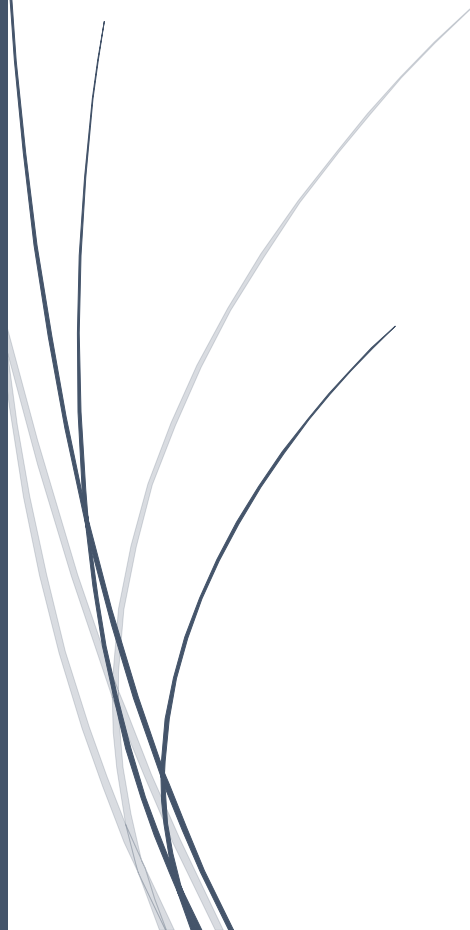




29-1-2016

Qwirkle

De ontwikkeling van Qwirkle in Java



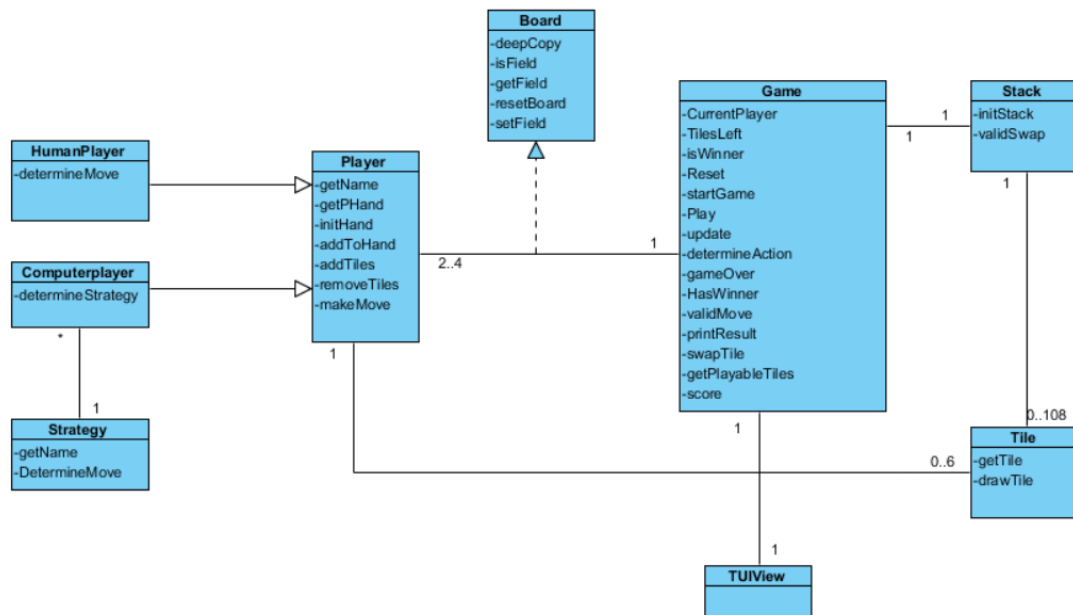
Lieke Hamelers(s1741640), Nick Kerckhoffs(s1742086)
BUSINESS&IT

