“Ladders and snakes”

**Prezentare generala a jocului:**

Bazat pe jocul traditional “Scari si serpi”, jucatorii arunca pe rand zarurile si se misca cu pionii pe tabla. Daca aterizezi pe o scara,vei ajunge mai rapid pana la sosire, dar daca aterizezi pe un sarpe, vei aluneca inapoi pe traseu! Primul jucator care ajunge pe casuta cu numarul 100 castiga!

**Prezentarea regulilor jocului implementat:**

La fel ca in jocul traditional, in implementarea acestuia s-au aplicat aceleasi reguli. Cateva reguli care pot fi schimbate fata de jocul original sunt urmatoarele:

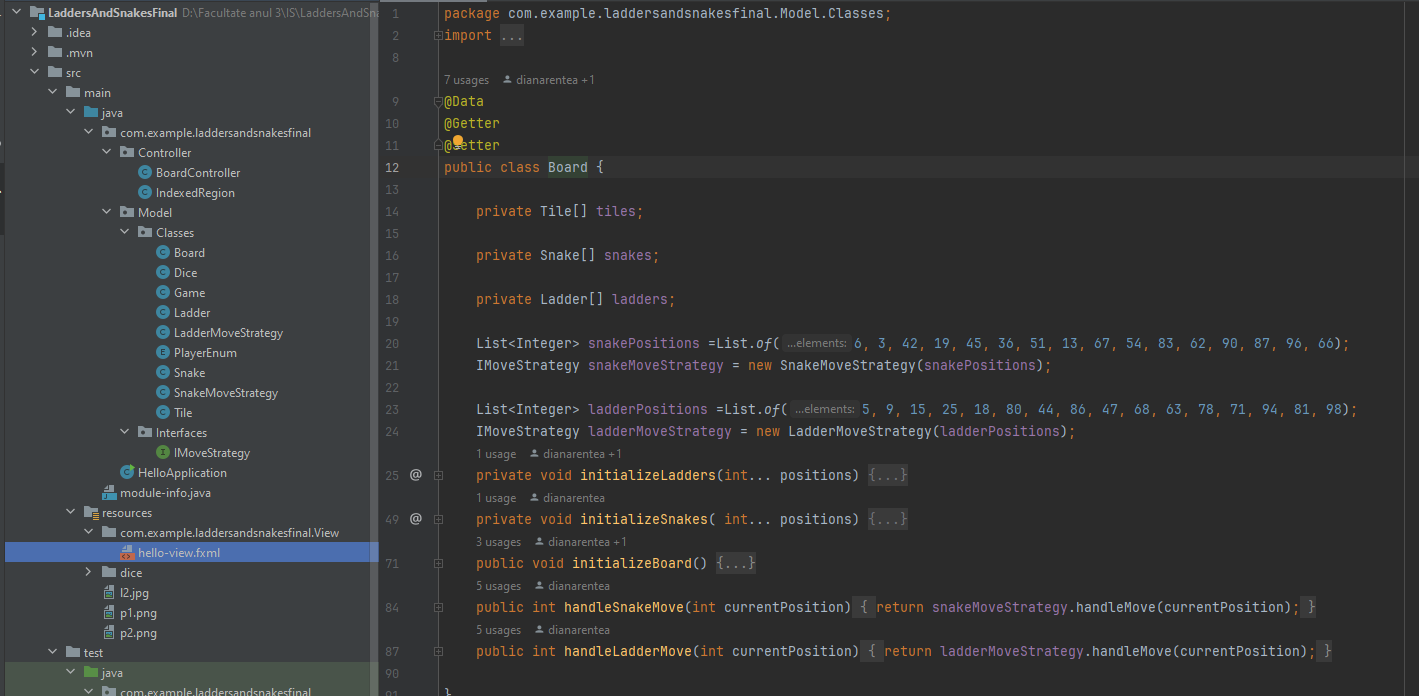
* Pozitia de start este pe prima casuta (nr. 1) si nu trebuie ca valoarea zarului sa fie sase pentru a incepe;
* Jocul este castigat doar daca pozitia curenta adunata cu valoarea zarului este egala cu o suta, altfel pionul nu se misca daca valoarea este mai mare ca o suta;
* Doi pioni se pot afla pe aceeasi casuta.

