PROGRAMMING PROJECT

SOFTWARE SYSTEMS

Jerre Starink  
Amber Altenburg  
januari 2016  
Universiteit Twente

Table of Contents

[1 Discussion of the Overall Design 3](#_Toc441877866)

[1.1 Class diagram 3](#_Toc441877867)

[1.2 Requirements 4](#_Toc441877868)

[1.2.1 Requirements voor de server 4](#_Toc441877869)

[1.2.2 Requirements voor de client 4](#_Toc441877870)

[1.3 Observer and Model-View-Controller patterns 5](#_Toc441877871)

[1.3.1 Model 5](#_Toc441877872)

[1.3.2 View 6](#_Toc441877873)

[1.3.3 Controller 6](#_Toc441877874)

[1.3.4 Observer 6](#_Toc441877875)

[1.4 Data storage and communication protocols 7](#_Toc441877876)

[2 Discussion per Class 8](#_Toc441877877)

[2.1 ArrayBoard 8](#_Toc441877878)

[2.2 Board 8](#_Toc441877879)

[2.3 Game 9](#_Toc441877880)

[2.4 Location 9](#_Toc441877881)

[2.5 Move 9](#_Toc441877882)

[2.6 MoveResult 9](#_Toc441877883)

[2.7 OpenHandPlayer 9](#_Toc441877884)

[2.8 Player 10](#_Toc441877885)

[2.9 Protocol 10](#_Toc441877886)

[2.10 Rectangle 10](#_Toc441877887)

[2.11 RemotePlayer 11](#_Toc441877888)

[2.12 Sequence 11](#_Toc441877889)

[2.13 SequenceDirection 11](#_Toc441877890)

[2.14 SequenceType 11](#_Toc441877891)

[2.15 Stone 11](#_Toc441877892)

[2.16 StoneType 12](#_Toc441877893)

[2.17 StoneColor 12](#_Toc441877894)

[2.18 View 12](#_Toc441877895)

[2.19 ComputerPlayer 12](#_Toc441877896)

[2.20 HumanPlayer 12](#_Toc441877897)

[2.21 NaiveStrategy 13](#_Toc441877898)

[2.22 QwirkleClient 13](#_Toc441877899)

[2.23 Strategy 13](#_Toc441877900)

[2.24 TUIView 14](#_Toc441877901)

[2.25 ClientHandler 14](#_Toc441877902)

[2.26 ClientPlayer 14](#_Toc441877903)

[2.27 GameRequest 14](#_Toc441877904)

[2.28 QwirkleServer 14](#_Toc441877905)

[3 Test Report 16](#_Toc441877906)

[3.1 Unit Testing 16](#_Toc441877907)

[3.1.1 LocationTest 16](#_Toc441877908)

[3.1.2 RectangleTest 16](#_Toc441877909)

[3.1.3 StoneTest 16](#_Toc441877910)

[3.1.4 MoveTest 17](#_Toc441877911)

[3.1.5 RemotePlayerTest 17](#_Toc441877912)

[3.1.6 GameTest 17](#_Toc441877913)

[3.1.7 ArrayBoardTest 17](#_Toc441877914)

[3.1.8 SequenceTest 18](#_Toc441877915)

[3.1.9 CheckMoveTest 18](#_Toc441877916)

[3.1.10 ScoreTest 18](#_Toc441877917)

[3.1.12 Samenvatting 19](#_Toc441877918)

[3.2 Systeemtest 20](#_Toc441877919)

[3.2.1 Testen van de TUI 20](#_Toc441877920)

[3.2.2 Testen van het netwerk protocol 20](#_Toc441877921)

[3.2.3 Testen van de AI 20](#_Toc441877922)

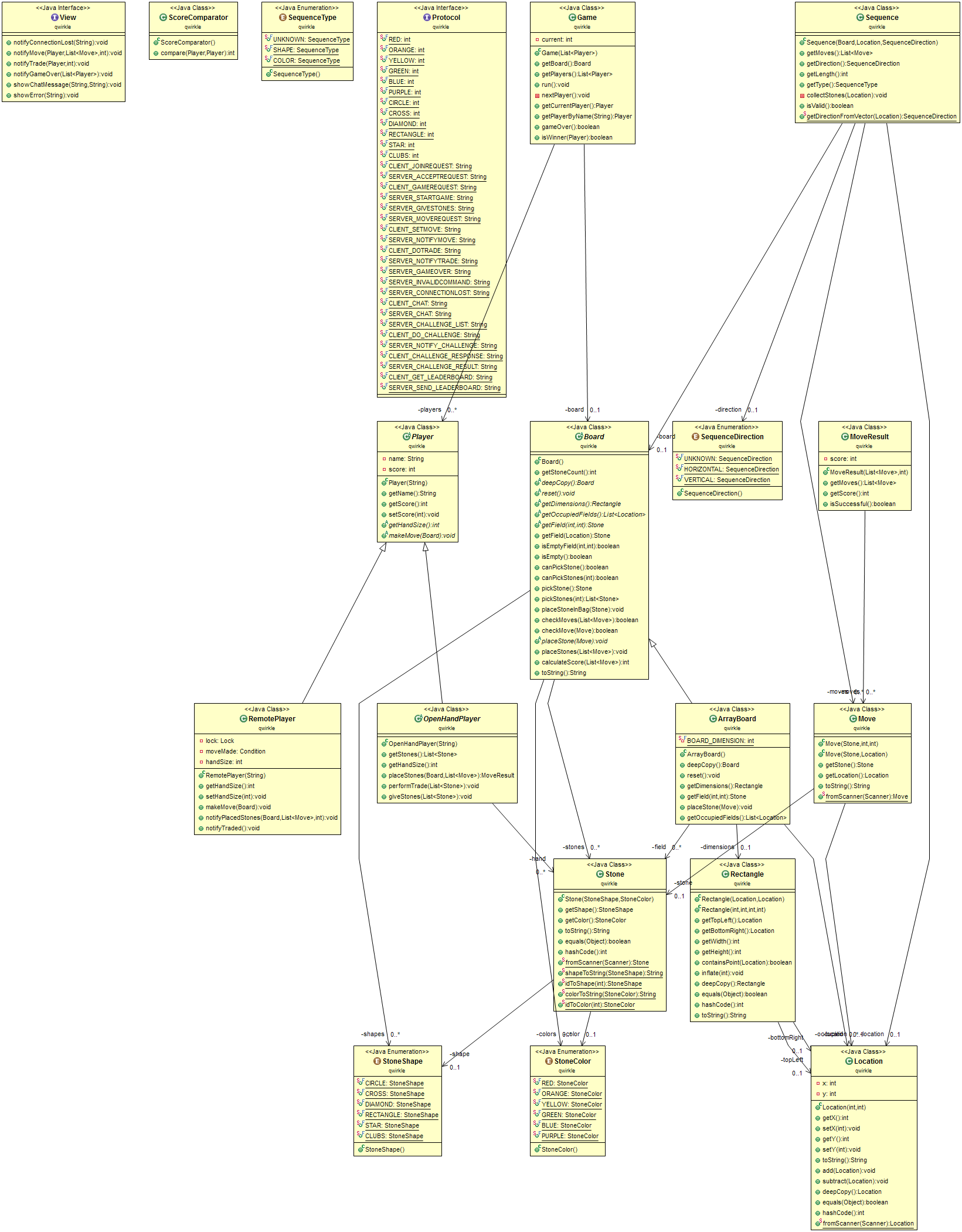
[4 Metrics Report 21](#_Toc441877923)

