\tWalls O")
    print(
        f"\tB: {xWallNumb[0]}\t\t|\tB: {oWallNumb[0]}")
    print(
        f"\tG: {xWallNumb[1]}\t\t|\tG: {oWallNumb[1]}\n")
```

Слика 4 функција за приказ произвољног стања игре

```
class Table:
    def __init__(self, n=11, m=14, blueWalls=set(), greenWalls=set(), X=None, O=None):
        self.n = n
        self.m = m
        self.blueWalls = set(blueWalls)
        self.greenWalls = set(greenWalls)
        self.fields = dict()
        self.X = X.getCopy() if X else None
        self.O = O.getCopy() if O else None

    def getCopy(self):
        copy = Table(self.n, self.m, self.blueWalls,
                     self.greenWalls, self.X, self.O)
        for key in self.fields.keys():
            copy.fields[key] = self.fields[key].getCopy(copy)
        return copy

    def getData(self):
        return (self.greenWalls, self.blueWalls, {"X": self.X.getCurrectPositions(), "O": self.O.getCurrectPositions()}, self.X.getWallNumber(), self.O.getWallNumber())

    def onInit(self, initial, players):
        self.setPlayers(players)
        self.setFields(initial)
        self.setNextHops(initial)
```

Слика 5 конструктор класе Table

Класа *Table*, као што је већ назначено се користи за представљање стања и у складу са тим је модификована како би се лакше обавило њено копирање. Унутар конструктора класе се иницијализују димензије табеле, скупови позиција плавих и зелених зидова, расута таблица је искоришћена за бележење стања појединачних поља табеле и индексирана је самом позицијом поља у табели како би нам олакшало идентификацију поља, с обзиром да индекси низова крећу од 0. Као што се може приметити, инстанца *View* класе је пребачена у оквиру *Game* класе, док су овде убачене инстанце играча X и O, како би се водила евиденција о њиховом стању.

Функције *getCopy(...)* су написане у оквиру сваке класе чије инстанце је потребно копирати, како би се вршило копирање референтних података у *Python*-у без употребе библиотеке за *deepcopy*. Такође, функција *getData(...)* је специфична за Table класу јер се користи у пару са *showTable(...)* методом како би се на лак начин пренели неопходни подаци за приказ.

У оквиру класе *Table*, дефинисана је функција *onInit(...)* која се позива једино при инстанцирању објекта ове класе у оквиру *Game*-а. Функција подешава иницијална стања играча, вредности поља, као и вредности *NextHop* атрибута уз помоћ којих је имплементиран *Distance Vector* алгоритам за брзо проналажење путања до циљних поља за одговарајућег играча.

```
def setPlayers(self, players):
    for player in players.keys():
        if player == "X":
            self.X = Player("X", *players[player])
        elif player == "O":
            self.O = Player("O", *players[player])
```

```
def setNextHops(self, initial):
    for pos in initial.keys():
        match initial[pos]:
            case "X1":
                self.fields[pos].nextHopToX[0] = (pos, 0)
                self.fields[pos].notifyNextHopToX(self.fields[pos], 0, True)
            case "X2":
                self.fields[pos].nextHopToX[1] = (pos, 0)
                self.fields[pos].notifyNextHopToX(self.fields[pos], 1, True)
            case "O1":
                self.fields[pos].nextHopToO[0] = (pos, 0)
                self.fields[pos].notifyNextHopToO(self.fields[pos], 0, True)
            case "O2":
                self.fields[pos].nextHopToO[1] = (pos, 0)
                self.fields[pos].notifyNextHopToO(self.fields[pos], 1, True)
```

