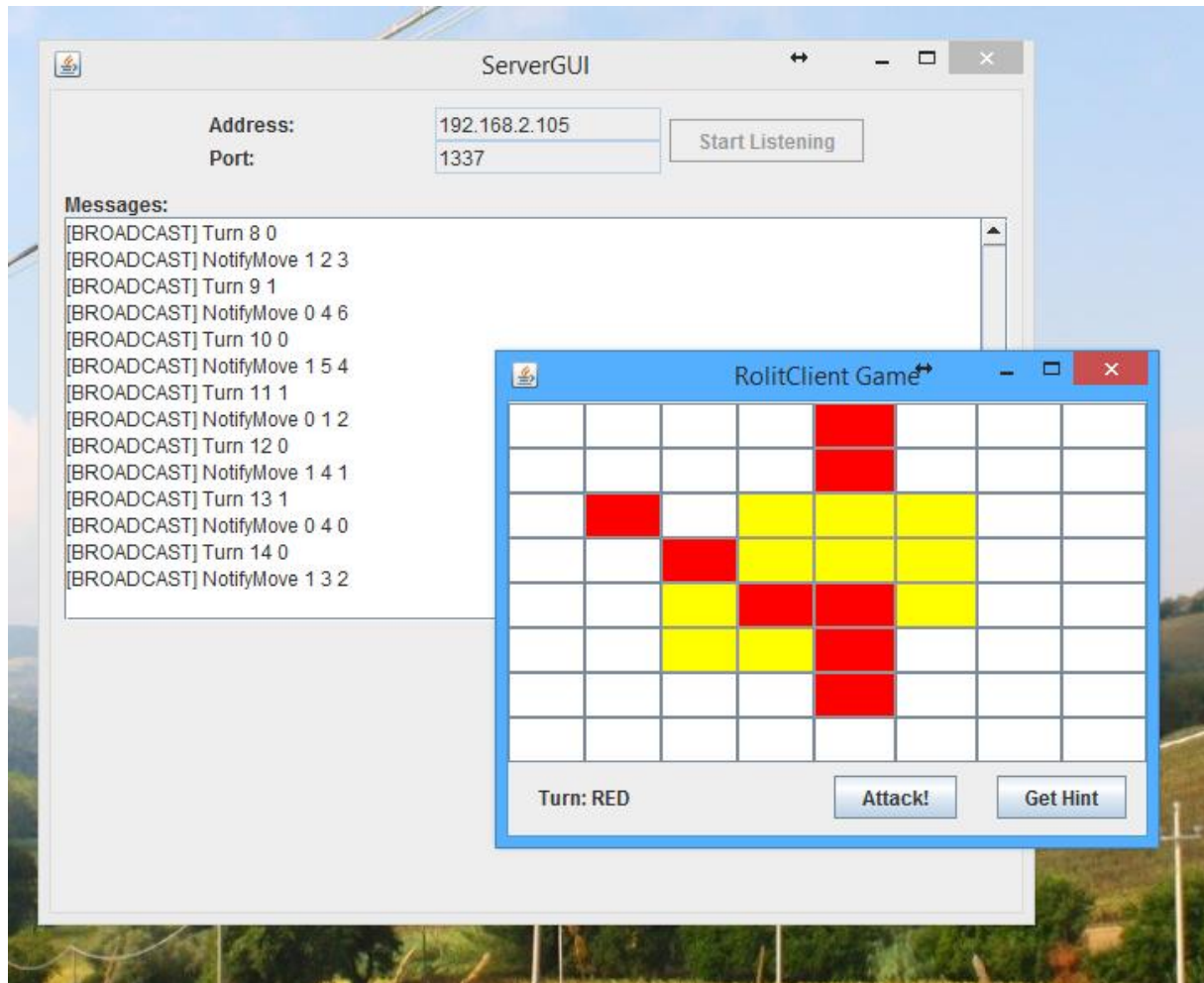


Rolit verslag en documentatie



- VICTOR LAP EN YURI VAN MIDDEN

30 JANUARI 2014

Inleiding

We hebben RolIt in Java geprogrammeerd voor module 2. RolIt is een strategisch spel waarbij geprobeerd moet worden om zoveel mogelijk vakjes naar jouw kleur te zetten. We hebben een werkend resultaat en een (met behulp van de server) speelbaar spel. We hebben veel van GitHub gebruik gemaakt om ons project op te bewaren en samen te kunnen werken aan de code van het programma. De link naar de repository is <https://github.com/victorlap/ss-rolit>.