[5 Known issues 22](#_Toc441877924)

[6 Reflection on Planning 23](#_Toc441877925)

# 1 Discussion of the Overall Design

## 1.1 Class diagram



zie ook: class-diagram-programming-project.png

## 1.2 Requirements

### 1.2.1 Requirements voor de server

|  |  |
| --- | --- |
|  | **Geïmplementeerd door** |
| Wanneer de server wordt gestart, moet er een poortnummer ingevuld worden waarnaar de server zal luisteren. | *zie Known Issues* |
| Als het poortnummer al in gebruik is, wordt er een passende foutmelding teruggegeven en kan er een nieuw poortnummer worden ingevoerd. | QwirkleServer |
| Een server moet meerdere exemplaren van het spel, die tegelijkertijd door verschillende cliënten worden gespeeld, kunnen ondersteunen. | QwirkleServer |
| De server heeft een TUI die ervoor zorgt dat alle communicatie berichten naar *System.out* worden geschreven. | TUIView |
| De server moet het protocol zoals gedefinieerd voor de tutorial groep tijdens de projectsessie in week 7 respecteren, dat wil zeggen, de server moet kunnen communiceren met alle andere cliënten uit de tutorial groep. 1.2.2 Requirements voor de client | Protocol,  QwirkleClient,  QwirkleServer,  ClientHandler |
|  | **Geïmplementeerd door** |
| De client moet een gebruiksvriendelijke TUI hebben, welke opties aan de gebruiker biedt om een spel aan te vragen bij de server. | TUIView |
| De client moet zowel menselijke spelers als computer spelers met kunstmatig intelligent gedrag ondersteunen. | HumanPlayer, ComputerPlayer, NaiveStrategy, SmartStrategy |
| De bedenktijd van de computer speler moet een parameter zijn die aangepast kan worden via de client TUI. | zie *Known Issues* |
| De client heeft een *hint* functionaliteit. Deze functie toont een menselijke speler een mogelijke zet, zoals aangegeven door de computer speler. De zet mag alleen worden voorgesteld, de menselijke speler moet de mogelijkheid hebben om zelf te beslissen of hij de voorgestelde zet of een andere zet wil doen. | TUIView |
| Als een spel is afgelopen moet de speler een nieuw spel kunnen starten. | TUIView |
| Als een speler het spel stopt voordat het is afgelopen, of als de client crasht, moeten de andere spelers geïnformeerd en het spel netjes afgesloten worden. In dit geval moeten de andere spelers zich kunnen registreren bij de server voor een nieuw spel. | TUIView |
| Een server kan te allen tijden ontkoppelen. De cliënten moeten hier goed op kunnen reageren door alle open verbindingen te sluiten etc. | QwirkleClient,  TUIView |
| De client moet het protocol zoals gedefinieerd voor de tutorial groep tijdens de projectsessie in week 7 respecteren, dat wil zeggen, de client moet kunnen communiceren met alle andere servers uit de tutorial groep. | ClientHandler |

## 1.3 Observer and Model-View-Controller patterns

### 1.3.1 Model

De volgende klassen vormen het model:

* *ArrayBoard.java*
* *Board.java*
* *Game.java*
* *Location.java*
* *Move.java*
* *MoveResult.java*
* *OpenHandPlayer.java*
* *Player.java*
* *Protocol.java*
* *Rectangle.java*
* *RemotePlayer.java*
* *ScoreComparator.java*
* *Sequence.java*
* *SequenceDirection.java*
* *SequenceType.java*
* *Stone.java*
* *StoneColor.java*
* *StoneShape.java*
* *View.java*
* *ComputerPlayer.java*
* *HumanPlayer.java*
* *NaiveStrategy.java*
* *PlayerType.java*
* *SmartStrategy.java*
* *Strategy.java*
* *GameRequest.java*

Bovenstaande klassen representeren de gegevens van het Qwirkle programma. Zij beheren de gegevens en voeren alle transformaties van die gegevens uit. Verder zijn deze klassen niet op de hoogte van de controllers en views; ze bevatten geen verwijzingen naar beiden.

### 1.3.2 View

De volgende klassen vormen de view:

* *TUIView.java*
* *View.java*

Bovenstaande klassen beheren de visuele weergave van de gegevens die door het model worden gerepresenteerd. De klassen zorgen ervoor dat er data zichtbaar wordt voor de gebruikers van het Qwirkle programma. Zo wordt er in *View.java* bijvoorbeeld een TUI gecreëerd.

### 1.3.3 Controller

De volgende klassen vormen de controller:

* *ClientHandler.java*
* *QwirkleClient.java*
* *QwirkleServer.java*
* *ServerHandler.java*

De controller klassen combineren de gegevens uit het model met de view. Daarnaast reageren de controllers op gebeurtenissen die meestal het gevolg zijn van handelingen van een gebruiker, bijvoorbeeld het doen van een zet.

Waar het bij het model en de view duidelijk is waarom de genoemde klassen tot deze objecten behoren, is dit bij het controller object niet altijd even duidelijk. // TODO: splitsen of niet

### 1.3.4 Observer

Java biedt ondersteuning voor het Model-View-Controller patroon door middel van de twee klassen:

* Observer - een object dat gewaarschuwd wil worden wanneer de toestand van een ander object verandert
* Observable - een object waarvan de staat van belang kan zijn, en bij wie een ander object iets belangrijks kan registreren

Waarom en welke klassen Observer/Observable zijn wordt beschreven in sectie 2 *Discussion per Class*.