```
def setFields(self, initial):
    connectInitialX = []
    connectInitialO = []
    for i in range(1, self.n + 1):
        for j in range(1, self.m + 1):
            pos = (i, j)
            connected = set(Table.createManhattan(
                pos, 1, self.n, self.m, 2))
            self.fields[pos] = Field(self, pos, initial.get(
                pos, None), connected, connected)
            match initial.get(pos, None):
                case "X1" | "X2":
                    manGen = list(map(lambda x: ((pos[0] - x[0], pos[1] - x[1]), x), Table.createManhattanGeneric(pos, 1, self.n, self.m, 1)))
                    connectInitialO += list(filter(lambda x: 0 < x[0][0] <= self.n and 0 < x[0][0] + x[1][0] <= self.n and 0 < x[0][1] <= self.m and 0 < x[0][1] + x[1][1] <= self.m, manGen))
                case "O1" | "O2":
                    manGen = list(map(lambda x: ((pos[0] - x[0], pos[1] - x[1]), x), Table.createManhattanGeneric(pos, 1, self.n, self.m, 1)))
                    connectInitialX += list(filter(lambda x: 0 < x[0][0] <= self.n and 0 < x[0][0] + x[1][0] <= self.n and 0 < x[0][1] <= self.m and 0 < x[0][1] + x[1][1] <= self.m, manGen))
    for k in initial.keys():
        self.setGamePiece((-100, -100), (k[0], k[1]), initial.get(k)[0])
    self.connectO(connectInitialO, False)
    self.connectX(connectInitialX, False)
```

Слика 6, 7, 8 Функције setPlayers(), setNextHops() і setFields() у оквиру класе Table

```
def start(self, initial):
    xPos = [x for x in initial.keys() if initial[x] in ["X1", "X2"]]
    oPos = [o for o in initial.keys() if initial[o] in ["O1", "O2"]]
    wallNumber = (initial['wallNumber'], initial['wallNumber'])
    initial.pop('wallNumber')
    self.view.showTable(self.table.greenWalls, self.table.blueWalls, {"X": xPos, "O": oPos}, wallNumber, wallNumber)
    while not self.next:
        match input("X/o?\n"):
            case ("X" | "x"):
                playerInfo = {"X": (False, wallNumber, xPos, xPos), "O": (True, wallNumber, oPos, oPos)}
            case ("O" | "o"):
                playerInfo = {"X": (True, wallNumber, oPos, oPos), "O": (False, wallNumber, oPos, oPos)}
            case _:
                print("Invalid player selection input!")
                continue
        self.table.onInit(initial, playerInfo)
        self.next = self.table.X
        newStates = self.genNewStates()
        ind = 1
        print('Done')
        for ns in newStates:
            print(ind, ns)
            ind += 1
    self.play()
```

Слика 9 Функција за постављање почетног стања игре

За представљање почетног стања игре користи се функција *start(...)*, чланица класе *Game*, приказана на слици 9.

У функцији *start(...)* постављају се вредности за почетне позиције играча, као и број зидова које играч поседује. Играч X увек ће бити први на потезу када игра отпочне, док се кориснику пружа могућност одабира фигуре са којом ће играти игру. Корисник уносом карактера “X”/”x” бира опцију да игра први, односно други уносом карактера “O”/”o”.

Конструктор класе *Player* је такође претрпео ситније измене у складу са *getCopy(...)*, а ради прегледности су придодати и *getter*-и за одговарајуће вредности коју складиште инстанце ове класе, то су: иницијална поља играча, односно крајња поља на која треба да стигне противник, тренутна позиција сваке од фигура којима управља играч, као и број преосталих зидова.

```
class Player:
    def __init__(self, name, isComputer=False, wallNumb=(9, 9), initialPositions=(None, None), currentPositions=(None, None)):
        self.name = name
        self.firstGP = GamePiece(initialPositions[0], currentPositions[0])
        self.secondGP = GamePiece(initialPositions[1], currentPositions[1])
        self.noBlueWalls = wallNumb[0]
        self.noGreenWalls = wallNumb[1]
        self.isComputer = isComputer

    def getCopy(self):
        return Player(self.name, self.isComputer, self.getWallNumber(), self.getInitialPositions(), self.getCurrectPositions())

    def getInitialPositions(self):
        return (self.firstGP.home, self.secondGP.home)

    def getCurrectPositions(self):
        return (self.firstGP.position, self.secondGP.position)

    def getWallNumber(self):
        return (self.noBlueWalls, self.noGreenWalls)
```

Слика 10 Конструктор класе Player са getter-има

На крају сваког потеза врши се испитивање да ли је игра окончана датим потезом позивом функције *checkState(...)* чија се имплементација може видети на слици 11. 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 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 11 Провера да ли је игра окончана