Inhoud

Inleiding.....	2
Verklaring gebruikte lettertypen en stijlen	3
Instructies om het programma uit te voeren.....	4
Algemeen ontwerp van het programma	4
Algemene afwegingen	4
Bedachte strategie om spelregels na te leven	4
Zelf gedefinieerde klassen	5
Board	5
Game.....	6
Color	6
Player.....	6
Strategy	6
ComputerPlayer.....	6
SmartComputerPlayer	6
Eisen en de bijhorende klassen.....	6
Implementatie van het Model-View-Controllerpatroon	7
Betrokken klassen	7
Klassendiagram van klassen van het Model-gedeelte	8
Functionele eisen van het serverprogramma	8
Betrokken klassen en de implementatie	8
Rollen van de server	8
Werking van de server.....	9
Testrapporten	9
Opgestelde tests.....	9
Board	9
Player.....	11
Interactie met behulp van het protocol	12
Dataformats.....	13
Planning	13

Wijzigingen in het plan	13
Reflectie	13
Bijlagen.....	14
1 - Raster van het speelbord als <code>DIM == 8</code>	14
2 - Wiskunde achter de vakjes-checks.	14

Verklaring gebruikte lettertypen en stijlen

Tekst die in dit lettertype is geschreven stellen Java-code, methoden, klassen, berekeningen, commando's of andere programmeertekst voor.

Dit is normale tekst.

Instructies om het programma uit te voeren

Er is een server-kant van het programma en een client-kant van het programma. Er moet in onze implementatie op dit moment altijd een server draaien om met de client een spel te kunnen starten.

In het archief staan twee Java-executables: `RolitServer.jar` en `RolitClient.jar`.

1. Start eerst de *RolitServer.jar* executable, en klik op *Start Listening*.
2. Start vervolgens minimaal twee clients door *RolitClient.jar* op te starten.
3. Kies een unieke spelersnaam en klik op *Connect*. (Als spelersnaam kan je ook AI of AI2 ingeven om een computerspeler in te zetten)
4. Kies uit het lijstje een kleur waarmee je wil spelen en klik op *Choose color*.
5. Als alles akkoord is kan je *Ready* klikken.

Zodra alle clients op Ready hebben geklikt start het spel.

Algemeen ontwerp van het programma

Algemene afwegingen

We hebben ervoor gekozen om alle klassen, JavaDOC en methodenamen in het Engels te noteren. Dit zorgt voor een uitstekende consistentie door het hele programma heen. Het protocol is ook in het Engels gedefinieerd, wat hier ook aan bijdraagt.

Verder bouwen we het bord om de statische variabele DIM heen. Die is standaard 8, maar kan ook aangepast worden, mocht dat ooit nodig zijn. Zo kunnen we ook RolIt spelen op een bord van 20 x 20 vakjes.

Ook hebben we ervoor gekozen om de GUI in de eerste instantie simpel te houden, zodat we meer aandacht kunnen besteden aan de implementatie van de eisen. Mocht er tijd over zijn, dan kunnen we alsnog beslissen om de GUI uit te breiden, of mooier te maken. Het is belangrijker dat het systeem functioneert.

Bedachte strategie om spelregels na te leven

`Board.java` is de klasse die bij ons het meest werk doet in het handhaven van de spelregels (zie ook Zelf gedefinieerde klassen). De strategie die we gebruiken om de spelregels na te leven is als volgt.

We hebben acht methoden gedefinieerd die vanaf een gegeven vakje, recursief, in alle windrichtingen kijkt voor een bepaalde kleur of daar een vakje ligt van dezelfde kleur, totdat de methode op een gegeven moment een vakje wil testen dat buiten het bereik van het geteste bord ligt. De methoden heten `checkNorth`, `checkNorthEast`, `checkEast`, `checkSouthEast`, ... en geven het vakje terug waar de geteste kleur in zit. Is er niet zo'n geval, dan krijgen we altijd -1 terug.

Voor een bepaalde zet die we zouden willen doen kijken we:

1. Of het bord niet vol is.
2. Of er voor EEN kleur (dus alle kleuren worden getest) een int door `checkNorth`, etc. wordt teruggegeven die wiskundig gezien naast het geteste vakje ligt. Als we bijvoorbeeld voor vakje met $i = 27$ willen kijken of er direct een vakje boven ligt, moet

in het vakje met $i = 19$ één van de kleuren liggen (zie ook bijlage 1). Dit doen we voor alle windrichtingen. Zodra één van de tests slaagt, zeggen we `isBordering = true` voor dat vakje.