Inhoud

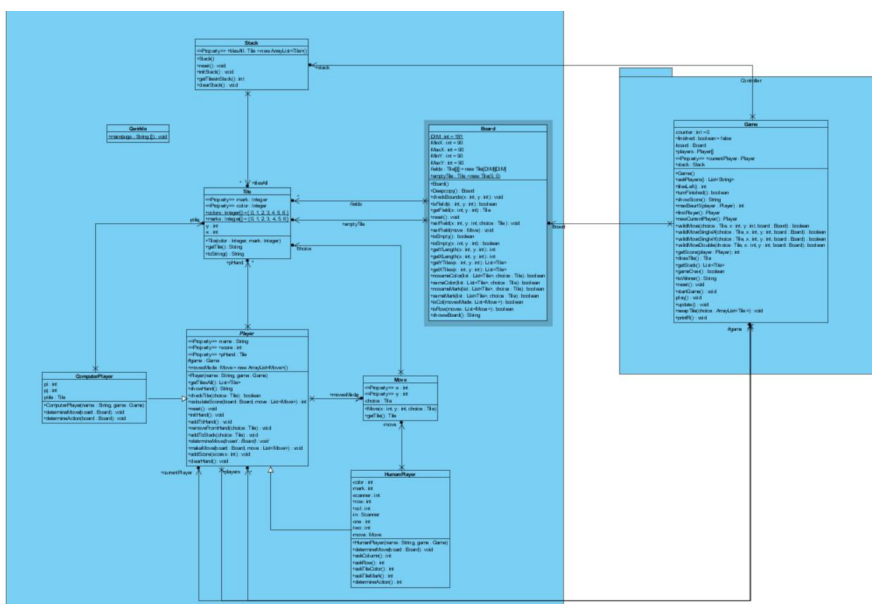
Discussion of the Overall design	2
Class Diagrams	2
Systematisch overzicht van de applicatie	3
Discussion per Class:	5
Board:.....	5
Move:	5
Tile:.....	5
Stack:.....	6
Player:	6
ComputerPlayer:	7
HumanPlayer:.....	7
Game:.....	8
Test Report:.....	9
Board:.....	9
Move:	9
Tile:.....	9
Stack:.....	10
Player:	10
ComputerPlayer:	10
Game:.....	10
Metrics report	12
McCabe Cyclomatic Complexity.....	12
Weighted methods per Class	12
Afferent Coupling.....	13
Efferent Coupling	13
Lack of cohesion of methods	13
Reflectie op de planning	14
Invloed van design planning	14
De planning	14
Correctheid van de planning.....	14
Maatregelen om planning te compenseren	15
Voor de volgende keer.....	15

Discussion of the Overall design

Class Diagrams



Aan het begin van het project hadden wij eerst een class diagram gemaakt om de structuur van de applicatie te bestuderen en bepalen. Tijdens het schrijven van de applicatie hebben wij ons daarom ook zo veel mogelijk hieraan gehouden. Het was in ieder geval duidelijk dat de class **Game** eigenlijk bijna alle classes zou verbinden en dit tot een werkend spel zou maken. Daarnaast was de class **Player** ook een belangrijke. Deze verbindt de class **HumanPlayer** en **ComputerPlayer** met de game. De class **Tile** was zowel verbonden aan **Player**, die deze nodig had om zijn hand ermee te vullen als aan de class **Stack** aangezien deze hem nodig had om zichzelf te vullen. Echter zijn wij tijdens het programmeren project tegen aardig wat dingen aangelopen, waardoor we moesten afwijken van dit model. Hierdoor zag het class diagram er zo uit na het project:



Dit class diagram hebben we laten maken van onze code door Visual Paradigm. Deze reflecteert dus precies de opbouw van onze code en de organisatie van de applicatie. Als eerste valt op aan het diagram dat er twee verschillende blauwe vakken zijn. Deze staan voor de controller en het model. De view is er niet bij betrokken aangezien deze leeg is en de TUI is ondergebracht bij board. In de controller zit alleen de game. In de game staan alle parameters en methoden die in de game zitten. Vanuit de game gaan er 3 pijlen naar het board, de stack en de player. Dit zijn alle koppelingen die game daarmee heeft. Dit komt omdat game de verbindende class is. Het heeft dus verschillende classes nodig maar geen andere heeft game nodig. Vanuit board, stack en player gaan alle pijlen naar de class Tile. Eigenlijk als je het diagram goed bekijkt gaan alle classes uiteindelijk naar Tile. Dit is op zich logisch aangezien wij de class Tile in elke class gebruiken. Daarom gebruikt elke class Tile, maar de class Tile is een op zichzelf staande class die geen enkele andere nodig heeft. Vandaar dat er alleen pijlen naar Tile gaan maar geen pijlen vanuit Tile naar andere klassen. Ook is in het diagram te zien dat de pijlen van de class ComputerPlayer en HumanPlayer naar Player een generalisatie zijn. Dit klopt ook aangezien deze twee classes de class Player extenden. De conclusie die we hieruit trekken is dat we redelijk ons idee hebben gevolgd dat de class Game de class is die alles verbindt. Ook zijn er een aantal verschillen te zien met de eerste Classdiagram. Zo zijn de classes Strategy en TUIView verdwenen. Dit hebben wij bewust gedaan. Voor de class ComputerPlayer is geen uitgebreide strategie, eigenlijk plaatst deze de eerste mogelijke steen. Hierdoor vonden wij het niet nodig om een aparte class Strategy te maken en hebben wij dit in de class ComputerPlayer zelf gezet. Ook verdween de TUIView op deze manier, wij hadden in de class Board al een functie gemaakt die het board uitprintte. Hierdoor werd de TUIView overbodig en vonden wij het niet nodig om deze in een aparte class te zetten. Voor de rest zijn we tevreden hoe we ons eerste Class diagram hebben aangehouden en hoe dit heeft uitgepakt.