```
def play(self):
    while not self.winner:
        parsedMove = Game.parseMove(input(f"{self.next.name} is on the move!\n"))
        if parsedMove[0]:
            move = parsedMove[1]
            validated = self.validation(move)
            if validated[0]:
                self.table.move(self.next.name, self.next.move(move[0][1], move[1], move[2]), move[1], move[2])
                self.view.showTable(*self.table.getData())
                self.next = self.table.X if self.next.name == self.table.O.name else self.table.O
                self.table.checkState()
            else:
                print(validated[1])
        else:
            print(parsedMove[1])
    print(f"{self.winner.name} won! Congrats!")
```

Слика 12 Функција која обезбеђује играње партије

Функција којом се обезбеђује играње саме партије 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 13 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. 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. Сваким позивом ове функције врши се и испитивање да ли је дошло до сценарија где је неко од иницијалних поља фигура ограђено и уколико јесте враћа се порука о томе.

```
def move(self, name, currentPos, nextPos, wall=None):
    if wall:
        if wall[0] == "Z":
            self.setGreenWall((wall[1], wall[2]))
        if wall[0] == "P":
            self.setBlueWall((wall[1], wall[2]))
    self.setGamePiece(currentPos, nextPos, name)
    if self.canPlayerXFinish():
        xPos1 = self.X.firstGP.position
        xPos2 = self.X.secondGP.position
        print("Shortest path X1 to O1:", self.fields[xPos1].getShortestPathToO(0))
        print("Shortest path X1 to O2:", self.fields[xPos1].getShortestPathToO(1))
        print("Shortest path X2 to O1:", self.fields[xPos2].getShortestPathToO(0))
        print("Shortest path X2 to O2:", self.fields[xPos2].getShortestPathToO(1))
    else:
        print("Player X can't finish!")
    if self.canPlayerOFinish():
        oPos1 = self.O.firstGP.position
        oPos2 = self.O.secondGP.position
        print("Shortest path O1 to X1:", self.fields[oPos1].getShortestPathToX(0))
        print("Shortest path O1 to X2:", self.fields[oPos1].getShortestPathToX(1))
        print("Shortest path O2 to X1:", self.fields[oPos2].getShortestPathToX(0))
        print("Shortest path O2 to X2:", self.fields[oPos2].getShortestPathToX(1))
    else:
        print("Player O can't finish!")
```

Слика 13 Функција која садржи логику одигравања потеза на табли

У функцији приказаној на слици 14, 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.

Дакле, када се играч помери, прослеђују се параметри *prevPos* и *position* како би се генерисали одгоаварајући низови који садрже поља које је неопходно раскинути односно повезати у односу на поља којим се креће противник. Овим се покривају крајњи случаји када противник заблокира пут играчу, па се овом пружа могућност кретања за 1 по вертикали или хоризонтали. Одговарајућим комбиновањем ових листи, за које се позивају *connect(...)* и *disconnect(...)* односно *connectX(...)* и *disconnectX(...)* , *connectO(...)* и *disconnectO(...)*.