3. Dan gaan we voor alle vakjes kijken of ze een geldige zet zijn (met het principe van hierboven). Zodra er een vakje wordt gevonden dat een geldige zet is (niet volgens de spelregels, maar volgens het principe hierboven), gaan we uitrekenen OF de zet een invloed heeft op andere vakjes. Daarbij moet minimaal één van de windrichtingentests aan de door ons berekende voorwaarden voldoen, bijvoorbeeld in het geval van `checkNorth` vanaf vakje $i = 27$ moet het gevonden vakje $i \leq 11$ zijn om te weten dat die zet invloed heeft. Als er wordt opgeleverd dat er een zet is die invloed heeft, MOET die zet gedaan worden, anders is `isBordering` voldoende.
4. Als extraatje kunnen we ook uitrekenen HOE GROOT de invloed is van een zet. Door bij elke windrichting uit te rekenen wat het verschil is tussen het geteste vakje en het gevonden vakje, kan je berekenen hoe groot de invloed precies is van je zet. Als, zoals bij het vorige voorbeeld, het geteste vakje $i = 27$ is en `checkNorth i = 11` oplevert, kan je in Java deze twee integers door elkaar delen, waarbij je 2 overhoudt. Dat klopt, want deze zet levert op dat er twee ‘balletjes’ draaien naar de eigen kleur. Deze test doen we uiteraard voor alle windrichtingen en de uitkomsten worden bij elkaar opgeteld, wat de kwaliteit van het veld oplevert.
5. Voor de beschrijving van de gebruikte wiskunde, kijk bijlage 2.

Zelf gedefinieerde klassen

In onderstaande kopjes worden van alle zelf gedefinieerde klassen beschreven wat de rollen zijn, wat de verantwoordelijkheden zijn en welke eisen ze vervullen. Ook wordt het uiteindelijke doel van de klasse beschreven. Voor gedetailleerdere over methoden, fields en constructors verwijzen we door naar de JavaDOC van het project.

Board

Board is mogelijk de belangrijkste klasse in onze versie van RollIt. Allereerst initialiseert Board een virtueel speelbord, met een array van $DIM * DIM$ vakjes. DIM is in dit geval de variabele van de grootte van het bord dat we willen realiseren. We hebben het hele spel zó gemaakt dat we DIM naar wens kunnen aanpassen. DIM is standaard 8.

Board kan allerlei bewerkingen en berekeningen uitvoeren op het bord dat gemaakt wordt in de constructor. Board kan bijvoorbeeld testen of het bord helemaal vol is (dat wil zeggen dat het spel is afgelopen), of een veld leeg is, maar Board kan ook testen welke Color de meeste velden heeft.

Naast dit soort standaardfunctionaliteit kan Board ook de tests uitvoeren die nodig zijn om te bepalen of een zet die een speler wil doen, geldig is. Board test bijvoorbeeld of een vakje grenst aan een reeds gevuld vakje, maar ook of er een zet is die beter is dan de zet die je wil doen. Zo controleert Board of er aan de spelregels wordt gehouden.

Ten slotte heeft Board een methode om de kwaliteit van alle mogelijke zetten te bepalen. Deze methode rekent voor alle zetten uit hoeveel “balletjes” er worden gedraaid bij het doen van een zet. De `SmartStrategy` en de `getHint` maken gebruik van deze methode om de beste eerstvolgende zet te bepalen.

Het doel van deze klasse is om alle benodigde bewerkingen die we willen doen op het bord, te realiseren.

Game

In Game wordt onder andere het bord geïnitieerd en worden de spelers onderhouden voor het spel. Game heeft op zich niet heel veel functionaliteit, maar start wel het spel en zorgt ervoor dat een Game-object een Board heeft en een array van Players. Zo kunnen we meerdere Game-instanties maken met elk haar eigen Board en eigen Players.

Color

Color is een enum klasse die alle kleuren definieert. Deze klasse heeft als doel om door het lijstje van kleuren heen te kunnen scrollen en een kleur te vertalen naar een int en omgekeerd (eisen voor interactie met het protocol). Ook kan met deze klasse de volgende kleur bepaald worden die aan zet is.

Player

Dit is de abstracte klasse die Player-objecten kan maken. Elke Player heeft een Strategy, een Name en een Color. Player heeft verder methodes om deze eigenschappen uit te lezen en om zetten te doen op het bord. Er zijn constructors om een computerspeler te maken en constructors om een menselijke speler te maken. Dit is afhankelijk van of er een strategie wordt meegegeven.

Strategy

Strategy is de interface waarmee de strategieën van de computerspelers worden geïmplementeerd. Bij strategy zijn de methode die de naam teruggeven en de methode die de zet teruggeeft belangrijk. Zie ook onderstaande implementaties van de klassen ComputerPlayer en SmartComputerPlayer.

ComputerPlayer

Deze klasse maakt voor haar functionaliteit gebruik van het Board en van de methoden die in Board zijn geïmplementeerd. Board heeft een methode om te bekijken welke zetten geldig zijn, en RandomStrategy kiest een willekeurige zet uit de lijst met geldige zetten.

Het doel van deze klasse is om een strategie te maken voor een ComputerPlayer die niet moeilijk is om te verslaan.