## 1.4 Data storage and communication protocols

De grootste hoeveelheid data die wordt opgeslagen in het geheugen van het spel is het bord. Omdat dit bord dynamisch is hebben we overwogen om een tweedimensionale *List* te gebruiken. Echter hebben we toch gekozen voor een tweedimensionale array die groot genoeg is om elk spel te kunnen spelen. Hoe we dit implementeren staat beschreven in de beschrijving van de klasse ArrayBoard.

Daarnaast hebben we te maken met een groot aantal stenen in de stapel en in de handen van de spelers. Hiervoor hebben we wel de *List* klasse gekozen. Omdat het in beide situaties slechts een kwestie van stenen toevoegen en verwijderen is, is dat gemakkelijk te behalen met een dergelijke klasse.

Daarnaast was het noodzakelijk om een protocol vast te stellen voor de communicatie tussen de server en de cliënten. Hoe dit protocol eruit zou gaan zien, hebben we afgesproken tijdens een van de werkcolleges, en kan gezien worden in *Protocol.java.*

# 2 Discussion per Class

## 2.1 ArrayBoard

*ArrayBoard* levert, als uitbreiding van de klasse Board, aan de hand van een 2D array als back-end een implementatie voor *Board*. Om tot deze implementatie te komen, maakt *ArrayBoard* gebruik van de klassen Stone en *Rectangle*.

Doordat er bij Qwirkle continu blokjes worden toegevoegd aan het speelbord is er sprake van een dynamisch bord. Toch wordt er in onze code gebruik gemaakt van statische dimensies; dit om de code zo simpel mogelijk te houden. Om te voorkomen dat er in de TUI continu een bord van 100 bij 100 velden zou worden geprint, hebben we ervoor gezorgd dat alleen de velden worden geprint waarop een steen ligt.

Bij het printen van het bord gaan we uit van een rechthoek die gedefinieerd wordt door middel van 2 punten, namelijk: de linker bovenhoek (*topLeft*) en de rechter benedenhoek (*bottomRight*). Bij het plaatsen van de eerste steen is er sprake van een rechthoek waarbij zowel de coördinaten van *topLeft* als de coördinaten van *bottomRight* (0,0) zijn.

Bij de uitleg over het printen van het bord wordt er uitgegaan van onderstaande afbeelding. Bij dit bord heeft *topLeft* de coördinaten (-1, 1) en *bottomRight* (1, -1). Vanaf dit punt zijn er drie mogelijkheden die invloed kunnen hebben op de grootte van het bord. Er kan een steen aan de linkerzijde of aan de rechterzijde van het bord worden gelegd. Daarnaast kan er ook een steen worden gelegd binnen het bord (op een leeg veld).

Stel, er wordt een steen geplaatst aan de rechterzijde van 3R. In dit geval is de x-coördinaat van de nieuwe steen groter dan de x-coördinaat van de huidige *bottomRight*. Dit betekent dat het bord moet worden uitgebreid. Stel, er wordt een steen geplaatst boven 3Y. In dat geval is de x-coördinaat van de nieuwe steen kleiner dan de x-coördinaat van de huidige *bottomRight*. Dit betekent dat het bord niet hoeft te worden uitgebreid.

In *ArrayBoard* wordt er bij het leggen van een steen gecheckt of:

* de x-coördinaat van de nieuwe steen kleiner is dan de x-coördinaat van *topLeft*;
* de y-coördinaat van de nieuwe steen groter is dan de y-coördinaat van *topLeft*;
* de x-coördinaat van de nieuwe steen groter is dan de x-coördinaat van *bottomRight*;
* de y-coördinaat van de nieuwe steen kleiner is dan de y-coördinaat van *bottomRight*;

Wanneer een van bovenstaande gevallen een positief resultaat oplevert, wordt het bord uitgebreid.



## 2.2 Board

Zoals de naam al doet vermoeden, vervult *Board* de rol van het speelbord van Qwirkle. *Board* heeft dan ook de verantwoordelijkheid om te zorgen voor de representatie van het speelbord. Daarnaast is *Board* verantwoordelijk voor het checken van moves; zij moet controleren of een zet geldig is. Verder is *Board* ook verantwoordelijk voor de zak met stenen; zij houdt bij hoeveel en welke stenen er nog in de zak zitten. Om de bovenstaande verantwoordelijkheden te kunnen nemen, maakt *Board* gebruik van de klassen *StoneShape*, *StoneColor* en *Stone*.

Voor de *View* is de status van het bord van groot belang; als speler wil je natuurlijk weten hoe het bord eruit ziet. *Board* heeft binnen het *Observer* patroon dan ook de rol van *Observable*. De View kan als *Observer* registreren welke veranderingen er binnen het bord hebben plaatsgevonden. Denk hierbij bijvoorbeeld aan de dimensies van het bord en nieuw geplaatste stenen op het bord.

## 2.3 Game

De klasse *Game* is verantwoordelijk voor de handhaving van een Qwirkle spel. Dit doet zij allereerst door bij te houden welke spelers er mee doen (als een lijst van ‘*Player*’s). Vervolgens wordt er bijgehouden welke speler er aan zet is. Verder wordt ook het speelbord (als instantie van *Board*) bijgehouden in *Game*.

*Game* maakt gebruik van de klassen *Player* en *Board* om aan haar verantwoordelijkheden te kunnen voldoen. De *Game* is ook *Observable*. Hij laat zijn observers weten wanneer een spel is afgelopen.

## 2.4 Location

De klasse *Location* vervult de rol van een locatie op het Qwirkle bord. Het is een relatief simpele klasse met twee attributen X en Y. Het kan ook gebruikt worden als een vector. Het heeft een aantal operaties *add* en *subtract* die coördinaat gewijs de twee vectoren bij elkaar optelt of aftrekt.

## 2.5 Move

*Move* representeert het plaatsen van een steen binnen het Qwirkle spel. Om een steen op het bord te kunnen plaatsen, vraagt Move de steen die geplaatst moet worden en de gewenste locatie op. Om deze gegevens op te kunnen vragen, heeft *Move* de klassen *Stone* en *Location* nodig.

