GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Lab 6 : Test-Driven development

27 april 2018
Cedric De Boom
(e-mail: {voornaam.familienaam}@ugent.be)

1 Inleiding

In deze labsessie wordt de spellogica van het Othellospel geïmplementeerd. Hiervoor wordt gebruik gemaakt van het test-driven paradigma, waarbij eerst de testcode wordt geschreven, en nadien pas de eigenlijke implementatie.

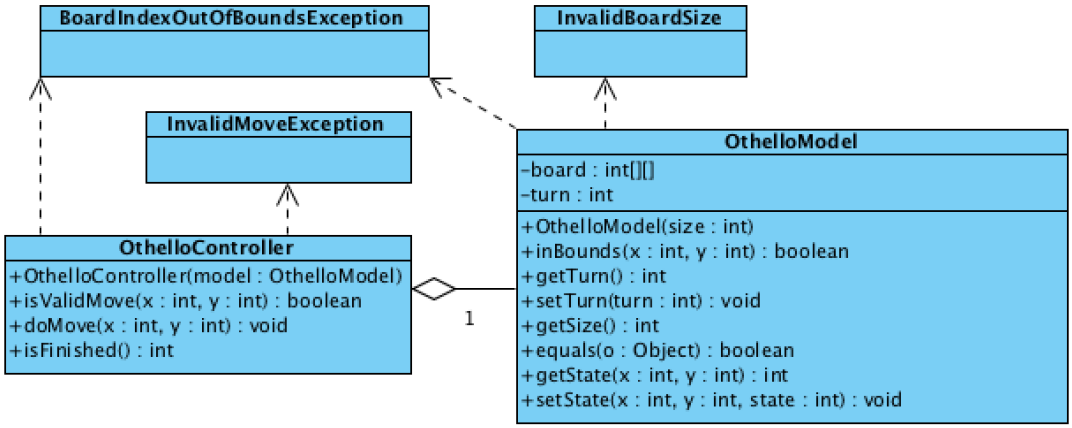
Er wordt gebruik gemaakt van de GUI-klassen `OthelloPiece`, `Counter` en `OthelloPanel` uit vorige lab sessie. De controller, die de eigenlijke spellogica bevat, wordt in deze zesde labsessie geschreven en getest, alsook het spelmodel zelf dat de speltoestand (onafhankelijk van zijn grafische voorstelling) bevat.

2 JUnit

In deze labsessie maken we gebruik van het JUnit-testraamwerk, dat in hoofdstuk 5 van de syllabus in detail uitgelegd wordt. We gebruiken hierbij **annotaties** om testcases te definiëren, waarbij elke testcase typisch een oproep naar een `assert()`-methode bevat.

Opgave 1 - De testklassen

De toestand van het spel wordt voorgesteld door een object van de klasse `OthelloModel`. De spellogica wordt geïmplementeerd in een object van de klasse `OthelloController`. Het UML-diagram van deze klassen, zoals het door het ontwikkelteam toegeleverd werd, vind je in Figuur 1. Ook krijg je een dummy-versie van deze klassen.

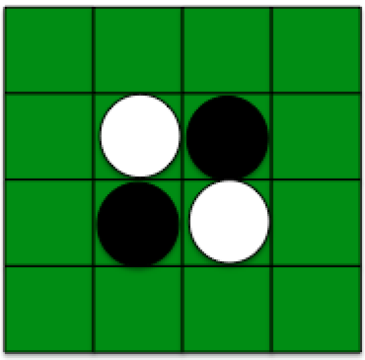


Figuur 1: UML-diagram van het spelmodel en de spellogica.

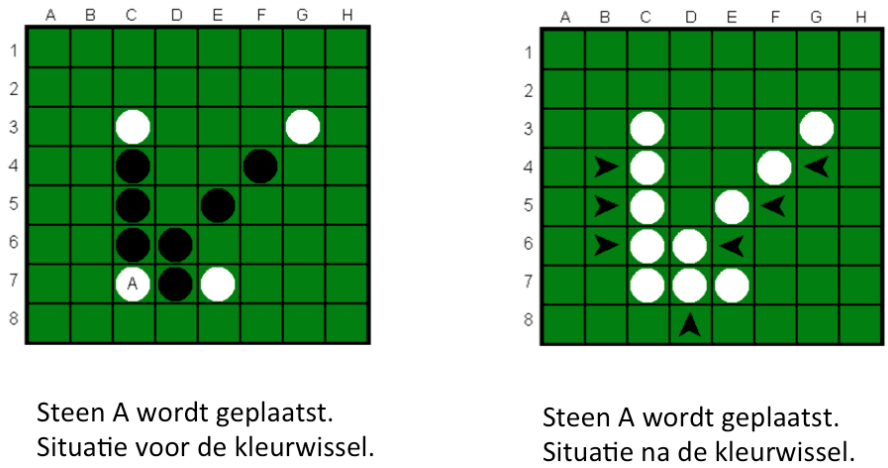
Genereer JUnit test klasses voor de klassen `OthelloController` en `OthelloModel`. In Eclipse kies je hierbij voor *New* > *JUnit Test Case*. Gebruik JUnit versie 4.