SmartComputerPlayer

Deze klasse maakt voor haar functionaliteit ook gebruik van de klasse Board. Board heeft namelijk een methode die van alle geldige zetten de kwaliteit berekent. De kwaliteit van een zet wordt bepaald door het aantal 'balletjes' dat een zet laat rollen. Dus hoe meer vakjes beïnvloed worden door een zet, hoe hoger de kwaliteit. SmartStrategy zoekt een zet uit met de hoogste kwaliteit en zal deze zet doen.

Doel van deze klasse is om te zorgen dat de computer altijd een zet zal doen die veel invloed heeft op andere vakjes, hoewel dat niks zegt over de zetten daarna.

Eisen en de bijhorende klassen

Implementatie van het Model-View-Controllerpatroon

Voor een aantal elementen in het programma hebben we het Model-View-Controllerpatroon gebruikt. Zo hebben we voor de authenticatie bij de authenticatieserver dit patroon toegepast, maar ook bij het onderhouden van de grafische gebruikersinterface gebruiken we het.

Betrokken klassen

De betrokken klassen bij de implementatie van het MVC-patroon zijn onderverdeeld in verschillende Packages. De views hebben we gemaakt in de packages `rolit.client.view` en `rolit.server.view`. De onderdelen communiceren met elkaar zoals je zou verwachten. Zodra er een actie plaatsvindt in één van de controllers, worden de views geüpdatet.

Controllers

We hebben een `ClientController` die aan de Client-kant het spel 'onderhoudt'. Deze zorgt ervoor dat we berichten naar de console kunnen sturen en dat we met de lobby en de players de game kunnen starten.

Verder hebben we een `NetworkController` die alle communicatie afhandelt en alle commando's herkent en interpreteert. De `NetworkController` heeft methoden om te luisteren naar nieuwe verbindingen en om de verbinding te stoppen. Ook de `NetworkController` kan commando's sturen met `sendMessage()`. Voor de server heeft deze klasse functionaliteit om spelers te beheren.

`ClientHandlerController`¹ is de klasse die de server-kant van het programma gebruikt om de client-kant te bedienen. Deze controller kan de commando's opvangen die door de `NetworkController` uitgevoerd worden.

Verder hebben we nog de `ServerController` klasse die de methoden (en datastromen) van de Server onderhoudt en de acties kan uitvoeren die de server zou willen uitvoeren.

Ten slotte zouden we ook een `AuthenticationController` geïmplementeerd hebben, ware het niet dat we deze niet volledig werkend kregen. We ontvingen telkens dat het ID niet bekend was en we hebben daarom besloten de `AuthenticationController` achterwege te laten.

View

De View klassen zijn ervoor om de GUI's te onderhouden van ons programma. De view krijgt commando's (of interacties) van de controllers en acteert daarop.

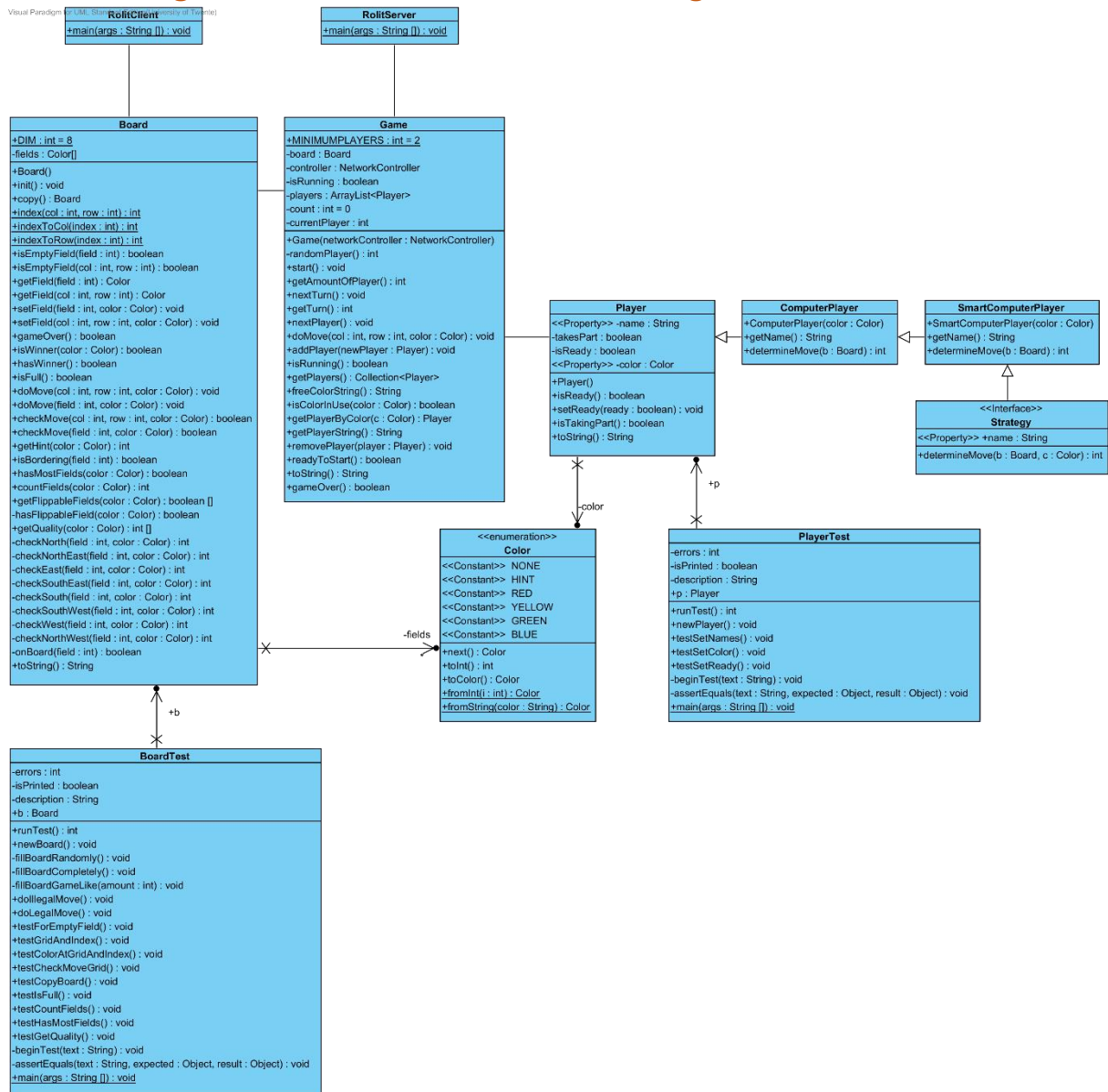
We hebben de `ServerGUI` die alle elementen instantieert en de verbinding legt met de `ServerController`. Zodra er een update komt van de controllers zal de GUI deze tonen.