Systematisch overzicht van de applicatie

Hierin wordt aangegeven welke deel van de vereisten door welke verschillende classes wordt geïmplementeerd. Dit is hieronder samengevat in de tabel.

Vereisten	Implementatie
De applicatie moet de functionele vereisten bevatten die zijn bediscussieerd.	Deze zijn niet geïmplementeerd. Wij hebben geen server/client gemaakt. Hieronder staat meer uitleg hierover.
De applicatie moet zonder problemen compileren.	Dit is gelukt door al onze lokale classes. Onze lokale game werkt en kan door zowel HumanPlayers als ComputerPlayers worden gespeeld.
Alle zelf-gedefinieerde classes moeten in minstens 3 verschillende packages zijn	De classes zijn verdeeld in 6 packages. De eerste 3 zijn de Model, Controller en View. Hierin zit al onze lokale classes en de TUIView. Daarnaast hebben wij een package gemaakt voor al onze JUnit test classes en hebben wij 1 package gemaakt voor onze server. De laatste package is voor onze exceptions.
De code moet een goede coding style hebben.	Dit hebben wij grotendeels proberen aan te houden. Tijdens het programmeren door te letten op namen van variabele, indeling van een class met behulp van commands en queries, en door de class continue te

	formatten. Aan het einde hebben wij nog een keer checkstyle over onze applicatie laten lopen en daarmee alle fouten eruit gehaald.
De implementatie moet gebruik maken van het Model-View-Controller patroon en het observer patroon.	Het observerpatroon hebben wij niet gebruikt aangezien wij geen server hebben gebouwd. Wij hebben tijdens het schrijven van het lokale spel wel het MVCpatroon gebruikt. Hierin is de class Game de controller, de class TUIView is de view en de rest van de classes zijn het model.
De waarde van een resultaat van een methode moet niet gebruikt worden om error-states te laten zien.	Dit hebben wij afgevangen door Exceptions te maken en gebruiken in onze applicatie. Wij hebben 5 exceptions zelf gemaakt en gebruikt. Hierdoor laat niet alleen de console zien dat er iets fout is maar stopt het proces ook.
Voor de 3 meest complexe classes, moeten de classes en hun methoden JML en Javadoc bevatten.	Al onze classes uit MVCmodel bevatten Javadoc. De 3 meest complexe classes zijn onze Player, Game en Board. Dit hebben we bepaald door de WMC per class te bekijken en de hoogste 3 te kiezen. Deze 3 classes bevatten dus ook JML.
De server moet multi-threaded zijn.	Wij hebben geen server gebouwd, dus is dit niet van toepassing op onze applicatie.
Er moeten testclasses en testruns zijn.	Wij hebben eerst tijdens het programmeren door middel van Main-methodes onze classes getest. Nadat een class af was, zijn er JUnit tests voor geschreven en gezorgd dat deze runnen en alle situaties afvangen.
Voor de 3 meest complexe classes moet de test coverage minstens 50% zijn.	Voor de class Player is dit: 99,4%. Voor de class Game is dit: 100%. Voor de class Board is dit: 95,4%. Ons doel was om dit boven de 95% te krijgen en dat is ook gelukt. Soms kregen we ze niet 100% omdat er toString-methoden inzaten die niet te testen waren of methodes die menselijke input vereisten. Wij hebben in onze test-package bepaalde classes gezet en aangepast dat deze wel te testen zijn zoals TestGame.

Wij hebben er uiteindelijk voor gekozen om niet te beginnen aan een server. Wij waren het weekend voor de deadline eindelijk klaar met de lokale game en de class ComputerPlayer. Wij hebben toen de twee dagen erop geprobeerd de server en client te bouwen maar kwamen tot de conclusie dat onze kennis ervan te weinig was en het heel veel tijd zou kosten om dit nog te leren. Daarom is er besloten om te focussen op de lokale game en het verslag. Hierdoor zijn het dan niet 3 semi-af onderdelen, maar 2 onderdelen die goed werken en volledig zijn. Hierdoor kunnen wij dus ook niet voldoen aan de functionele eisen van de server.