```
def setGamePiece(self, prevPos, position, name="X"):
    forConnect = list(map(lambda x: ((position[0] - x[0] * 2, position[1] - x[1] * 2), x), Table.createManhattanGeneric(position, 1, self.n, self.m, 1)))
    forConnect = list(filter(lambda x: 0 < x[0][0] <= self.n and 0 < x[0][0] + x[1][0] <= self.n and 0 < x[0][1] <= self.m and 0 < x[0][1] + x[1][1] <= self.m, forConnect))
    forPrevConnect = list(map(lambda x: ((prevPos[0] - x[0] * 2, prevPos[1] - x[1] * 2), x), Table.createManhattanGeneric(prevPos, 1, self.n, self.m, 1)))
    forPrevConnect = list(filter(lambda x: 0 < x[0][0] <= self.n and 0 < x[0][0] + x[1][0] <= self.n and 0 < x[0][1] <= self.m and 0 < x[0][1] + x[1][1] <= self.m, forPrevConnect))
    forDisconnect = Table.createManhattanGeneric(position, 1, self.n, self.m, 2)
    forDisconnect = list(zip([position]*len(forDisconnect), forDisconnect))
    forPrevDisconnect = Table.createManhattanGeneric(prevPos, 1, self.n, self.m, 2)
    forPrevDisconnect = list(zip([prevPos]*len(forPrevDisconnect), forPrevDisconnect))
    if name == "X":
        self.disconnectO(forDisconnect + forPrevConnect)
        self.connectO(forConnect, False, position)
        self.connectO(forPrevDisconnect)
    else:
        self.disconnectX(forDisconnect + forPrevConnect)
        self.connectX(forConnect, False, position)
        self.connectX(forPrevDisconnect)
```

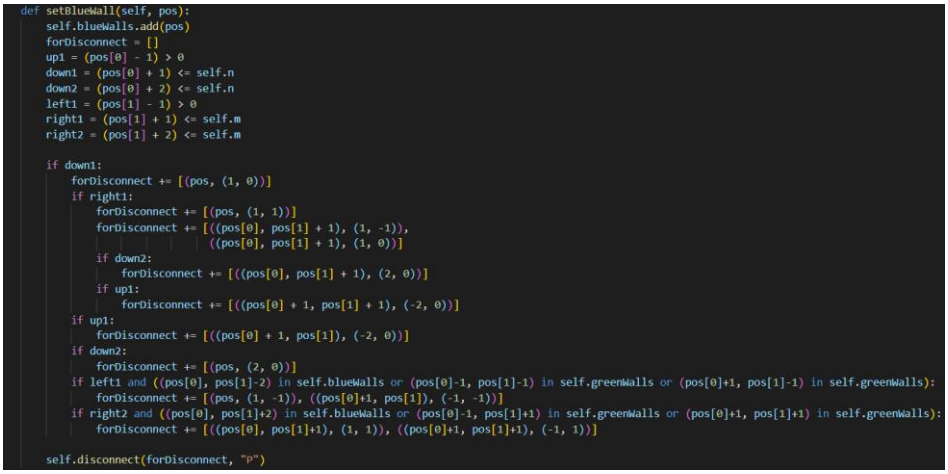
Слика 14 Функција која ограничава кретање пешака

На слици 15 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.

```
def isWinner(self, positions):
    return self.firstGP.position in positions or self.secondGP.position in positions

def move(self, pieceNum, positon, wall=None):
    if wall != None:
        if wall[0].upper() == "Z":
            self.noGreenWalls -= 1
        elif wall[0].upper() == "P":
            self.noBlueWalls -= 1
    if pieceNum == 1:
        return self.firstGP.move(positon)
    else:
        return self.secondGP.move(positon)
```

Слика 15 Функција којом се мења стање играча

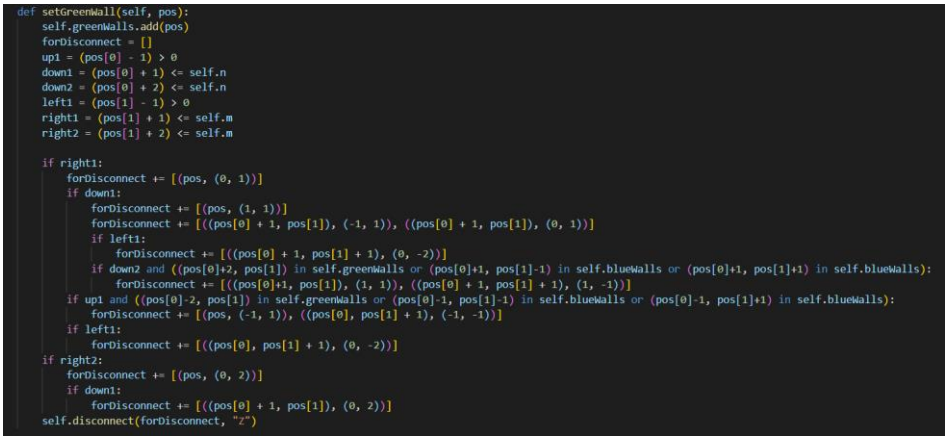