## 2.6 MoveResult

De methode *OpenHandPlayer*.*placeStones* geeft een instantie van de *MoveResult* klasse. Deze klasse beschrijft het resultaat van het plaatsen van stenen op het bord. Het geeft aan welke zet geprobeerd is, of hij geldig was of niet, en hoeveel score ermee verdiend is.

## 2.7 OpenHandPlayer

In eerste instantie hadden we alleen te maken een ‘human player’. Toen ook de ‘computer player’ in beeld kwam, hebben we nog eens goed nagedacht hoe we de client met deze verschillende soorten spelers om konden laten gaan. Omdat zowel de *HumanPlayer* als de *ComputerPlayer* spelers zijn die hun hand publiekelijk vrijgeven, hebben we een klasse *OpenHandPlayer* gedefinieerd.

De *OpenHandPlayer* is niet alleen handig voor de client, maar ook voor de server. De server moet namelijk van alle spelers bijhouden welke stenen zij in hun bezit hebben.

Naast de *OpenHandPlayer* hebben we ook een *RemotePlayer* gedefinieerd (zie sectie 2.11). In deze klasse kan alleen het aantal stenen in de hand worden opgevraagd en dus niet welke stenen.

*OpenHandPlayer* representeert een speler die zijn hand met stenen publiekelijk vrijgeeft. De klasse is een uitbreiding van *Player* waarin de stenen van een speler kunnen worden opgevraagd en de stenen van een speler op het bord geplaatst kunnen worden. Het opvragen van de stenen levert een lijst met stenen op; hier wordt dus gebruik gemaakt van de klasse Stone. Bij het plaatsen van stenen op het bord worden het bord en een lijst van moves als argument meegegeven; hier maakt *OpenHandPlayer* gebruik van Board en Move.

## 2.8 Player

*Player* representeert een speler van het Qwirkle spel en is verantwoordelijk voor het bijhouden van de score van een speler en het bijhouden van het aantal stenen in de hand van een speler.

Tijdens het spelen van een spel wil je als speler graag weten wat de scores van de andere spelers zijn en hoeveel stenen iedereen heeft. Dit zijn dan ook de redenen waarom *Player* een *Observable* object is. Doordat Player *Observable* is kan de *View* registreren wat de score van een speler is en hoeveel stenen een speler heeft. Deze gegevens kunnen vervolgens weer worden weergegeven bij de andere spelers.

## 2.9 Protocol

In het protocol wordt gedefinieerd hoe alle operaties binnen het netwerk moeten samenwerken. Dit zorgt ervoor dat alle spelers die gebruik maken van hetzelfde protocol Qwirkle via een server kunnen spelen.

Zo moet er in het protocol bijvoorbeeld de representatie van een steen gedefinieerd worden. Dit wordt gedaan om te voorkomen dat in een spel met 2 spelers de ene speler ‘1G’ voor een gele steen gebruikt en de andere speler ‘1Y’.

## 2.10 Rectangle

Zoals besproken in sectie 2.1 wordt er bij het printen van het bord gebruikt gemaakt van een rechthoek die gedefinieerd wordt door middel van 2 punten. Deze rechthoek wordt gedefinieerd in de klasse *Rectangle*. De twee punten waarmee de rechthoek wordt gecreëerd, zijn de zogenaamde linker bovenhoek (*topLeft*) en de rechter benedenhoek (*bottomRight*). Deze punten, *topLeft* en *bottomRight*, zijn variabelen van het type Location. Rectangle maakt dan ook veelvoudig gebruik van de klasse Location.

## 2.11 RemotePlayer

*RemotePlayer* representeert een speler waarvan de inhoud van de hand onbekend is. Aangezien het niet de bedoeling is dat je de inhoud van de hand van je tegenstanders ziet, zou je een ‘remote player’ als een tegenstander kunnen beschouwen.

*RemotePlayer* is een implementatie van *Player* waarin daadwerkelijk een score aan een speler wordt gegeven en waarin het aantal stenen in de hand van een speler kan worden opgevraagd.

## 2.12 Sequence

*Sequence* representeert een rij van stenen op een Qwirkle bord. Met behulp van een beginpositie en een richting kan de hele rij verkregen worden op het bord. Het bevat een aantal query methoden die de rij goed beschrijft.

Met behulp van de *getLength* en *getMoves* methodes kan er nagegaan worden hoeveel en welke plaatsingen van stenen er nodig waren om deze rij te construeren. Daarnaast kan de richting worden verkregen met de *getDirection* methode. Met de *getType* query kan men te weten komen of deze rij uit dezelfde vormen of kleuren bestaat, of dat het nog vrij te kiezen is. Als laatste is het mogelijk om de geldigheid van de rij te controleren met de *isValid* methode.

## 2.13 SequenceDirection

Aangezien we bij de richting van een rij met stenen te maken hebben met een vast aantal mogelijkheden, hebben we ervoor gekozen om van *SequenceDirection* geen normale klasse te maken maar een enum type. *SequenceDirection* bevat een vooraf gedefinieerde set van de mogelijke richtingen, namelijk: *VERTICAL*, *HORIZONTAL* en *UNKWOWN*.

## 2.14 SequenceType

Net zoals bij de richting van een rij met stenen is er bij het type van een rij met stenen ook een vast aantal mogelijkheden. Daarom wordt ook het type van een rij met stenen gerepresenteerd in een enum type. *SequenceType* bevat een vooraf gedefinieerde set van de mogelijke typen, namelijk: *UNKWOWN*, *COLOR* en *SHAPE*.

## 2.15 Stone

De klasse *Stone* representeert een steen in het Qwirkle spel. Een steen heeft twee kenmerken, namelijk: een kleur en een vorm. Voor het opvragen van de kleur van een steen maakt Stone gebruik van de enum *StoneColor* en voor de vorm van *StoneShape*.

Om het bord zo overzichtelijk mogelijk te houden, leek het ons praktisch om een de vormen van een steen afgekort weer te geven. Omdat er meerdere stenen zijn met de beginletter ‘C’, zou de eerste letter als afkorting voor verwarring zorgen. Daarom hebben we ervoor gekozen om de vorm van een steen weer te geven aan de hand van een cijfer. Zo wordt de vorm *Circle* weergegeven als een 1.