Discussion per Class:

Board:

De class Board maakt gebruik van de class Tile en Move.

Het Board zorgt ervoor dat er een board wordt gecreëerd met een grote van 181 bij 181. In eerste instantie zijn alle plaatsen op het board leeg. Alle plaatsen in het board hebben een corresponderende x en y coördinaat. Het board heeft ook een functie waarmee gecontroleerd kan worden of een x en y coördinaat bestaat op het board. Op het board kunnen tiles geplaatst worden door een setField methode, deze krijgt een x en y coördinaat mee en een tile. setField maakt de waarde van het gegeven x en y coördinaat, de meegegeven tile. Je kunt ook een tile op een coördinaat opvragen met de GetField methode. De coördinaten moeten wel op het board bestaan anders throwt de methode een FieldNotExistingException. Om te controleren het gehele board leeg is kan de methode isEmpty gebruikt worden. Indien er gecontroleerd moet worden of een specifiek coördinaat leeg is kan isEmpty aangeroepen worden met een x en y coördinaat.

Om makkelijker een validMove te controleren kan in de Board de methode getXTiles en getYtiles aangeroepen worden. Deze methodes geven een lijst van alle tiles die in een aangesloten lijn liggen aan de tile op het meegegeven x en y coördinaat. Met de methodes getXLength en getYlength kan je ook nog de lengte van de lijst van tiles oproepen.

De methodes SameColor, noSameColor, sameMark en noSameMark hebben ook als functie, het controleren van de validMove makkelijker te maken. Samecolor krijgt een lijst van tiles mee en controleer of alle tiles dezelfde kleur hebben. Het zelfde geldt voor SameMark, maar dan met de symbolen van de tiles. NoSameColor controleert in de meegeven lijst, alle tiles een andere kleur hebben. Hierbij werkt NoSameMark ook hetzelfde, maar dan met symbolen.

Voor het bereken van de score was het belangrijk om te weten of een neergelegde lijst van tiles in een rij lagen of in een column. Hiervoor zijn isRow en isCol. IsCol controleert of alle tiles in de lijst hetzelfde x coördinaat hebben en dus in een column liggen. IsRow doet hetzelfde maar dan kijkt hij naar de y coördinaten om te concluderen of het een rij is of niet.

Ten slotte print board het board nog uit in de methode showwboard. Showwboard laat niet het volledige board zien, maar slecht het deel waar tiles liggen. In het begin is het board 3 bij 3. Elke keer als er een tile geplaatst wordt kijkt de methode checkBounds of de x en y coördinaten van de geplaatste tile groter zijn dan de MaxBounds of kleiner zijn dan de MinBounds. Indien dit het geval is wordt het huidige MaxBound of MinBound vervangen en hierdoor wordt er een groter board uitgeprint dan in de vorige beurt.

Board maakt deel uit van de Model in de Model-View-Controller.

Move:

De class Move maakt gebruik van de Tile class.

De class Move creëert een nieuwe move met een meegegeven x en y coördinaat en een tile. De inhoud van de Move kan opgeroepen worden met de methodes: GetX, GetY en GetTile. Move maakt deel uit van de Model in de Model-View-Controller.

Tile:

Tile gebruikt geen andere classes.

De class Tile creëert een nieuwe tile met twee meegegeven integers. De integers representeren een kleur en een symbool. De getallen lopen van 0 tot en met 6. 0 is een lege tile, alles erboven is een kleur of symbool.

Met de methodes GetMark en GetColor, kan het symbool en de kleur van een tile opgeroepen worden. De tile kan ook als een string opgeroepen worden door de functie toString te gebruiken.

Tile maakt deel uit van het Model in de Model-View-Controller.

Stack:

De class Stack maakt alleen gebruik van de class Tile.

Stack creëert een lege stack in zijn constructor. De lege stack kan vervolgens gevuld worden door de reset methode, die de stack vult met 3 keer elke unieke steen. Dit komt neer op een totaal van 108 stenen in de stack. Reset maakt gebruik van initStack die voor elke kleur samen met elk symbool, 3 tiles maakt en deze in de TilesAll stopt van de stack. Tiles al is een lijst met alle tiles die in de stack zitten. Alle tiles uit deze lijst kunnen opgeroepen worden met de getTilesAll methode, die de volledige lijst terug geeft als resultaat. De grote van de lijst kan opgeroepen worden met de GetTilesinStack methode, die dus de hoeveel tiles terug geeft die nog in de tilesAll zitten.