We hebben ook een `ConnectGUI`, die met behulp van de `NetworkController` en `ClientController` ervoor zorgt dat de gebruiker van het programma het spel kan beginnen. Er zijn listeners voor de `ComboBox` en de `TextFields`.

Ten slotte hebben we de `GameGUI`, die het spelbord laat zien en ervoor zorgt dat de gebruiker het spel kan spelen. Deze klasse maakt alle veldjes en verandert ze met behulp van de `SetField()` methode.

¹ Delen van de code zijn overgenomen van voorbeeldcode van Theo Ruys, zoals vermeld in de JavaDOC

Klassendiagram van klassen van het Model-gedeelte



Functionele eisen van het serverprogramma

Betrokken klassen en de implementatie

De implementatie van de server zorgt ervoor dat het spel gespeeld kan worden zonder de spelregels te overtreden, maar ook dat nieuwe clients kunnen verbinden.

De implementatie van het MVC-patroon zorgt ervoor dat de server kan communiceren en aan de eisen voldoet, namelijk dat zetten in het spel alleen gedaan mogen worden als ze volgens de spelregels zijn en als de speler aan de beurt is.

Rollen van de server

De server moet een aantal dingen kunnen.

1. Hij moet nieuwe verbindingen kunnen accepteren door te luisteren op een poort.

2. Hij moet de zetten die spelers willen doen, valideren. Dus hij moet ervoor zorgen dat er geen illegale zetten gedaan worden. Dit wordt in principe ook aan de client-kant gedaan, maar om valsspelers tegen te gaan controleert de server het ook nog.
3. Hij moet zelf een speelbord bijhouden, om te testen op zetten.
4. Hij moet nieuwe clients een kleur laten kiezen.
5. Hij moet volgens de regels van het protocol kunnen communiceren.

Werking van de server

De server maakt in zijn `ServerController` een nieuwe `Socket` waarop hij kan luisteren. Elke keer als er een nieuwe client komt, maakt de server een nieuwe `ClientHandler` aan om de client te bedienen en alle verdere communicatie afhandelt. Die `ClientHandlers` worden in nieuwe threads gemaakt in een collectie van `ClientHandlers` op de server. Dit gebeurt in de `NetworkController`.

Testrapporten

Opgestelde tests

Voor de `Board` klasse hebben we het grootste aantal tests opgebouwd, omdat de rest van de werking het programma hier op rust. Als er een fout in `Board` zit, wordt het spel op een zeker moment onspeelbaar.

De methode die we hebben gebruikt om de klassen te testen is geïnspireerd door de methode² die werd gebruikt om de opdrachten tijdens de rest van de module te controleren op fouten. Er is een `assertEquals`-methode die controleert of de uitkomst van een test hetzelfde is als de verwachte uitkomst.

De tests zijn uit te voeren door `rolit.test.Test.java` uit te voeren, deze klasse heeft een `main`-methode.