Wederom omwille van de overzichtelijkheid van het bord wordt ook de kleur van een steen afgekort weergegeven. Hier wordt voor de afkorting wel gebruik gemaakt van de eerste letter van de kleur. Zo wordt de kleur *Yellow* weergegeven als ‘Y’.

## 2.16 StoneType

*StoneType* representeert het type van een bepaalde steen in het Qwirkle spel. Omdat er sprake is van een vooraf gedefinieerde set van types, is *StoneType* een enum type.

## 2.17 StoneColor

*StoneColor* representeert de kleur van een bepaalde steen in het Qwirkle spel. Omdat er sprake is van een vooraf gedefinieerde set van kleuren, is *StoneColor* een enum type.

## 2.18 View

De interface *View* is verantwoordelijk voor de visuele weergave van het Qwirkle spel. Zo wordt er binnen deze interface een TUI gecreëerd en bevat zij een methode om de TUI bij te werken. De daadwerkelijke implementatie van de view vindt plaats in *TUIView* (zie sectie 2.24).

Binnen het *Observer* patroon vervult View de rol van *Observer*. Als *Observer* kan View bijhouden of de dimensies van het Qwirkle bord veranderd zijn en welke stenen er op het bord zijn geplaatst. Dit is natuurlijk wel zo fijn als je het Qwirkle spel aan het spelen bent.

## 2.19 ComputerPlayer

*ComputerPlayer* representeert een computer speler in het Qwirkle spel en is een uitbreiding van *OpenHandPlayer*. Zoals eerder beschreven is een computer speler een speler waarvan de inhoud van de hand publiekelijk mag worden weergegeven.

Een computer speler zal altijd gebruik maken van een bepaalde strategie, bijvoorbeeld een naïeve strategie. In de constructor van *ComputerPlayer* wordt de soort strategie als argument meegegeven.

De methode makeMove(Board board) instrueert de computer speler om een zet te doen. Binnen deze methode maakt *ComputerPlayer* gebruik van Board; het bord waarop de speler een zet moet doen, wordt als argument meegegeven.

## 2.20 HumanPlayer

De klasse *HumanPlayer* representeert een ‘menselijke’ speler in het Qwirkle spel en is net zoals *ComputerPlayer* een uitbreiding van *OpenHandPlayer*. Ook bij deze speler mag de inhoud van de hand publiekelijk worden weergegeven.

Het mag duidelijk zijn dat er, in tegenstelling tot *ComputerPlayer*, binnen *HumanPlayer* geen gebruik wordt gemaakt van een strategie. Een menselijke speler zal het spel zelf moet spelen.

In de methode *makeMove(Board board)* wordt gebruik gemaakt van de *notifyObservers* om de observers zoals de View te laten weten dat de gebruiker een zet moet doen. Vervolgens wordt de huidige Thread geblokkeerd totdat de zet gemaakt is. Wanneer één van de methoden *performTrade* of *placeStones* uitgevoerd is, zal *makeMove* stoppen met wachten en de executie van de Thread voortzetten.

## 2.21 NaiveStrategy

*NaiveStrategy* representeert als uitbreiding van *Strategy* een naïeve strategie binnen het Qwirkle spel. *NaiveStrategy* is één van de strategieën die door *ComputerPlayer* als strategie gebruikt zou kunnen worden.

*NaiveStrategy* is verantwoordelijk voor instrueren van een computer speler om een zet te doen op het bord. Om een computer speler en het bord waarop de speler een zet moet doen als argument mee te kunnen geven maakt *NaiveStrategy* gebruik van de klassen *ComputerPlayer* en Board.

Verder is *NaiveStrategy* verantwoordelijk voor het bepalen van de mogelijke zetten. Ook hier wordt er gebruik gemaakt van *ComputerPlayer* en Board om een computer speler en het bord waarop de speler een zet moet doen als argument mee te kunnen geven. Bij het bepalen van de mogelijke zetten wordt er een lijst van zetten opgeleverd. Om dit te kunnen realiseren maakt NaiveStrategy gebruik van de klasse Move.

## 2.22 QwirkleClient

*QwirkleClient* representeert een cliënt die met de server verbonden is. Het zorgt ervoor dat alle commando’s van de server worden geïnterpreteerd en uitgevoerd. Daarnaast zorgt het er ook voor dat zetten die de speler doet, terug worden gestuurd naar de server. Op deze manier blijft het spel gesynchroniseerd op zowel de server als de lokale computer.

Deze klasse is tevens ook meteen de controller van de Java applicatie die een speler zal gebruiken om het spel te spelen. De gebruiker kan via opdrachtprompt aangeven met welke server de cliënt moet verbinden en welk poortnummer gebruikt moet worden.

Omdat *QwirkleClient* de controller is, is de klasse verantwoordelijk voor de inlichting van de view met updates uit het model. Een voorbeeld is de plaatsing van een aantal stenen op het bord, of een notificatie dat de speler aan de beurt is.

## 2.23 Strategy

De interface *Strategy* representeert een strategie die gebruikt wordt door een computer speler. Deze interface zou beschouwd kunnen worden als de overkoepelende factor van alle strategieën.

*Strategy* is verantwoordelijk voor het uitvoeren van een zet volgens de strategie die op dat moment gebruikt wordt. Voor deze verantwoordelijkheid maakt *Strategy* gebruik van *ComputerPlayer* en Board om een computer speler en het bord waarop de speler een zet moet doen als argument mee te kunnen geven.

## 2.24 TUIView

*TUIView* is een implementatie van de interface *View* die de standaard I/O gebruikt om veranderingen in het spel te visualiseren, en om acties die een speler maakt, door te sturen naar de controller.

## 2.25 ClientHandler

De *QwirkleServer* klasse creëert voor elke speler die verbonden is een *ClientHandler*. Deze klasse zorgt ervoor dat alle commando’s die een cliënt naar de server verstuurt, worden geïnterpreteerd en uitgevoerd. Daarnaast bevat het ook methoden om de cliënt instructies te geven, zoals het melden dat de speler aan de beurt is, of dat bepaalde stenen op het bord zijn gelegd. De *ClientHandler* speelt dus een grote rol in de synchronisatie van het spel.