```
def setBlueWall(self, pos):
    self.blueWalls.add(pos)
    forDisconnect = []
    up1 = (pos[0] - 1) > 0
    down1 = (pos[0] + 1) <= self.n
    down2 = (pos[0] + 2) <= self.n
    left1 = (pos[1] - 1) > 0
    right1 = (pos[1] + 1) <= self.m
    right2 = (pos[1] + 2) <= self.m

    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 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 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 16 Функција за постављање плавих зидова



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

Слика 17 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 сликама 16 и 17. Такође постављањем зидова на одређене позиције смањује се број могућих потеза који следе након одигравања тренутног, јер није валидно преклапање зидова, или пролазак пешака кроз зидове. То се контролише позивом функције *disconnect(...)* након постављања зида на задату позицију.

```
def areConnected(self, currentPos, followedPos, name="X"):
    if name == "X":
        return followedPos in self.fields[currentPos].connectedX
    else:
        return followedPos in self.fields[currentPos].connectedO

def isCorrectBlueWall(self, pos):
    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])

def isCorrectGreenWall(self, pos):
    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])
```

Слика 18 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.

```
def validation(self, move):
    if self.next.name != move[0][0]:
        return (False, "Not your turn!")
    if self.table.n < move[1][0] or move[1][0] < 1:
        return (False, "Row index out of bounds!")
    if self.table.m < move[1][1] or move[1][1] < 1:
        return (False, "Column index out of bounds!")
    if move[1] == (self.next.firstGP.position if move[0][1] == 1 else self.next.secondGP.position):
        return (False, "You're already on that position!")
    if move[1] == (self.next.firstGP.position if move[0][1] == 2 else self.next.secondGP.position):
        return (False, "Can't step on your pieces!")
    if not move[2] and self.next.noBlueWalls + self.next.noGreenWalls > 0:
        return (False, "You didn't put up a wall!")
    if move[2]:
        if move[2][1] > self.table.n-1 or move[2][1] < 1:
            return (False, "Wall row index out of bounds!")
        if move[2][2] > self.table.m-1 or move[2][2] < 1:
            return (False, "Wall column index out of bounds!")
        if move[2][0] == "Z":
            if self.next.noGreenWalls < 1:
                return (False, "You don't have any green walls left to place...")
            if not self.table.isCorrectGreenWall((move[2][1], move[2][2])):
                return (False, "Green wall cannot be set on the given position!")
        elif move[2][0] == "P":
            if self.next.noBlueWalls < 1:
                return (False, "You don't have any blue walls left to place...")
            if not self.table.isCorrectBlueWall((move[2][1], move[2][2])):
                return (False, "Blue wall cannot be set on the given position!")
    if not self.table.areConnected(self.next.firstGP.position if move[0][1] == 1 else self.next.secondGP.position, move[1], move[0][0]):
        return (False, "Invalid move!")
    return (True, "Valid move!")
```

Слика 19 Функција провере 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 19. 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 поруку да нешто није у реду са унетим потезом. Порука о грешки 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).