3 Othello spelregels

De bedoeling van het spel Othello (of Reversi) bestaat erin zoveel mogelijk stenen van de eigen kleur te hebben op het eind van het spel. De startsituatie wordt in Figuur 2 weergegeven voor een 4×4 speelbord. Spelers leggen om beurt een steen van de eigen kleur op het bord, waarbij de speler met kleur zwart start. Een dergelijke zet is enkel geldig, indien hierdoor één of meer stenen van de tegenspeler ingesloten worden. Dit insluiten mag horizontaal, verticaal of diagonaal gebeuren. Indien een speler geen geldige zet kan doen, verliest hij zijn beurt. Kunnen beide spelers geen geldige zet doen, dan eindigt het spel.

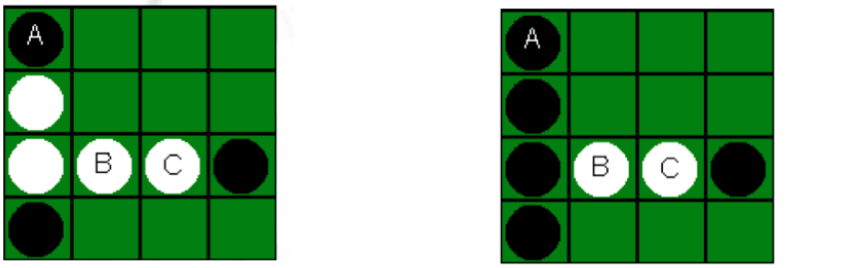


Figuur 2: Spelregels: startsituatie.



Figuur 3: Insluiten van stenen: principe.

Het insluiten heeft als gevolg dan de stenen van kleur wisselen. Hierbij wisselen enkel de stenen van kleur die rechtstreeks door het plaatsen van de nieuwe steen ingesloten worden (Figuur 3). Je moet dus GEEN rekening houden met de gevolgen van het wisselen van kleur (waardoor in principe weer andere reeksen zouden ingesloten worden - zie Figuur 4).



Steen A wordt geplaatst.
Situatie voor de kleurwissel.

Steen A wordt geplaatst.
Situatie na de kleurwissel.
Stenen B en C wisselen NIET.

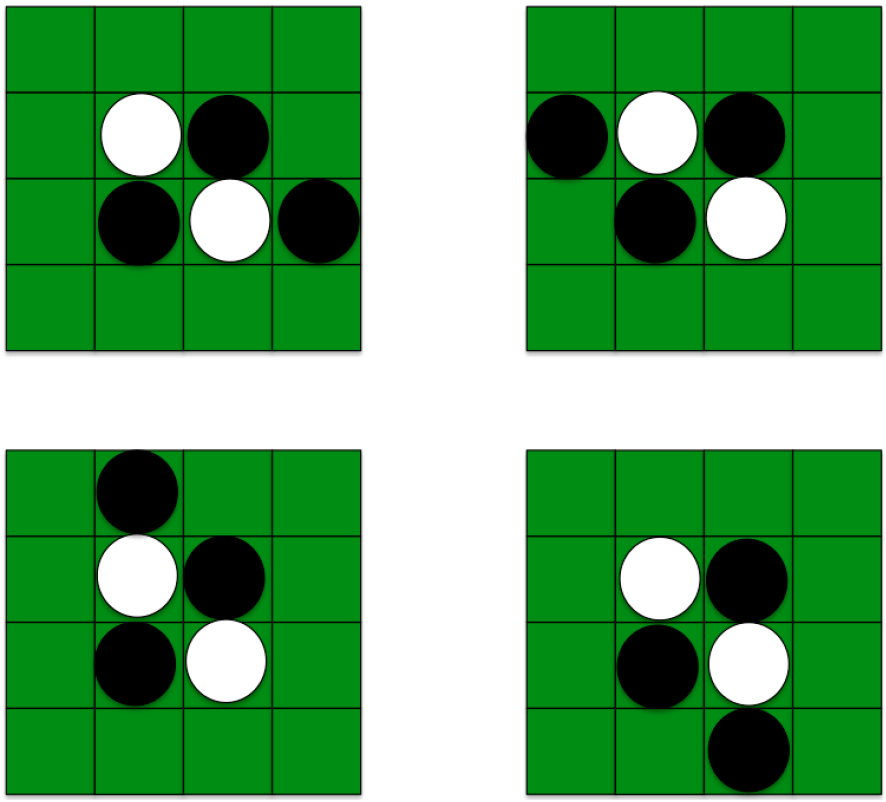
Figuur 4: Insluiten van stenen: enkel het onmiddellijk gevolg van de zet telt.