## 2.26 ClientPlayer

Elke *ClientHandler* die een spel begint, wordt gekoppeld met een *ClientPlayer*. De *ClientPlayer* klasse representeert een speler in een Qwirkle spel die instructies via het netwerk krijgt. De methode *makeMove* instrueert daarom ook de *ClientHandler* om een move request te versturen en blokkeert de huidige thread totdat de cliënt een zet heeft gedaan. Wanneer de *ClientHandler* een antwoord krijgt van de speler, wordt deze zet geïnterpreteerd en wordt de executie van de geblokkeerde thread voortgezet.

## 2.27 GameRequest

Een *GameRequest* is een klasse die een soort game-lobby representeert. Wanneer een speler een spel aanvraagt met een aantal spelers, wordt dit aantal opgeslagen in een *GameRequest* instantie. Spelers kunnen deze lobby toetreden als ze met hetzelfde aantal spelers willen spelen.

## 2.28 QwirkleServer

De *QwirkleServer* representeert de centrale game server. Het is verantwoordelijk voor het tot stand brengen van de verbinding tussen spelers, en speelt een grote rol in de synchronisatie van de spellen die gespeeld worden.

De server biedt mogelijkheden om een spel aan te vragen door middel van de *requestGame* methode. Gebruik makend van de *GameRequest* klasse, zorgt de server ervoor dat de openstaande verzoeken worden bijgehouden en bijgewerkt. Als er genoeg cliënten zijn verbonden die hetzelfde verzoek hebben, wordt er een spel gestart met deze cliënten. Hierna wordt de *GameRequest* niet meer gebruikt.

De server is ook verantwoordelijk voor het bijhouden welke spellen er op het moment worden gespeeld, en wie in elk spel zit. Ook biedt *QwirkleServer* methoden die ervoor zorgen dat bepaalde berichten worden uitgezonden naar alle cliënten in een bepaalde game. Met behulp van de *ClientHandler* klasse worden alle spelers op de hoogte gehouden van veranderingen in het spel en krijgen ze te weten of ze aan de beurt zijn of niet.

## 

# 

# 3 Test Report

## 3.1 Unit Testing

Een groot deel van onze classes hebben we in isolatie kunnen testen met behulp van JUnit. De tests die gebruik maken van JUnit focussen zich vooral op classes die uit het model komen, omdat die tests gemakkelijk te automatiseren zijn. Op deze manier halen we 82,5% coverage voor de *qwirkle* package, wat 56,7% van het hele project is.

### 3.1.1 LocationTest

De LocationTest test de Location klasse. Dit is een relatief simpele klasse. Daarom hoeft er ook niet heel veel getest te worden. Er wordt op de volgende delen gelet:

* Correctheid van de opslag van de coördinaten.
* De gelijkheid van coördinaten.
* Correctheid van de rekenkundige operaties *add* en *subtract*
* Het kopiëren van de coördinaten.

Er wordt hiermee 73,5% code coverage behaald op de Location class.

### 3.1.2 RectangleTest

De RectangleTest test de Rectangle klasse. Er wordt op de volgende delen gelet:

* Correctheid van de top-left en bottom-right coördinaten.
* Correctheid van de breedte en hoogte berekening.
* De gelijkheid van vierhoeken.
* Het kopiëren van de vierhoek.
* Het vergroten van de vierhoek.
* Het bekijken of een bepaald punt in de vierhoek bevindt of niet.

Er wordt hiermee 84,6% code coverage behaald op de Rectangle class.

### 3.1.3 StoneTest

De StoneTest bekijkt of de Stone klasse naar behoren werkt. Er wordt gelet op de volgende onderdelen:

* Opslag van kleur en vorm.
* De gelijkheid van stenen.
* Het converteren van vorm en kleur naar getal en terug. Dit wordt gebruikt door het netwerk protocol.
* De *toString* methode.

Op deze manier wordt er een code coverage van 100% op de Stone class behaald.

### 3.1.4 MoveTest

De MoveTest test de functionaliteit van de Move klasse. Er wordt gelet op de volgende onderdelen:

* De steen die gelegd wordt.
* De locatie van de steen op het bord.
* De gelijkheid van een zet.
* De toString methode.

Er wordt een code coverage van 76,9% behaald op de Move class.

### 3.1.5 RemotePlayerTest

De RemotePlayerTest test een deel van de klasse Player en RemotePlayer. Er wordt op de volgende onderdelen gelet:

* De hoeveelheid stenen die de speler heeft.
* De score van de speler.

De code coverage van de Player klasse bedraagt 100%. Echter, hoewel we de RemotePlayer klasse geheel automatisch kunnen testen zonder een compleet spel te starten, haalt deze test maar een magere 21,2% op code coverage in deze klasse. Dit zal later gecompenseerd worden in de systeemtest.

### 3.1.6 GameTest

De GameTest bekijkt de klasse Game van zo veel mogelijk kanten. Er wordt op de volgende zaken gelet:

* Het verkrijgen van alle spelers.
* Het verkrijgen van een speler bij zijn of haar naam.
* Het verkrijgen van de speler aan zet.
* De bepaling of het spel afgelopen is.
* De bepaling of een speler gewonnen heeft.

Merk op dat de *run* methode niet wordt getest in deze test. Dit is omdat deze methode veel afhangt van hoe de spelers zijn geïmplementeerd en heeft niet zoveel te maken met de game klasse zelf.

Op deze manier halen we een code coverage van 83,3%.

### 3.1.7 ArrayBoardTest

Met behulp ArrayBoardTest testen we de correctheid van de klasse ArrayBoard. De volgende elementen worden getest:

* Het kopiëren van het board met de methode *deepCopy*
* Het plaatsen van stenen met de methode *placeStone.*
* Het bekijken van al bezette velden.
* Het pakken van stenen van de stapel met de methoden *pickStone, pickStones*, *canPickStone* en *canPickStones.*
* Het verkrijgen van de dimensies van het bord met de methode *getDimensions.*

Merk op dat de methode *checkMove* niet in deze test wordt behandeld. Dit is omdat *checkMove* onafhankelijk is van de implementatie van het board. ArrayBoardTest doelt vooral op het testen van de integriteit en correctheid van de array die het ArrayBoard behandeld, en heeft niets te maken met het verifiëren van moves.

Met behulp van deze tests behalen we 100% code coverage in de ArrayBoard klasse.

### 3.1.8 SequenceTest

De SequenceTest kijkt of de klasse Sequence goed functioneert. Er wordt op de volgende delen gelet:

* De detectie van de richting van een rij.
* De verzameling van de stenen in een rij.
* De geldigheid van een rij.