У класи *Field* постоје функције које касније користи класа *Table* и управо ове чланице класе *Table* представљају већи део логике која стоји иза сваког потеза: померања фигуре и/или постављања зидова. Функцијама *connect* і *disconnect* појединачних фигура омогућава се додавање/укидање веза за поједина поља у зависности од тога где је фигура померена и где је зид постављен. Уколико се постави зид на одређено поље, оно ће изгубити везу са околним пољима и тиме је спречено „прескакање“ зидова у оквиру игре, односно ако се фигура постави на одређено поље добија конекције са пољима која су по хоризонтали, односно вертикали на удаљености два, или удаљености 1 за случај кретања по дијагонали.

```
def disconnect(self, vals, w=None):
    for (x, y) in vals:
        self.fields[x].disconnect(
            self.fields[(x[0] + y[0], x[1] + y[1])], w)
    for (x, y) in vals:
        self.fields[x].notifyNextHopToX(None, 0, False)
        self.fields[x].notifyNextHopToX(None, 1, False)
        self.fields[x].notifyNextHopToO(None, 0, False)
        self.fields[x].notifyNextHopToO(None, 1, False)

def disconnectX(self, vals):
    for (x, y) in vals:
        self.fields[x].disconnectX(
            self.fields[(x[0] + y[0], x[1] + y[1])])
    for (x, y) in vals:
        self.fields[x].notifyNextHopToO(None, 0, False)
        self.fields[x].notifyNextHopToO(None, 1, False)

def disconnectO(self, vals):
    for (x, y) in vals:
        self.fields[x].disconnectO(
            self.fields[(x[0] + y[0], x[1] + y[1])])
    for (x, y) in vals:
        self.fields[x].notifyNextHopToX(None, 0, False)
        self.fields[x].notifyNextHopToX(None, 1, False)
```

```
def connect(self, vals):
    for (x, y) in vals:
        self.fields[x].connect(self.fields[(x[0] + y[0], x[1] + y[1])])
    for (x, y) in vals:
        f = self.fields[(x[0] + y[0], x[1] + y[1])]
        self.fields[x].notifyNextHopToX(f, 0, False)
        self.fields[x].notifyNextHopToX(f, 1, False)
        self.fields[x].notifyNextHopToO(f, 0, False)
        self.fields[x].notifyNextHopToO(f, 1, False)

def connectX(self, vals, mirrored=True, position=None):
    if position:
        con = [(x, y) for (x, y) in vals if x
               in self.fields[position].connectedO]
        for (x, y) in con:
            self.fields[x].connectX(self.fields[(x[0] + y[0], x[1] + y[1])], mirrored)
        for (x, y) in vals:
            f = self.fields[(x[0] + y[0], x[1] + y[1])]
            self.fields[x].notifyNextHopToO(f, 0, False)
            self.fields[x].notifyNextHopToO(f, 1, False)
    else:
        for (x, y) in vals:
            self.fields[x].connectX(self.fields[(x[0] + y[0], x[1] + y[1])], mirrored)
        for (x, y) in vals:
            f = self.fields[(x[0] + y[0], x[1] + y[1])]
            self.fields[x].notifyNextHopToO(f, 0, False)
            self.fields[x].notifyNextHopToO(f, 1, False)

def connectO(self, vals, mirrored=True, position=None):
    if position:
        con = [(x, y) for (x, y) in vals if x
               in self.fields[position].connectedX]
        for (x, y) in con:
            self.fields[x].connectO(self.fields[(x[0] + y[0], x[1] + y[1])], mirrored)
        for (x, y) in vals:
            f = self.fields[(x[0] + y[0], x[1] + y[1])]
            self.fields[x].notifyNextHopToX(f, 0, False)
            self.fields[x].notifyNextHopToX(f, 1, False)
    else:
        for (x, y) in vals:
            self.fields[x].connectO(self.fields[(x[0] + y[0], x[1] + y[1])], mirrored)
        for (x, y) in vals:
            f = self.fields[(x[0] + y[0], x[1] + y[1])]
            self.fields[x].notifyNextHopToX(f, 0, False)
            self.fields[x].notifyNextHopToX(f, 1, False)
```

Слика 20, 21 Функције за креирање и укидање веза међу пољима табле

Битно је напоменути да *connect* има и параметре *mirrored* и *position*, њима се контролишу гранични случаји када се два пешака супротних играча нађу један поред другог и онемогућава се обојици да се врате за по једно поље, већ из тог стања могу или међусобно да се прескоче, или да прате *Manhattan* образац кретања. Из овог разлога су се увели и низови *oneWayConnectedX* и *oneWayConnected*O, који бележе све суседе који су повезани на тренутно поље, али да тренутно поље нема везу ка њима, односно да је сада реч о оријентисаном путу.