Voor het testen was het handig om een stack in één keer te kunnen legen en dit wordt gedaan in de clearStack methode. Die de tilesAll volledig leeg maakt.

Stack maakt deel uit van het Model in de Model-View-Controller.

Player:

De class Player maakt gebruik van de classes Tile, Move en om de class board om een tile te kunnen plaatsen op het board.

De abstracte class Player maakt een nieuwe speler aan. Aan de constructor moet een naam en een game meegegeven worden. De speler krijgt dan de meegegeven naam en wordt gekoppeld aan de meegegeven game. De speler krijgt ook een legen lijst met tiles mee die hij in zijn hand kan houden en de score van de speler wordt op nul gezet.

De query's van deze class bestaan uit getName, waarmee de naam van de speler opgevraagd kan worden, getPHand waarmee de hand met tiles van de speler opgevraagd kan worden en een methode getTilesAll, die kijkt wat er nog in de stack zit. Ook kan je de score van de speler zien met de getScore methode. Om de spelers hand met tiles weer te geven op het board tijdens het spel is er de showHand methode die een string representatie geeft van alle tiles die de speler nog in zijn hand heeft met een | ertussen.

Om te controleren of een speler wel een tile in zijn heeft gedurende een beurt was het belangrijk om een checkTile methode te maken. Deze methode krijgt een tile meegegeven en dan wordt er gecontroleerd of de speler die de tile heeft opgegeven, deze tile daadwerkelijk in zijn hand zit. Indien dit het geval is, is het resultaat van checkTile true, anders is het false.

De laatste query is calculateScore deze methode rekent uit hoeveel punten de neergelegde move van de speler waard is. Aan het begin wordt er gekeken hoeveel tiles er neergeld zijn. Indien dit maar 1 steen is, is de lengte van de x rij plus 1 plus de lengte van de y rij plus 1 de totale score. De plus 1 moet gedaan worden omdat de methode getXlength en getYlength de lengte van de rij of column teruggeeft zonder de geplaatste tile.

Indien de Move groter is dan één tile maakt de methode is gebruik van isRow en isCol om te kijken of de stenen horizontaal of verticaal zijn neergelegd. Als ze horizontaal zijn neergelegd wordt eerst gekeken hoe lang de horizontale lijn is en vervolgens wordt voor elke steen in de neergelegde rij de verticale lengte bekeken. Als er een verticale rij is neergelegd werkt het proces hetzelfde, maar dan wordt er eerst naar de verticale lengte gekeken en daarna voor elke tile naar de horizontale lengte. Indien de lengte van een rij 5 is zonder de geplaatste tile wordt er een extra zes punten toegevoegd aan het totaal aantal punten van de Move. Want zodra de tile geplaatst zal worden, zal de rij zes stenen bevatten en hiervoor wordt een bonus zes punten toegekend aan de speler. Aan het einde van het berekenen van de score wordt het totaal aantal punten van de gemaakte Move gegeven. De class bevat ook een aantal commands. Met de methode wordt de hand van de speler opnieuw gevuld met tiles en zijn score wordt weer op nul gezet. De methode inithand vult de spelers hand met zes tiles die uit de stack van de game gehaald worden. De tiles zijn volledig random doordat de methode drawtile uit game gebruikt wordt, die een willekeurige tile terug geeft. Als er een beurt voorbij is, is er een mogelijkheid dat de speler nieuwe tiles wil pakken. Dit kan met addToHand, addToHand voegt stenen toe aan de speler zijn hand toe, maar alleen als de hand minder dan zes

stenen bevat en er nog tiles in de stack zijn om uit te delen. Een speler mag maximaal zes tiles in zijn hebben. Als een tile op het board geplaatst wordt, moet deze ook uit de hand van de speler verwijderd worden. `removeFromHand` zorgt hiervoor. Eerst wordt er gekeken of de tile die uitkozen is om te verwijderen wel daadwerkelijk in de hand van de speler zit en daarna wordt hij verwijderd. Een tile moet ook toegevoegd kunnen worden aan de stack in het geval van een swap. Dit wordt gedaan door `addToStack`. `addToStack` throwt een `TileNotExistingException` in het geval dat de keuze nul is. Als een speler een tile wil plaatsen wordt de abstracte methode `determineMove` gebruikt. Hij is abstract omdat zowel een `HumanPlayer` als een `ComputerPlayer` een tile kan plaatsen, maar beide doen het op een andere manier. `HumanPlayers` kiezen zelf waar ze hun tiles willen neerleggen, maar `ComputerPlayer` berekenen de beste plek. Bij de methode `determineMove` wordt een `FieldNotExistingException`, `ListEmptyException` gethrowt voor het geval een speler een tile wil plaatsen op een niet bestaande plaats of een lege lijst wil plaatsen.