Met deze test wordt een coverage van 100% in de Sequence klasse behaalt.

### 3.1.9 CheckMoveTest

De CheckMoveTest test of het verifiëren van een zet met *Board.checkMove* naar behoren werkt. Door verschillende situaties van het bord te simuleren, kunnen we verifiëren of bepaalde zetten geldig zijn of niet. Er wordt gelet op de volgende situaties:

* Het plaatsen van een enkele steen op een leeg bord.
* Het plaatsen van meerdere stenen in horizontale of verticale richting.
* Het verhinderen van het plaatsen op een al bezet veld.
* Het verhinderen van het plaatsen van een steen naast een rij met al 6 stenen.
* Het verhinderen van het plaatsen van een steen naast een rij die al een dergelijke steen bevat.
* Het plaatsen van een steen die rijtjes met elkaar verbindt, zowel in horizontale als verticale richting.

Omdat de methode Board.checkMove de klasse veel van de klasse Sequence gebruik maakt, wordt er ook veel van deze klasse meegenomen.

Deze test behaalt 100% van de checkMove methode, en 100% van de Sequence.isValid methode die verantwoordelijk is voor het bekijken of een rij geldig is of niet.

### 3.1.10 ScoreTest

Deze test simuleert, net zoals de CheckMoveTest, een aantal situaties van het bord om van bepaalde zetten de score te berekenen. Er wordt gelet op de volgende situaties:

* Het plaatsen van een enkele steen op een leeg bord.
* Het plaatsen van meerdere stenen in horizontale of verticale richting.
* Het verlengen van een rij.
* Het verbinden van twee of meerdere rijen.
* Het verkrijgen van de qwirkle-bonus.

Deze test behaalt 100% van de *calculateScore* methode, en gebruikt ook 49% van de Sequence klasse.

### 3.1.12 Samenvatting

Wanneer we alle unit tests uitvoeren, behalen we de volgende percentages code coverage in de qwirkle package:

|  |  |
| --- | --- |
| **Class** | **Code coverage** |
| ArrayBoard | 100% |
| Player | 100% |
| ScoreComparator | 100% |
| Sequence | 99,2% |
| Board | 97,3% |
| StoneColor | 94,4% |
| StoneShape | 94,4% |
| SequenceDirection | 91,7% |
| Location | 87,6% |
| Rectangle | 84,6% |
| Game | 83,3% |
| Move | 76,9% |
| Stone | 74,0% |
| SequenceType | 65,0% |
| RemotePlayer | 21,2% |
| OpenHandPlayer | 15,7% |
| MoveResult | 0,0% |
| **Totaal:** | **83,5%** |

## 3.2 Systeemtest

Niet alle tests kunnen worden geautomatiseerd met JUnit. Veel onderdelen, zoals de communicatie tussen de view en controller, kunnen alleen getest worden met de hand.

### 3.2.1 Testen van de TUI

Deze test doelt op het verifiëren van de functionaliteit van de TUIView class. Het gaat om de volgende onderdelen:

* De visualisatie van het bord.
* De visualisatie van de stenen in de hand.
* De hoeveelheid stenen en score die de tegenspelers hebben.
* Het invoeren van commando’s (vooral het afvangen van verkeerd ingevoerde commando’s).

Vooral in het laatste punt is veel tijd gaan steken. Omdat de gebruiker al gauw een verkeerd commando invult, zoals het invullen van een letter in plaats van een getal, of het verkeerd typen van een instructie, moeten er veel checks worden uitgevoerd.

### 3.2.2 Testen van het netwerk protocol

Deze test gaat om het testen van de klassen QwirkleClient, QwirkleServer en ClientHandler. Tijdens één van de werkcolleges werd er een toernooi gehouden. Ook al was dit toernooi vooral berust op het toetsen van de AI, het bleek ook meteen een goede test te zijn voor het communicatiegedeelte van de applicatie. Omdat de andere groepen ook een server hadden gebouwd volgens hetzelfde protocol, konden we goed testen of de implementatie van het protocol wel voldoende was. Op deze manier hebben we veel bugs weten te verwijderen.

### 3.2.3 Testen van de AI

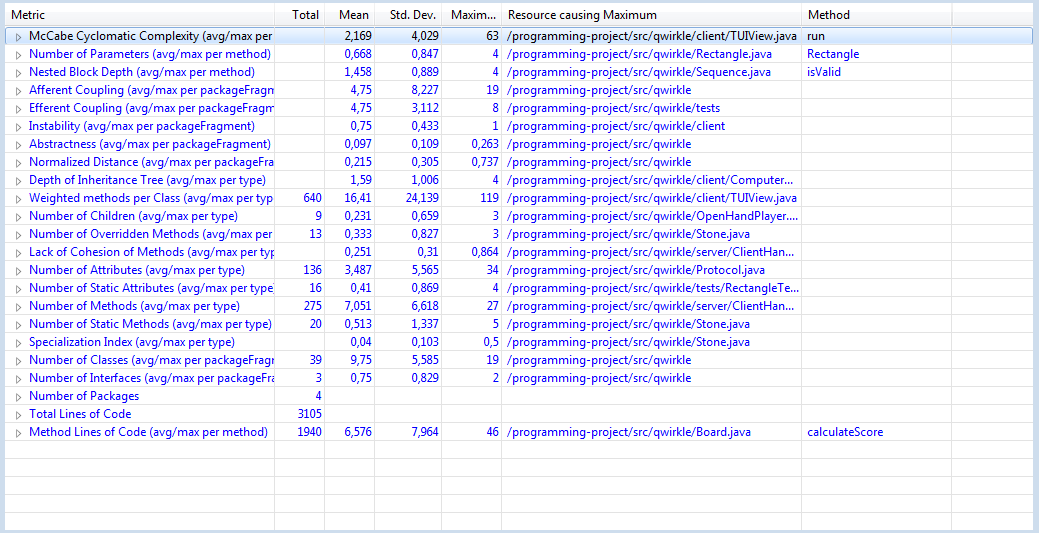
Deze test doelt op het verifiëren van de kracht van de verschillende strategieën die een computerspeler kan gebruiken om het spel te spelen. Het gaat om de volgende onderdelen:

* Het correct plaatsen van stenen op het bord.
* Voor de NaiveStrategy: Het vinden van een willekeurige zet
* Voor de SmartStrategy: Het vinden van de zet met de meeste punten.
* Het ruilen van stenen als geen zet kan worden gemaakt.