У споменутим функцијама класе *Field* врши се и „креирање“ путање за дату фигуру у зависности од стања након одиграног потеза, што се може видети у имплементацији функција *notifyNextHopToX(...)*, *notifyNextHopToO(...)*, *annihilateNextHopToX(...)*, *annihilateNextHopToO(...)*, *getShortestPathToX(...)* и *getShortestPathToO(...)*. Ове функције представљају алгоритам чија се имплементација базира на познатом алгоритму под називом *Distance Vector*.

```
def annihilateNextHopToX(self, ind=0):
    if self.nextHopToX[ind][0] != self.position:
        self.nextHopToX[ind] = (None, 99999)
        for n in self.connectedO | self.oneWayConnectedO:
            if self.table.fields[n].nextHopToX[ind][0] == self.position:
                self.table.fields[n].annihilateNextHopToX(ind)

def annihilateNextHopToO(self, ind=0):
    if self.nextHopToO[ind][0] != self.position:
        self.nextHopToO[ind] = (None, 99999)
        for n in self.connectedX | self.oneWayConnectedX:
            if self.table.fields[n].nextHopToO[ind][0] == self.position:
                self.table.fields[n].annihilateNextHopToO(ind)

def notifyNextHopToX(self, f, ind=0, change=False):
    for n in self.connectedO:
        if self.table.fields[n].nextHopToX[ind][1] < (self.nextHopToX[ind][1]-1):
            self.nextHopToX[ind] = (
                self.table.fields[n].position, self.table.fields[n].nextHopToX[ind][1]+1)
            change = True
    if change:
        for n in self.connectedO | self.oneWayConnectedO:
            if self.table.fields[n] != f:
                self.table.fields[n].notifyNextHopToX(self, ind)

def notifyNextHopToO(self, f, ind=0, change=False):
    for n in self.connectedX:
        if self.table.fields[n].nextHopToO[ind][1] < (self.nextHopToO[ind][1]-1):
            self.nextHopToO[ind] = (
                self.table.fields[n].position, self.table.fields[n].nextHopToO[ind][1]+1)
            change = True
    if change:
        for n in self.connectedX | self.oneWayConnectedX:
            if self.table.fields[n] != f:
                self.table.fields[n].notifyNextHopToO(self, ind)
```

Слика 22 Функције којима је имплементирана модификована верзија Distance Vector алгоритма

Функцијама *annihilateNextHopToX(...)*, *annihilateNextHopToO(...)* се бришу сви суседи који тренутно поље имају за свој nextHop на одговарајућем путу, ово се пропагира све док се поља имају валидан показивач на заправо невалидно поље, то јест поље преко којег више не могу да дођу до краја, наравно, циљна поља за број скокова имају вредност 0 и за њих се nextHop никад не мења.

Што се тиче функција *notifyNextHopToX(...)*, *notifyNextHopToO(...)*, оне се позивају кад се деси форимирање нових веза између поља, за случај да је пут преко тих поља са мање скокова, такође, ове функције је неопходно позвати и након раскидања веза са суседима, како би алгоритам покушао да пронађе алтернативне руте.

```
def genNewStates(self):
    playerStates = list()
    if self.next.name == "X":
        for pos in self.next.getCurrectPositions():
            for n in self.table.fields[pos].connectedX:
                playerStates.append(self.table.getCopy())
                playerStates[-1].setGamePiece(pos, n, self.next.name)
    elif self.next.name == "O":
        for pos in self.next.getCurrectPositions():
            for n in self.table.fields[pos].connectedO:
                playerStates.append(self.table.getCopy())
                playerStates[-1].setGamePiece(pos, n, self.next.name)
    if self.next.noBlueWalls == self.next.noGreenWalls == 0:
        return playerStates
    newStates = list()
    if self.next.noBlueWalls > 0:
        for i in range(1, self.table.n):
            for j in range(1, self.table.m):
                for ps in playerStates:
                    if ps.isCorrectBlueWall((i, j)):
                        temp = ps.getCopy()
                        temp.setBlueWall((i, j))
                        if temp.canBothPlayersFinish():
                            newStates.append(temp)
    if self.next.noGreenWalls > 0:
        for i in range(1, self.table.n):
            for j in range(1, self.table.m):
                for ps in playerStates:
                    if ps.isCorrectGreenWall((i, j)):
                        temp = ps.getCopy()
                        temp.setGreenWall((i, j))
                        if temp.canBothPlayersFinish():
                            newStates.append(temp)
    return newStates
```