Opgave 2 - Test implementatie

Het spelmodel

Zoals het UML-diagram uit Figuur 1 aangeeft, beschikt deze klasse over de volgende te testen methoden:

- contstructor en `getSize()`
  - nakijken dat na constructie, het object verschilt van het `null`-object
  - nakijken dat het constructor-argument correct behandeld wordt, namelijk er wordt een uitzondering gegeneerd indien het argument niet strikt positief is of indien het argument oneven is. Indien het argument een geldige waarde heeft, moet nagekeken worden of de bordgrootte correct geïnitialiseerd werd.
- `inBounds()`: deze methode kijkt na of een (x, y)-coördinaat binnen het spelbord valt. Bouw testcases via de analyse van randwaarden (je hebt in principe zeker 49 testcases nodig, per coördinaat heb je immers 7 waarden te testen (namelijk ongeldig en veel te klein, -1, 0, geldige waarde, lengte-1,lengte, veel te groot)). Probeer de hoeveelheid code te beperken!
- `setState()` en `getState()`
  - ongeldige coördinaten leveren een uitzondering. Hier kan 1 test volstaan (vermits `inBounds()` reeds getest werd in het vorige puntje)
  - bij doorgeven van geldige coördinaten, kijk je na of `getState()` de correcte toestand in het model oplevert.
- `setTurn()` en `getTurn()`: kijk na of `getTurn()` de verwachte toestand oplevert na het oproepen van `setTurn()`. Hier hoef je niet te testen of het argument geldig is (wegens de beperkte tijd beschikbaar in de labsessie).
- `equals()`: deze methode kijkt na of twee spelmodellen logisch gelijk zijn. Hierbij is het resultaat `false` indien één van onderstaande voorwaarden geldt:
  - de methode werd opgeroepen met een `null`-argument
  - de methode werd opgeroepen met een object van een fout type
  - de afmetingen van de spelborden stemmen niet overeen
  - de toestand van minstens één speelveld verschilt in beide modellen
  - de beurt-variabele verschilt



Figuur 5: Openingszetten van zwart als testcases voor `doMove()`.

De spellogica

Voor de spellogica, wensen we volgende testen uit te voeren (zie UML-diagram uit Figuur 1):

- `isValidMove()`: test voor een 4×4 speelbord of het leggen van een steen vanuit de startpositie al dan niet geldig is. Test daarenboven het leggen van een steen op een ongeldige coördinaat.
- `doMove()`: test of het plaatsen van een steen op een een 4×4 speelbord tot de juiste nieuwe toestand leidt. Deze nieuwe toestand houdt ook in dat de beurt na uitvoering van de zet gewisseld is. Hierbij beperk je je tot het testen van de geldige openingszetten van zwart, zoals weergegeven in Figuur 5.
- `isFinished()`: deze methode kijkt na of het spel afgelopen is. Ze geeft volgende return-waarden:
  - `0`: spel nog niet afgelopen
  - `-1`: spel afgelopen en zwart is gewonnen
  - `1`: spel afgelopen en wit is gewonnen
  - `2`: spel afgelopen en gelijkspel

  Maak testcases voor elk van deze vier situaties. Voor de laatste 3 gevallen, test je afzonderlijk de gevallen:
  - alle speelvelden zijn bezet
  - geen van beide spelers kan nog een geldige zet uitvoeren (Tip: probeer een triviale opstelling van het bord te vinden waar geen geldige zet meer mogelijk is - bijvoorbeeld met slechts 1 onbezet veld)