We hebben de twee computerspelers een aantal keer tegen elkaar laten spelen. Als men twee naïeve of twee slimme computerspelers tegen elkaar spelen, liggen de scores dicht bij elkaar. Echter, als een slimme computer tegen de naïeve computer speelt, wint de slimme computer vrijwel altijd, zoals in de volgende tabel is beschreven:

|  |  |  |
| --- | --- | --- |
| **Spel** | **Score SmartStrategy** | **Score NaiveStrategy** |
| 1 | 302 | 265 |
| 2 | 234 | 142 |
| 3 | 234 | 148 |

# 4 Metrics Report



Om de complexiteit van onze klassen te bepalen, kunnen we onder andere gebruik maken van de metric ‘Weighted Methods per Class’ (WMC). Deze metric is de som van de complexiteit van de methoden die in een klasse zijn gedefinieerd. Daarmee vertegenwoordigt het ook de complexiteit van een klasse als geheel.

Als we naar het resultaat van de WMC van ons project kijken, zien we een gemiddeld resultaat van 16,41. Gezien het feit dat een bovengrens van 50 voor WMC een goed referentiepunt is voor de meeste projecten kunnen we stellen dat de code van ons project relatief eenvoudig is.

De TUIView heeft een WMC waarde van 119 en met een bovengrens van 50 voor WMC is dit behoorlijk hoog. Zo’n hoge waarde betekent dat de TUIView in twee of meer klassen had kunnen/moeten worden opgedeeld.

Verder zijn er een nog een aantal andere resultaten die opvallen. Zo is de McCabe Cyclomatic Complexity waarde van de klassen TUIView, QwirkleClient, Stone en Board hoog. Klassen met hoge McCabe waarden zijn vaak moeilijk te begrijpen en hebben een grote kans op gebreken. Hieruit kunnen we concluderen dat de klassen TUIView, QwirkleClient, Stone en Board van redelijk slechte kwaliteit zijn.

Het cyclomatic complexiteit getal geeft ook het aantal testgevallen weer die moeten worden geschreven om alle paden in een programma uit te voeren. Aangezien er voor de eerdergenoemde klassen dus uitgebreide testen moeten worden geschreven en de kans dat er een testgeval vergeten wordt groot is, kan het testen van deze klassen erg complex zijn.

# 5 Known issues

Bij het plaatsen van een reeks stenen (steen 2 en 3) wordt in de checkMove methode gecontroleerd of de eerste steen van deze reeks grenst aan een steen die al op het bord ligt (steen 1). Bij het tournament zagen wij dat er ook een aantal groepjes waren waarbij eerst de buitenste steen (3) van een reeks op het bord werd geplaatst.

We hebben ervoor gekozen om onze checkMove niet aan te passen, omdat je bij het fysiek spelen van Qwirkle ook vrijwel altijd eerst de aangrenzende steen zult plaatsen.



Een van de requirements voor de server is: “Wanneer de server wordt gestart, moet er een poortnummer ingevuld worden waarnaar de server zal luisteren.” In plaats van het poortnummer in de TUI in te vullen, hebben we ervoor gekozen om het portnummer (en ook de host) als programma argument mee te geven. Dit maakt het namelijk een stuk gemakkelijker om de server te testen.

Zoals beschreven staat in de module handleiding zouden de drie meest complexe klassen gedocumenteerd moeten worden in JML. Bij ons was TUIView een van de meest complexe klassen. Omdat de TUIView voornamelijk bestaat uit methoden die output genereren, leek het ons niet interessant om JML aan deze klasse toe te voegen.

Tot slot hebben we de requirement “*de bedenktijd van de computer speler moet een parameter zijn die aangepast kan worden via de client TUI*” helaas niet kunnen implementeren wegens tijdgebrek.

# 

# 

# 6 Reflection on Planning

Onze planning kwam voor ongeveer 50% overeen met de daadwerkelijke progressie tijdens de project weken.

Designing

In de eerste week hebben we ons met name gericht op het design van het project. Tijdens deze fase correspondeerde onze progressie volledig met de planning.

Implementing

De implementeerfase heeft voor veel veranderingen binnen onze planning gezorgd. Zo zijn we bijvoorbeeld eerder dan gepland begonnen met het implementeren van de netwerk klassen en bleek ook het implementeren van de andere klassen niet goed ingepland te zijn. Dit heeft er uiteindelijk voor gezorgd dat we alle klassen ‘door elkaar’ aan het implementeren waren en onze voortgang in geen enkel opzicht meer met de planning correspondeerde.

Documenting code

Aanvankelijk leek het ons handig om javadoc en JML aan alle klassen toe te voegen voordat we de klassen ook daadwerkelijk gingen implementeren. Op deze manier zouden we een duidelijk overzicht krijgen van wat de klassen zouden moeten doen. Voor deze taak hadden we dan ook maar twee dagen ingepland. Achteraf gezien zijn we eigenlijk voortdurend bezig geweest met het toevoegen van documentatie. Dit kwam doordat er in de loop van het project regelmatig nieuwe klassen en methoden toegevoegd/verwijderd werden.

Testing

Doordat de implementatie van het project de nodige vertraging had opgelopen, correspondeerde het testen ook niet met onze planning. Zo hebben we het testen grotendeels uit moeten stellen tot de laatste week van het project.

Writing the report

Het schrijven van het verslag stond verspreid over de gehele project periode ingepland; dit om te voorkomen dat we niet op het laatste moment nog het hele verslag moesten schrijven. Op dit gebied is het ons gelukt om ons aan de planning te houden, al moest er tijdens de laatste dagen van het project toch nog redelijk veel typewerk worden gedaan.

Om onze afwijking van de planning te compenseren hebben we regelmatig met elkaar besproken wat er nog moest gebeuren om het project voor de deadline af te kunnen krijgen. Dit leek ons handiger dan het aanpassen van de planning, omdat we die tijd beter in de implementatie van het project konden steken.

Twee do’s voor de studenten van volgend jaar:

1. Begin zo snel mogelijk met de implementatie van het project.
2. Werk naast de ingeroosterde projecturen ook in je vrije tijd aan het project.

Twee don’ts voor de studenten van volgend jaar:

1. Onderschat het schrijven van het verslag niet.
2. Wacht niet te lang met het implementeren van de netwerk klassen.