Board

Om de tests te ondersteunen hebben we in de testklasse een aantal hulpmethoden geïmplementeerd. Deze zijn `fillBoardRandomly()`, die een willekeurig aantal vakjes op willekeurige plaatsen met willekeurige kleuren vult; `fillBoardCompletely()`, die het complete bord vult met willekeurige kleuren; `fillBoardGameLike(int)`, die het bord vult als een gamesituatie, met de kleuren op de spelvolgorde (volgens de techniek van de `smartStrategy`) voor het meegegeven aantal zetten (`int`).

`doIllegalMove()`

Probeert een ongeldige zet te doen en test of deze zet geweigerd (genegeerd) wordt en het veld `Color.NONE` blijft.

Verwachte uitkomst: `Color.NONE`

Daadwerkelijke uitkomst: `Color.NONE`

`doLegalMove()`

Probeert een geldige zet met de kleur `RED` te doen, afgeleid van `getFlippableFields` die een lijst maakt van geldige zetten. Nadat de zet gedaan is wordt gekeken of de zet daadwerkelijk gedaan is.

² Deze methode is opgesteld door Arend Rensink

Verwachte uitkomst: Color.RED

Daadwerkelijke uitkomst: Color.RED

testForEmptyField()

Kijkt voor alle veldjes of isEmptyField(int) van Board true geeft als het veldje Color.NONE heeft.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

testGridAndIndex()

Kijkt voor elk veldje of de vertaling van index(col, row) de juiste index terug geeft. Dit gebeurt door voor alle veldjes index(indexToCol(index), indexToRow(index)) aan te roepen.

Verwachte uitkomst: true (voor elk veld)

Daadwerkelijke uitkomst: true (voor elk veld)

testColorAtGridAndIndex()

Maakt eerst een willekeurig gevuld bord, met fillBoardRandomly(). Kijkt voor elk veldje of de benadering per grid dezelfde Color terug geeft als de benadering per index. Als er een fout wordt gevonden terwijl de for-loop wordt uitgevoerd, wordt foundMistake op true gezet.

Verwachte uitkomst: false

Daadwerkelijke uitkomst: false

testCheckMoveGrid()

Test of checkMove() bij de grid-benadering voor elk veld hetzelfde resultaat geeft als bij de index-benadering. Als er een fout wordt gevonden terwijl de for-loop over de velden wordt uitgevoerd, wordt foundMistake op true gezet.

Verwachte uitkomst: false

Daadwerkelijke uitkomst: false

testCopyBoard()

Roept eerst fillBoardRandomly() aan die een willekeurig bord genereert en maakt daarna een kopie met copy(). Ten slotte wordt getest of de velden in copy dezelfde waarden hebben als het oorspronkelijke bord. Als er een fout wordt gevonden terwijl de for-loop wordt uitgevoerd, zeggen we foundMistake = true.

testIsFull()

Maakt met de functie fillBoardCompletely() een compleet, willekeurig gevuld bord. Dan kijken we of de isFull() methode van Board true teruggeeft.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

testCountFields()

Roept eerst fillBoardRandomly() aan om een willekeurig bord te maken. Dan wordt met een handmatige functie een som gemaakt van de gekleurde veldjes voor elke kleur. Deze sommen worden naast de resultaten van countFields() gehouden en vergeleken. Als een fout gevonden wordt, zeggen we passed = false.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

Dekking (coverage) van de BoardTest.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
rolit	84,8 %	1.218	218	1.436
Board.java	90,5 %	1.085	114	1.199
Board	90,5 %	1.085	114	1.199
hasMostFields(Color)	0,0 %	0	77	77
hasWinner()	0,0 %	0	20	20
doMove(int, int, Color)	0,0 %	0	8	8
isWinner(Color)	0,0 %	0	4	4
gameOver()	0,0 %	0	3	3
isFull()	89,5 %	17	2	19
Board()	100,0 %	21	0	21
checkEast(int, Color)	100,0 %	30	0	30
checkMove(int, Color)	100,0 %	26	0	26
checkMove(int, int, Color)	100,0 %	8	0	8
checkNorth(int, Color)	100,0 %	26	0	26
checkNorthEast(int, Color)	100,0 %	32	0	32
checkNorthWest(int, Color)	100,0 %	34	0	34
checkSouth(int, Color)	100,0 %	26	0	26
checkSouthEast(int, Color)	100,0 %	32	0	32
checkSouthWest(int, Color)	100,0 %	34	0	34
checkWest(int, Color)	100,0 %	32	0	32
copy()	100,0 %	23	0	23
countFields(Color)	100,0 %	19	0	19
doMove(int, Color)	100,0 %	178	0	178
getField(int)	100,0 %	5	0	5
getField(int, int)	100,0 %	7	0	7
getFlippableFields(Color)	100,0 %	110	0	110
getHint(Color)	100,0 %	9	0	9
getQuality(Color)	100,0 %	152	0	152
hasFlippableField(Color)	100,0 %	19	0	19
index(int, int)	100,0 %	6	0	6
indexToCol(int)	100,0 %	4	0	4
indexToRow(int)	100,0 %	4	0	4
init()	100,0 %	21	0	21
isBordering(int)	100,0 %	164	0	164
isEmptyField(int)	100,0 %	9	0	9
isEmptyField(int, int)	100,0 %	7	0	7
onBoard(int)	100,0 %	9	0	9
setField(int, Color)	100,0 %	13	0	13
setField(int, int, Color)	100,0 %	8	0	8
Color.java	59,3 %	108	74	182
Player.java	45,5 %	25	30	55