Wanneer de speler zijn keuze gemaakt heeft en de tile daadwerkelijk wil gaan plaatsen wordt de `MakeMove` methode gebruikt. Deze controleert of alle Moves in de Move lijst wel geldig zijn met behulp van de `validMove` methode uit de Game class. Indien het een geldige lijst moves is wordt de steen geplaatst en krijgt de speler nieuwe tiles in zijn hand. Als het geen geldige move is wordt er: "Choice is not valid, try again to place a tile or swap/pass." Uitgeprint, waarna de speler opnieuw een keuze kan maken wat hij wil gaan doen. De lijst met moves mag geen nul zijn anders wordt er een `ListEmptyException` exception gethrowt.

Om punten toe te voegen aan de spelers zijn huidige score is er een methode `addScore` die een meegegeven integer punten toevoegt aan de score van de speler. De meegegeven moet groter zijn dan nul, want er kunnen geen negatieve punten toegekend worden.

Ten slotte hadden we voor de test nog een methode nodig die de hand van de speler helemaal leeg maakt, dit wordt gedaan door `clearHand`.

Player maakt deel uit van de Model in de Model-View-Controller.

ComputerPlayer:

De class `ComputerPlayer` extends de `Player` class en maakt gebruik van dezelfde classes als `Player`. In de constructor van de class `ComputerPlayer` wordt de super aangeroepen zodat er een speler wordt aangemaakt met een naam, game, een hand en een score. Een `ComputerPlayer` kiest een Move met behulp van de `determineAction` methode. Deze methode kijkt of de computerPlayer een tile in zijn hand heeft die hij op het board zou kunnen leggen. Zodra hij deze vindt slaat hij de x, y en tile van de Move op. Indien de `ComputerPlayer` geen tiles in zijn hand heeft die hij op het Board kan neerleggen swapt hij de volledige inhoud van zijn hand. De computer speler plaats een tile met de `determineMove` methode die gebruikt maakt van `MakeMove` uit de `Player` class om een Tile te plaatsen en de `determineMove` van `Player` override.

`ComputerPlayer` maakt deel uit van de Model in de Model-View-Controller.

HumanPlayer:

`HumanPlayer` extends net zoals `ComputerPlayer` de class `Player` en maakt ook gebruik van dezelfde classes als `Player`.

In de constructor van de class `HumanPlayer` wordt de super opgeroepen om een `Player` te maken met een naam, game, hand en score. `HumanPlayers` krijgen in hun constructor ook nog een Scanner mee.

De `HumanPlayer` heeft een `determineAction` class, deze vraagt wat de speler tijdens zijn beurt wil doen. Hij heeft de keuze uit: het plaatsen van een tile, tiles swappen, zijn beurt passen of om een hint vragen. De speler geeft antwoord in de vorm van een integer. Als de speler er voor kiest om een tile te plaatsen wordt `determineMove` gebruikt, die de `determineMove` van `Player` override.

`DetermineMove` roept vier aparte methodes op om de speler om de plaats van de tile te vragen, de kleur en het symbool. Als het een `validMove` is wordt de tile geplaatst, als het niet het geval is krijgt de speler een melding en kan hij het opnieuw proberen.

HumanPlayer maakt deel uit van de Model in de Model-View-Controller.

Game:

De class Player maakt gebruik van de classes: Tile, Move, Stack, Player en Board.

Game creëert in de constructor een nieuwe Game met een board, stack en spelers. De namen van de spelers moeten gegeven worden door de gebruikers van de Qwirkle applicatie. Indien de gebruiker een ComputerPlayer wilt aanmaken, moet hij de naam van de speler met een: "C", laten beginnen, anders wordt er een HumanPlayer aangemaakt. Er moeten minimaal 2 spelers aangemaakt worden en maximaal 4. Als de gebruiker een incorrect aantal spelers aanmaakt wordt er een IncorrectNoOfplayersException gethrowt. Als het aantal wel correct is, krijgt de gebruiker de optie of hij de instructies van het spel weer doorlezen of niet, daarna begint het spel echt.