**Tehnologii, implementare, structura:**

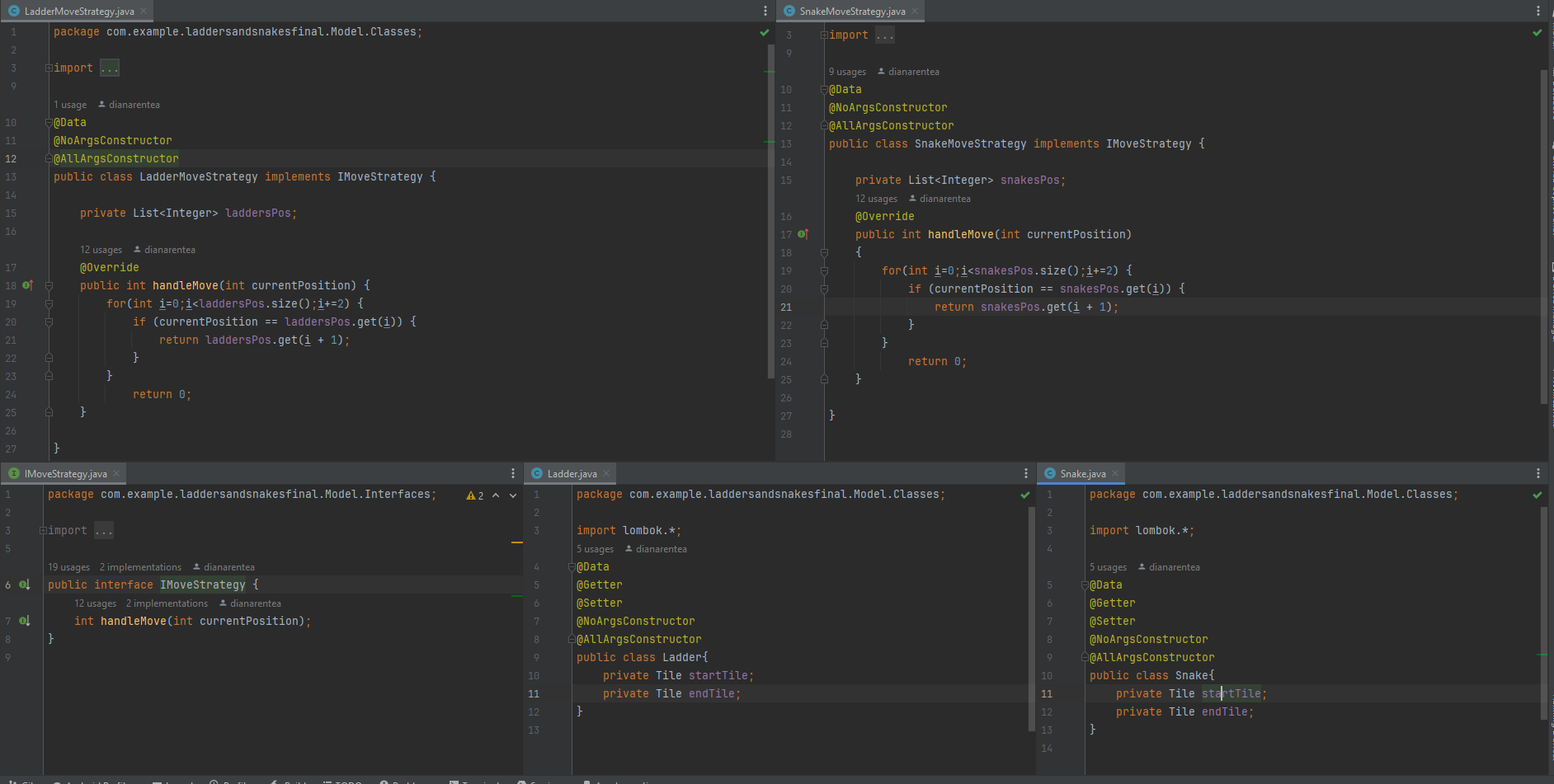
Pentru implementarea jocului s-a folosit framework-ul ***IntelliJ***, iar codul sursa este scris in ***JavaFX***. Jocul este creat local, pentru doi jucatori. Cand vine randul unui jucator acesta apasa click cu mouse-ul pe butonul de aruncare a zarurilor si apoi ofera mouse-ul celuilalt jucator cand ii trece randul.