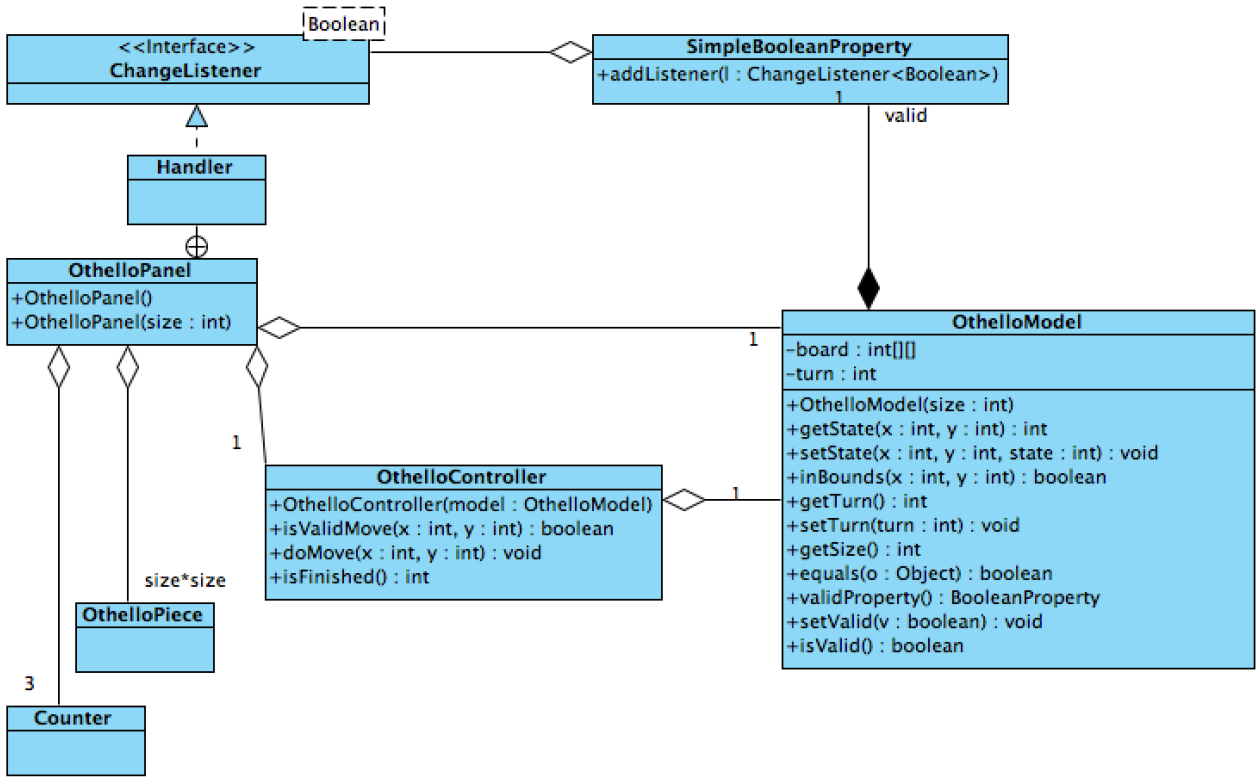
Opgave 3 - Implementatie van Othello

Implementeer de hierboven beschreven voor de klassen `OthelloModel` en `OthelloController` methoden volgens de Othellospelregels. Voer nadien de beschreven testcases uit.

4 Othello

Opgave 4

Koppel nu de spellogica aan de GUI klassen uit lab sessie 5, en dit volgens het UML-diagram uit Figuur 6. Gebruik hierbij opnieuw het Java FX Beans model, en het bijhorende eventmodel, zoals uitgelegd in labsessie 5. We gebruiken een `BooleanProperty` om aan te duiden wanneer de toestand van het model veranderd is (door een actie in de controller). Het `OthelloPanel` kan zich dan als luisteraar registeren bij deze Property zodat de nieuwe toestand van het model steeds in de GUI gereflecteerd wordt.



Figuur 6: UML-diagram van de grafische toepassing.

Maak een Java FX-applicatie `MainOthello` waarop je een `OthelloPanel` instantie zet om het spel te testen. In de klasse `OthelloPanel` zorg je dat het volgende gedrag gerealiseerd wordt:

- indien de muis over een speelveld gaat waarbij dit speelveld een geldige zet is voor de speler die aan de beurt is, wordt op dit speelveld een steen in de kleur van die speler getekend.
- indien geklikt wordt op een geldig speelveld, wordt die zet effectief uitgevoerd.
- indien geklikt wordt buiten het speelveld, of op een ongeldig speelveld, wordt dit genegeerd.

Wanneer geen geldige zetten meer mogelijk zijn, verschijnt een venster die de winnaar aanwijst.

5 Tot slot

Upload je bestanden naar Indianio. We verwachten volgende bestanden :

- package `game`
  - `GameInterface.java`
- package `othello`
  - `OthelloModel.java`
  - `OthelloController.java`

  - `Counter.java`
  - `OthelloPanel.java`
  - `OthelloPiece.java`
  - `MainOthello.java`
- package `othello.exception`
  - `BoardIndexOutOfBoundsException.java`
  - `InvalidBoardSizeException.java`
  - `InvalidMoveException.java`
- package `test`
  - `OthelloControllerTest.java`
  - `OthelloModelTest.java`