Aan het begin reset de Game eerst alles. Het board wordt reset, de stack en de spelers. Hierna moet de Game uitrekenen wie de eerste beurt krijgt, dit gebeurt in de firstPlayer methode die gebruik maakt van de maxBeurt1. MaxBeurt1 rekent uit hoeveel punten een speler maximaal kan verdienen in zijn eerste beurt. De gene die de meeste punten kan verdienen in zijn eerste beurt mag beginnen met het spel. Als eerste print de game het board uit en de hand van de speler die aan de beurt is. Omdat het de eerste beurt is van het spel moet de speler een tile plaatsen en mag hij niet zijn beurt overslaan of tiles gaan swappen. Als hij dit wel doet print de game een bericht uit waarin staat dat het niet mag. Zodra de speler zijn keuze heeft gemaakt wordt deze verwerkt door de Game. Daarna wordt er gekeken of het spel gameOver is of niet en als de Game niet gameOver is wordt de currentPlayer de volgende speler. Dit wordt gedaan door de newCurrentPlayer methode. Als de currentPlayer verwisselt is wordt de applicatie geüpdatet, het beurt wordt dan uitgeprint met de nieuw geplaatste tiles, de score is geüpdatet en de hand voor de volgende speler wordt op het scherm weergegeven.

Als de speler een tile wil plaatsen wordt de class Player gebruikt om te kijken waar de speler zijn tile wilt neerleggen, hierbij wordt ook gebruik gemaakt van de methode validMove uit de Game.

ValidMove kijkt of een steen wel daadwerkelijk op het board geplaatst mag worden, door gebruik te maken van verschillende methodes uit board. Eerst kijkt validMove of een tile in een verlenging van een x rij of y rij ligt of beide. Daarna kijkt hij of in die rij alle tiles een andere kleur hebben en dezelfde vorm of allemaal dezelfde vorm. Als dit het geval is, is de move valid. Nadat de tile geplaatst is wordt drawTile aangeroepen die een random tile uit de stack haalt, de stack moet hiervoor wel tiles bevatten.

Als de speler een tile wil gaan swappen, wordt de swapTile methode gebruikt die(zolang de tile keuze van speler geen nul is) de tile verwijdert uit de hand van de speler en er een nieuwe voor in de plaats voor terug geeft.

Als de speler voor kiest om zijn beurt over te slaan gebeurt er niets alleen de currentPlayer wordt de volgende speler.

Om erachter te komen of de Game GameOver is kijkt te GameOver methode naar alle tiles in de handen van de spelers en naar alle tiles in de stack. Zolang 1 van die tiles nog een validMove is, is het spel niet GameOver, anders wel. Als het de Game GameOver is wordt de winnaar uitgerekend door de isWinner methode. isWinner kijkt aan het einde van het spel naar alle scores van de spelers en geeft de speler met de hoogste score terug in de vorm van een string. Daarna krijgt de speler tot slot nog de optie om het spel nog een keer te spelen of om de applicatie af te sluiten.

Game maakt deel uit de controller in de Model/Controller/View.

Test Report:

Om de classes goed te kunnen testen hebben we een aparte testpackage gemaakt waarin alle JUnit testen in staan en een test variant van de normale classes. Het verschil tussen de normale classes en de testclasses is dat je bij de testclasses geen waardes hoeft mee te geven om spelers aan te maken aan het begin van het spel. Bij het echte spel moet dit wel zodra de applicatie gestart wordt. Elke class wordt apart getest.

Board:

De boardTest maakt voordat hij een test uitvoert eerst een nieuw board aan en reset deze zodat elk field een emptytile bevat. In de eerste test wordt gekeken of de Deepcopy methode werkt door een deepCopy te maken van het board en daarna kijken of de inhoud van de deepCopy gelijk is aan de inhoud van het board waarvan de deepcopy gemaakt is. Daarna wordt Isfield getest door te vragen of bepaalde velden op het board bestaan. Als ze bestaan moet het resultaat true zijn, zo niet moet het false zijn. Het board is 181 bij 181, daarom testen we de randwaarden 0,0 ; 181,181 en het midden ervan 90,90. De Methode EmptyField wordt getest door op een paar plaatsen tiles neer te leggen en dan aan de methode te vragen of het field nog leeg is of niet. Als hij niet leeg is moet isEmpty gelijk zijn aan false anders moet het true terug geven.

De methode getX en getY Length worden getest door een rij tiles op het board te zetten en daarna eerst zelf de waarde te berekenen om vervolgens te kijken of deze waarde gelijk is aan wat de methode als resultaat heeft.

Om de methode SameColor en noSamecolor te testen worden er tiles aangemaakt met allemaal de zelfde kleur of verschillende kleuren en daarna laten we de methode controleren of de meegegeven tile in een rij ligt met allemaal verschillende gekleurde tiles of tiles met dezelfde kleur. Op dezelfde manier worden SameMark en noSameMark getest, maar dan met verschillende of dezelfde symbolen.