Codul sursa este structurat pe baza ***design pattern-ului MVC*** -Model View Controller (poza de mai jos), avand separate partile de back-end (Model), front-end (View) si de notificari (Controller).

In partea de back-end exista clase cu functionalitati separate si bine structurate. Clasa cea mai importanta care ofera si principala functionalitate a jocului este *“Board”* unde au loc initializarile pozitiilor serpilor, scarilor, dar si a tablei in sine.



Pentru gestionarea mutarilor, atunci cand pionul ajunge pe o casuta cu sarpe/scara, s-a utilizat ***design pattern-ul “Move Strategy”***. Ideea acestui design pattern este de a ingloba algoritmi in clase separate, care implementeaza o interfata comuna. Structura este: contextul (clasa care contine obiectul), strategia (interfata) si strategia precisa (implementeaza interfata si implementeaza algoritmul). Avantajul acestui design-pattern este ca daca dorim sa schimbam/adaugam strategii, o putem face in clasele definite ca *“strategii precise” (LadderMoveStrategy/SnakeMoveStrategy)*, fara a afecta contextual actual.



In cazul de fata, strategiile de mutare sunt asemanatoare deoarece exista o casuta de plecare si alta de sosire atat pentru serpi, cat si pentru scari. Diferenta este intre vectorii de pozitii care sunt folositi pentru determinarea casutelor corespunzatoare fiecarei variante de mutare.