Figuur 1 - Dekking van BoardTest

Player

Player is getest op of de bewerkingen die we zouden willen doen, mogelijk zijn.

testSetNames()

Probeert de naam van een speler te veranderen en kijkt of de originele naam verschilt met de nieuwe naam.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

testSetColor()

Kijkt of de kleur van de speler verandert als setColor() wordt uitgevoerd. De test zet eerst de variabele orig naar de originele kleur en gaat met setColor() de kleur veranderen naar Color.next(). De orig wordt vergeleken met deze nieuwe waarde.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

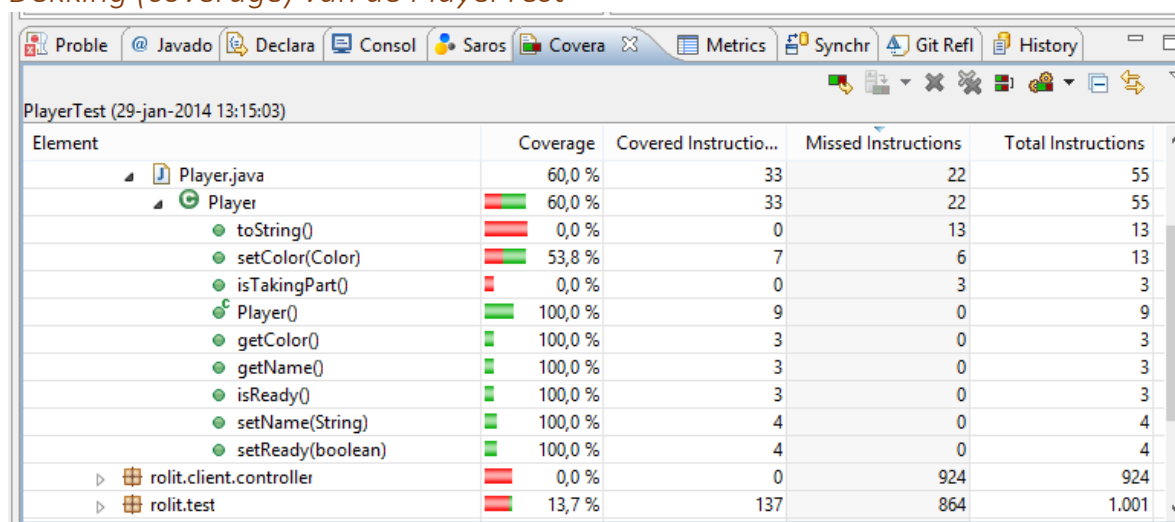
testSetReady()

Kijkt of de boolean `isReady` verandert als `setReady(boolean arg)` wordt uitgevoerd. De oorspronkelijke waarde wordt in `orig` gezet en daarna wordt de oorspronkelijke waarde gewijzigd naar `!orig`. Ten slotte wordt gekeken of `orig` ongelijk is aan het resultaat van `isReady`.

Verwachte uitkomst: true

Daadwerkelijke uitkomst: true

Dekking (coverage) van de PlayerTest



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
PlayerTest (29-jan-2014 13:15:03)				
Player.java	60,0 %	33	22	55
Player	60,0 %	33	22	55
toString()	0,0 %	0	13	13
setColor(Color)	53,8 %	7	6	13
isTakingPart()	0,0 %	0	3	3
Player()	100,0 %	9	0	9
getColor()	100,0 %	3	0	3
getName()	100,0 %	3	0	3
isReady()	100,0 %	3	0	3
setName(String)	100,0 %	4	0	4
setReady(boolean)	100,0 %	4	0	4
rolit.client.controller	0,0 %	0	924	924
rolit.test	13,7 %	137	864	1.001

Figuur 2 - Dekking van PlayerTest

Interactie met behulp van het protocol

In het protocol staan alle acties die we moeten uitvoeren om met elkaar te kunnen interacteren. Het uitgangspunt was dat we commando's kunnen broadcasten op de volgende manier: "Commando int int int". Bijvoorbeeld als de GameGUI een zet wil sturen naar de server, dan gaat dat in het formaat String "Move int_ID int_col int_row", bijvoorbeeld "Move 0 2 3".