Слика 23 Функција за генерисање стања у која може да се дође на основу тренутног

Функција *genNewStates(...)* је искоришћена како би се на основу тренутног стања, односно *Table*, прво направила копија произвољан број пута, а онда на основу повезаних поља (на основу *Manhattan* шаблона) одиграли потези у оквиру копије. Уколико у тренутном стању играч нема могућност постављања зида, јер нема зидове, онда ће се вратити листа са овако генерисаним стањима, док уколико играч има барем један плави или зелени зид, за генерисана међустања ће се одиграти могуће постављање зида и уколико оба играча имају за сваког пешака пут до било ког иницијалног поља противника, овако изгенерисано стање ће бити прихваћено као валидно и биће додато у листу могућих стања. Унутар ове функције врши се и позив функције *canBothPlayersFinish(...)* која уколико је након одиграног потеза некој од фигура блокиран пут до неког од иницијалног поља противника враћа логичку вредност False и генерисање нових стања самим тим није могуће.

```
def canBothPlayersFinish(self):
    return self.canPlayerXFinish() and self.canPlayerOFinish()

def canPlayerXFinish(self):
    xPos1 = self.X.firstGP.position
    xPos2 = self.X.secondGP.position
    return None not in [self.fields[xPos1].nextHopToO[0][0], self.fields[xPos1].nextHopToO[1][0], self.fields[xPos2].nextHopToO[0][0], self.fields[xPos2].nextHopToO[1][0]]

def canPlayerOFinish(self):
    oPos1 = self.O.firstGP.position
    oPos2 = self.O.secondGP.position
    return None not in [self.fields[oPos1].nextHopToX[0][0], self.fields[oPos1].nextHopToX[1][0], self.fields[oPos2].nextHopToX[0][0], self.fields[oPos2].nextHopToX[1][0]]
```

Слика 24 Функције којима се врши провера да ли је неки од играча блокиран тако да не може да стигне до иницијалног поља противника

Функције чија је имплементација представљена на слици 25 користе се ради формирања путање помоћу *nextHop* низова које поседује свако поље табле.

```
def getShortestPathToX(self, ind=0):
    path = []
    nextHop = self
    while nextHop != None:
        path += [nextHop.position]
        if nextHop != self.table.fields[nextHop.nextHopToX[ind][0]]:
            nextHop = self.table.fields[nextHop.nextHopToX[ind][0]]
        else:
            nextHop = None
    return path

def getShortestPathToO(self, ind=0):
    path = []
    nextHop = self
    while nextHop != None:
        path += [nextHop.position]
        if nextHop != self.table.fields[nextHop.nextHopToO[ind][0]]:
            nextHop = self.table.fields[nextHop.nextHopToO[ind][0]]
        else:
            nextHop = None
    return path
```

Слика 25 Функције којима се реконструише путања до фигура

Функција која проверава унос задатог потеза је функција *parseMove(...)* класе *Game*, док функција која позива ову функцију и захтева унос новог потеза уколико претходни позив врати грешку је функција *play(...)* класе *Game*. Ова функција приказана је на слици 12 раније у документу. Промена тренутног стања се добија као последица позива функције *move(...)* над објектом класе *Table*, док се то ново стање на табли може видети у конзоли због позива функције *showTable(...)* над објектом типа *View*. Описани позиви функција могу се видети у оквиру функције *play(...)* на слици 12, док се имплементација функција *move(...)* класе *Table* и функције *showTable(...)* класе *View* могу видети редом на сликама 13 и 4. За проверу краја игре користи се функција *checkState(...)* класе *Table* (слика 11), која се такође позива у

функцији *play(...)*, а у оквиру које се позива функција *isWinner(...)* класе *Player*, која врши проверу да ли је играч стао на неку од две иницијалне позиције противника (слика 15).