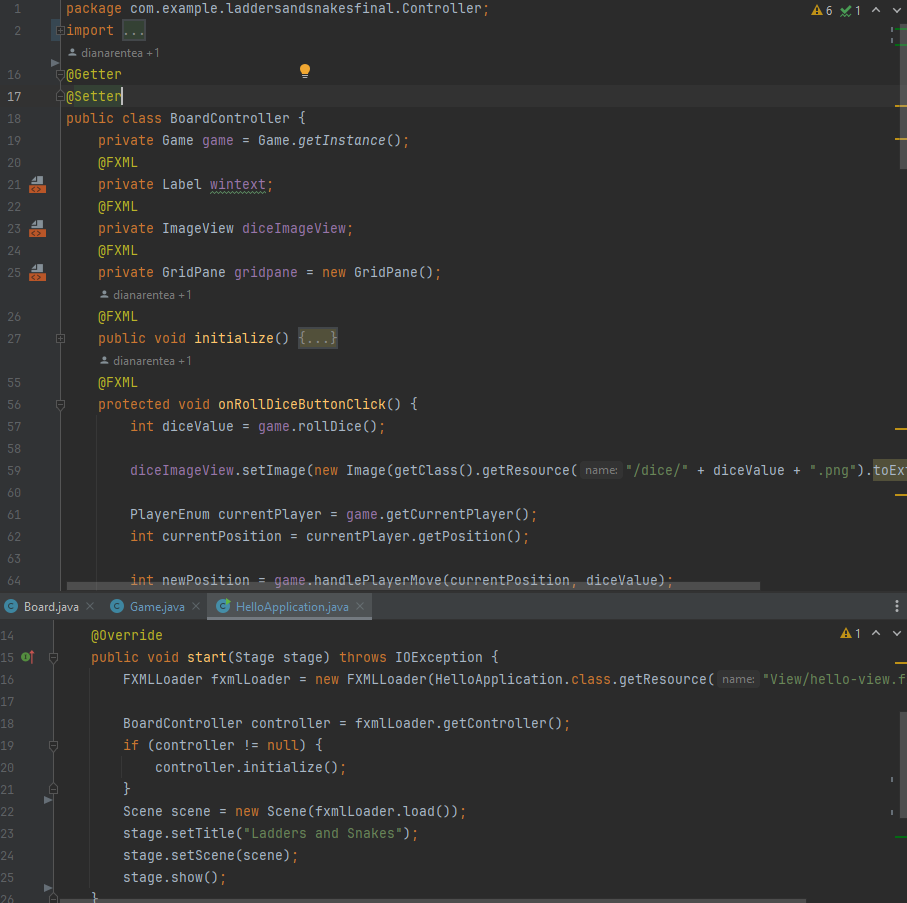
Vectorii de pozitii ai serpilor si scarilor sunt construiti astfel: numerele care indica pozitia de inceput si sfarsit ale scarilor, respective ale serpilor, sunt pozitionate consecutive in vectorii de pozitii, mai intai fiind pozitia de start si apoi pozitia finala pentru fiecare caz in parte (ex: 6 3 42 19 – in acest caz pozitia pentru casuta de start este 6, pozitia finala 3, apoi pozitia de start 42, pozitia finala 19 s.a. ). Cu ajutorul design pattern-ului utilizat mai sus se face aceasta determinare a pozitiilor, parcurgand in vectori doar pozitiie de start ca mai apoi sa se returneze pozitia finala pentru serpi, respectiv scari.

**Clasa “Game”**

Clasa “Game” contine toate metodele aplicate in dezvoltarea jocului: initializarea tablei, a jucatorilor, aruncarea zarului, mutarea pionului pe o casuta simpla sau pe casute cu serpi/scari, anuntarea castigatorului schimbarea turei jucatorilor.

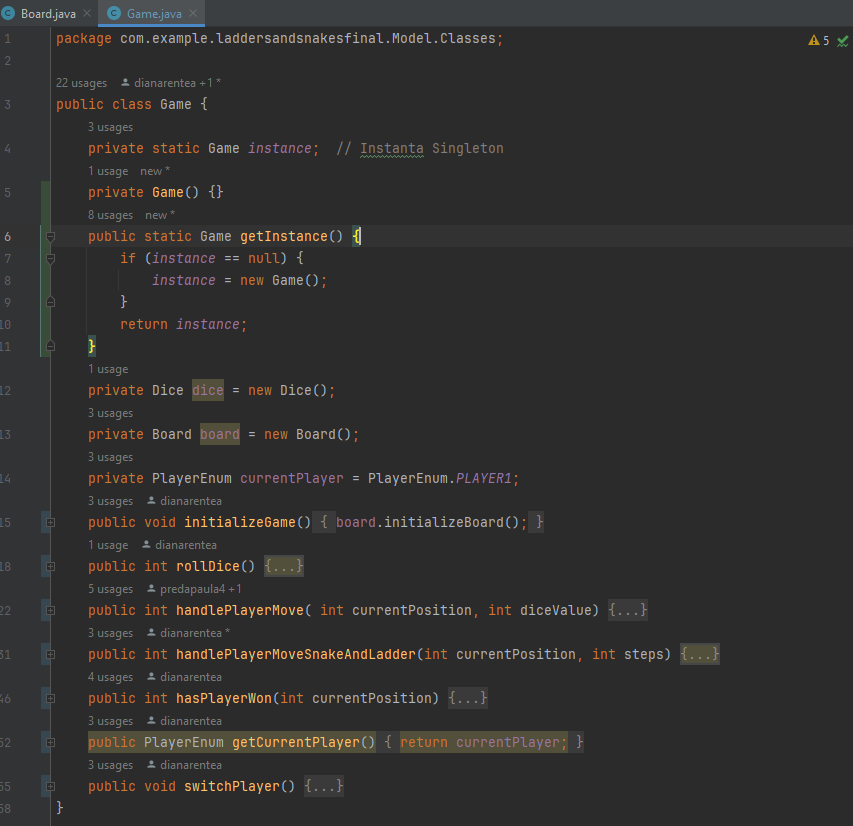
Pentru clasa “Game” s-a utilizat ***design pattern-ul “Singleton”.*** Scopul acestuia este de a restrictiona instantierea mai multor obiecte, respectiv in acest caz de a initializa un singur obiect de tipul “Game”, un singur joc.