Bijna alle berichten worden op deze manier verstuurd en het is aan de server de taak om deze commando's te herkennen en te interpreteren.

De gedachte achter deze manier van berichten versturen was dat de commando's makkelijk te interpreteren zijn, omdat we ze kunnen laten binnenkomen in een Scanner, die alle Strings woord voor woord kan ontleden. Dus stel dat we een commando krijgen "Turn 21 2", kunnen we met `in.next()` eerst herkennen dat het om het commando Turn gaat, dat het de 21^{ste} turn wordt en dat die turn voor de speler met id 2 is.

Voor een uitgebreide beschrijving van het protocol en haar commando's verwijzen we door naar de documentatie van het *bit-A MODULE 2 Tcp-protocol* oftewel het AMULET-protocol³, hier staat precies in vermeld hoe (en met welke commando's) de server moet reageren op nieuwe verbindingen, etc.

³ Het AMULET-protocol is bepaald door BIT A en uitgewerkt door Stefan Braams en Rick Harms

Dataformats

Alle berichten worden als String verstuurd tussen de server en de client. Zoals eerder gezegd wordt op deze Strings een scanner losgelaten die telkens `in.next()` doet en met de uitkomst haar commando's uitvoert.

Planning

We zijn tweeëneenhalve week voor het inlevermoment begonnen met de implementatie van het programma. Tot dat moment zijn we voornamelijk bezig geweest met de standaardopdrachten voor programmeren. We hebben de volgende werkwijze gehandhaafd:

1. Implementeer een klasse `Board.java`, die alle basisbewerkingen die je op een RollIt-bord zou willen doen, mogelijk maakt. Bijvoorbeeld: maak een array met de inhoud van de vakjes, verander de kleur van het vakje.
2. Gaandeweg gingen we verder met de implementatie van de spelregels. Deze hebben we ook in `Board` geïmplementeerd. Bijvoorbeeld: een zet moet altijd in één van de windrichtingen aangrenzend zijn met een reeds gevuld vakje, of een zet moet indien mogelijk een andere speler blokkeren. Voor de implementatie van de spelregels hebben we de wiskunde gebruikt zoals beschreven in bijlage 2.
3. Daarna gaan we een GUI bouwen om te testen of de implementatie van de spelregels en de velden allemaal werkt.
4. Werkt dat volledig (oftewel is een offline game mogelijk)? Dan gaan we de server-implementatie bouwen met behulp van het MVC-patroon.

Wijzigingen in het plan

Het oorspronkelijke plan was om een authenticatie-check te bouwen die ervoor zorgde dat we ons konden identificeren met een username en password, maar we kregen het voortdurend niet werkend. Toen de aangepaste eisen vrijkwamen, hebben we besloten de implementatie van de authenticatie te laten vallen.

Verder hadden we nog graag de GUI wat strakker en mooier willen maken, maar dat is door tijdnood niet gelukt. Daarom hebben we besloten de GUI te laten zoals die nu is.

Reflectie

We zijn er van overtuigd dat we het project niet beter hadden kunnen plannen, omdat de reguliere opdrachten van de module ook al waren uitgelopen terwijl we wel de functionaliteit van die opdrachten nodig hadden. We hebben naar eigen idee erg hard gewerkt om de implementatie op tijd af te krijgen, met de kennis die we hadden.

Bijlagen

1 - Raster van het speelbord als DIM == 8

Tabel 1 - Speelbord van RollIt

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

2 - Wiskunde achter de vakjes-checks.

Er zijn acht methoden die recursief in een windrichting testen of er een vakje is met dezelfde kleur als de geteste kleur.

Om naar het volgende vakje te komen om te testen, moet je de volgende berekeningen doen:

1. CheckNorth - DIM
2. CheckNorthEast - DIM + 1
3. CheckEast + 1
4. CheckSouthEast + DIM + 1
5. CheckSouth + DIM
6. CheckSouthWest + DIM - 1
7. CheckWest - 1
8. CheckNorthWest - DIM - 1

Om te berekenen wat het verschil is tussen het gevonden vakje (Ans) en het geteste vakje (i):

1. CheckNorth $(i - \text{Ans}) / \text{DIM}$
2. CheckNorthEast $(i - \text{Ans}) / (\text{DIM} - 1)$
3. CheckEast $\text{Ans} - i$
4. CheckSouthEast $(\text{Ans} - i) / (\text{DIM} + 1)$
5. CheckSouth $(\text{Ans} - i) / \text{DIM}$
6. CheckSouthWest $(\text{Ans} - i) / (\text{DIM} - 1)$
7. CheckWest $i - \text{Ans}$
8. CheckNorthWest $(i - \text{Ans}) / (\text{DIM} + 1)$