Implementarea acestuia este simpla (vezi imaginea alaturata), metoda *getInstance()* este declarata statica deci poate fi apelata inainte de crearea obiectului. Prima data cand aceasta metoda este apleata creaza un nou obiect de tip “singleton” si apoi retunreaza acelasi obiect. Obicetul de tip “singleton” nu este creat pana cand nu avem nevoie de el si nu chemam metoda *getInstance()*.



Pentru initializarea tablei s-a utilizat un grid ce reprezinta o matrice de dimensiune 10x10, dar care in back-end devine un vector de o suta de pozitii.

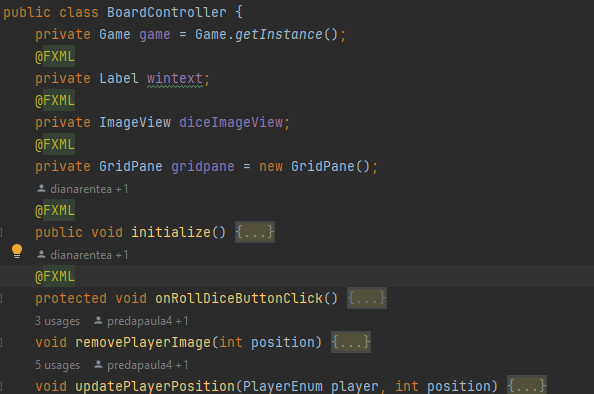
Metodele pentru actualizarea front-end-ului sunt: de a elimina imaginea care reprezinta pionul jucatorului si de a o afisa pe noua pozitie a acestuia.



In imaginea alaturata se poate observa ca in clasa *BoardController* este apelata metoda *getInstance()* din clasa *Game*, iar in metoda de initializare a UI-ului se observa ca lla randul ei *BoardController* este instantiata.

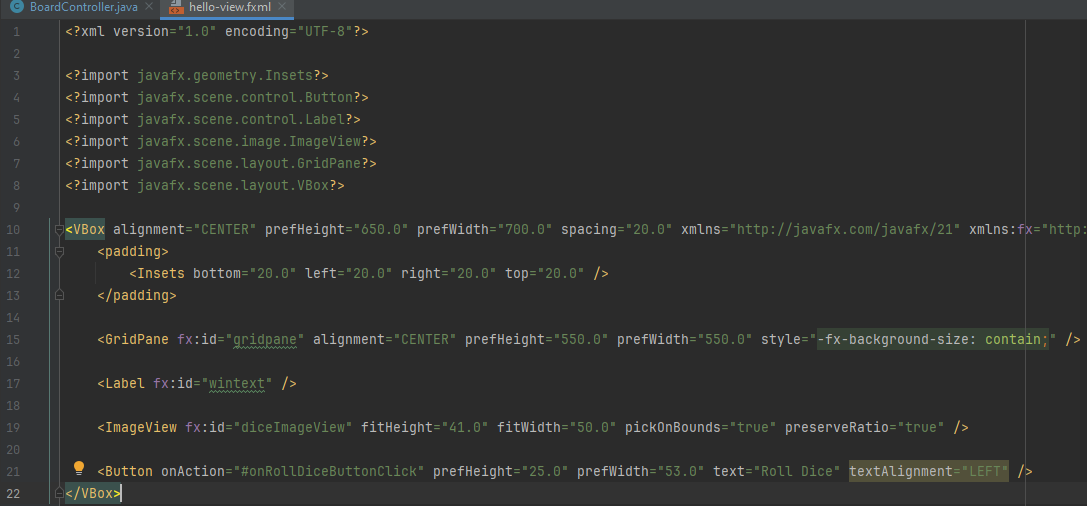
Astfel atunci cand porneste programul va exista o singura instanta a clasei *Game*.

Tot in controller are loc legatura dintre back-end si front-end, unde totodata sunt si metodele pentru initializarea si actualizarea front-end-ului.



**FRONT-END**

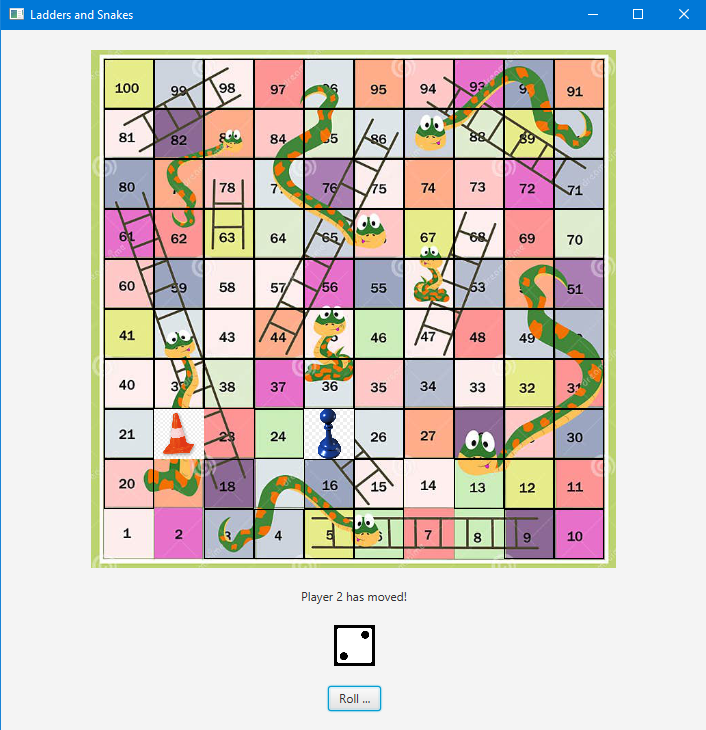
Proeictul creat cu ***JavaFx***  isi are principala atribuire in crearea interfetei jocului. In partea de *View* exista un fisier .fxml unde se initializeaza si are loc jocul in sine. Elementele din controller sunt apelate prin numele lor cu ajutorul adnotarii *@FXML*. Pentru a face legatura dintre campurile din *Controller* si cele din front-end, in *View* elementul de UI are o porprietate “id” careia i se atribuie numele campului din controller.



In final, jocul in sine are urmatoarea configuratie:

* tabla de joc
* text pentru a afisa care jucator a mutat
* valoarea zarului
* buton pentru a da cu zarul

Se observa si imaginile cu cei doi pioni (portocaliu si albastru) pentru fiecare dintre cei doi jucatori.



**TESTING**

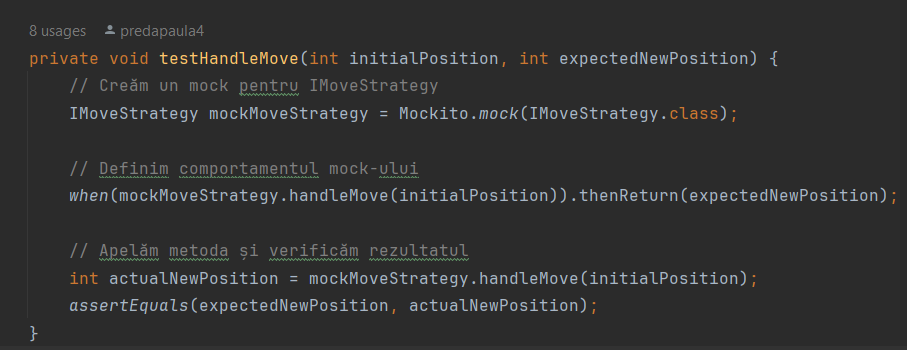
***JUnit si Teste Unitare***

Pentru asigurarea calitatii si corectitudinii implementarii, am adoptat abordarea de dezvoltare bazata pe teste unitare utilizand framework-ul JUnit. Acesta ne permite sa definim si sa executam teste pentru fiecare componenta a codului, asigurand astfel ca functionalitatile individuale indeplinesc cerintele specificatiilor.



***Mockito si Mocking***

In cadrul testelor noastre, am integrat Mockito pentru a efectua simularea (mocking) obiectelor si comportamentelor. Mocking-ul ne ofera capacitatea de a crea obiecte simulatoare care pot inlocui componentele reale, permitandu-ne astfel sa izolam unitar fiecare componenta pentru teste independente si repetabile. Aceasta faciliteaza validarea comportamentelor si interactiunilor intr-un mod controlat si predictibil.



Clasele de test care se afla in proiectul nostru sunt:

1. *SnakeMoveStrategyTest*: Aceasta clasa de teste verifica functionalitatea clasei

SnakeMoveStrategy, care este responsabila pentru gestionarea mutarilor jucatorilor pe tabla atunci cand

acestia aterizeaza pe o caseta cu sarpe. Avem implementate functii de verificare pentru fiecare sarpe (cap

si coada).

1. *PlayerEnumTest*: Aceasta clasa de teste verifica metodele din clasa PlayerEnum, care definesc

enumerarile pentru jucatori. Aici verificam daca se atribuie corect numele si imaginea fiecarui jucator.

1. *LadderMoveStrategyTest*: Aceasta clasa de teste verifica functionalitatea clasei

LadderMoveStrategy, responsabila pentru gestionarea mutarilor jucatorilor pe tabla atunci cand acestia

aterizeaza pe o caseta cu scara. Avem implementate functii de verificare pentru fiecare scara (punct de

plecare si punct de oprire).

1. *GameTest*: Aceasta clasa de teste verifica metodele din clasa Game, care controleaza logica

jocului. Avem metode implementate pentru a verifica daca jocul este initializat corect, iar jucatorul incepe

de la pozitia 1, verificam comportamentul atunci cand un jucator castiga, verificam mutarile jucatorilor si

verificam daca metoda "hasPlayerWon" determina corect castigatorul.

1. *DiceTest*: Aceasta clasa de teste verifica metodele din clasa Dice, responsabila pentru

simularea aruncarii unui zar.

1. *BoardTest*: Aceasta clasa de teste verifica metodele din clasa Board, care gestioneaza logica

tablei cu serpi si scari. Verifica corectitudinea mutarii unui jucator atunci cand aterizeaza pe un sarpe si

verifica corectitudinea mutarii unui jucator atunci cand aterizeaza pe o scara.

1. *IndexedRegionTest*: Aceasta clasa de teste verifica metodele din clasa IndexedRegion, care

reprezinta o regiune indexata in interfata utilizator.

1. *BoardControllerTest*: Aceasta clasa de teste verifica metodele din clasa BoardController, care

controleaza interactiunea intre model si interfata utilizator. Verificam daca controller-ul si elementele sale

asociate sunt initializate corect, verificam eliminarea corecta a imaginii unui jucator de pe tabla, verificam

actualizarea corecta a pozitiei unui jucator pe tabla, verificam scenariul in care un jucator castiga si mesajul

asociat.