Bij TestCol en TestRow wordt er een rij tiles aangemaakt en daarna daarmee een aantal moves die dan in een Move lijst worden gestopt. Vervolgens kunnen de methodes isRow en isCol er op aangeroepen worden om te kijken of de lijst met moves in een rij worden gelegd of in een column. Board heeft een coverage van 95,2 %, wat ruim genoeg is.

Move:

Aan het begin van de Move worden er tiles aangemaakt en daarmee worden nieuwe Moves gemaakt voor de test. Het enige wat de Move class hoeft te kunnen is de x, y en tile van zijn move geven. Daarom zijn er drie testen geschreven elk roept een ander deel van de tile op en kijkt of deze gelijk is aan de waarde die de Move heeft gekregen in de constructor.

De MoveTest covered the methode Move 100%.

Tile:

Aan het begin van de tileTest worden er verschillende tiles aangemaakt met een kleur en een symbool. Net zoals Move heeft Tile maar drie methodes dus heeft de test ook 3 testen. De eerste kijkt of de toString methode van tile goed werkt, door op een tile de toString methode aan te roepen en deze te vergelijken met het resultaat de applicatie zou moeten geven. De andere twee testen zijn om te kijken of de kleur en het symbool goed worden opgeroepen uit een tile. Dit gebeurt op dezelfde manier als de toString. De methode wordt aangeroepen en er wordt gekeken of de kleur/symbool gelijk is aan de kleur/symbool die de tile daadwerkelijk heeft.

De TileTest covered de methode tile 100%.

Stack:

Aan het begin van de StackTest wordt er een nieuwe stack aangemaakt en deze wordt gevuld met 108 tiles. Om Stack te testen is er maar één test nodig. De StackSizeTest controleert of er aan het begin 108 tiles in de stack zitten en of een tile daadwerkelijk in de stack zit als deze aan de stack wordt toegevoegd.

De StackTest heeft een coverage percentage van 100% over de class Stack.

Player:

Aan het begin van de PlayerTest wordt er een Game aangemaakt met twee spelers. Er wordt ook een speler genaamd two aangemaakt. Om te testen of het aanroepen van de naam werkt, wordt vergeleken of het oproepen van de naam gelijk is aan two. De score van two wordt getest door te kijken of de begin score van two nul is en als hij 1 punt krijgt, dat zijn score dan gelijk is aan 1.

De hand van de speler wordt getest door een lege hand te vullen met verschillende tiles en daarna te kijken of de tiles in de hand zitten met behulp van de contains functie. De speler kan ook Moves doen en om kijken of de moves er ook echt voor zorgen dat er op het veld echt tiles gelegd worden, wordt er eerst een Move gedaan en daarna wordt de getField methode gebruikt om te kijken wat er op het veld ligt en als het goed is ligt de geplaatste tile op het gevraagde veld. Er is ook een StackEmptyTest waarbij de speler tiles plaatst en daarna tiles uit de stack wil pakken terwijl die leeg is. Er wordt in dit geval dus getest of de hand van de speler na de move geen nieuwe tiles bevat.

Als er op de speler een reset wordt aangeroepen moet hij een nieuwe hand krijgen en zijn score moet weer op nul gezet worden. Bij de resetTest krijgt de speler punten en een hand en na de Test wordt gekeken of hij een nieuwe hand met tiles heeft en dat zijn score weer op nul staat.

Voor het berekenen van de score zijn verschillende testen geschreven elk pakt een andere situatie.

Bij de eerst worden er 6 tiles op een rij worden gelegd, daarnaast is er een test om de punten te berekenen als er tiles alleen in een rij worden gelegd, alleen in een column of als er maar één steen wordt neergelegd. Bij alle score testen worden de tiles in een move lijst gezet, op voorhand hebben we al berekent hoeveel punten de move waard is. Door te vergelijken hoeveel punten de calculateScore geeft aan de Move en onze waarde konden we kijken of het werkte.

Tot slot wordt de ClearHand methode nog getest. Deze methode was alleen maar geschreven voor de test, maar er moest alsnog gekeken worden of hij werkte. ClearHandTest vult een hand met stenen en kijkt daarna of de hand wordt geleegd als de ClearHand methode er op aangeroepen word.

De PlayerTest heeft een coverage van 92,1% of de testPlayer class wat voldoende is.

ComputerPlayer:

De computerPlayerTest, test of de computer speler wel daadwerkelijk doet wat wij willen. Aan het begin van de test wordt een nieuwe game gestart en een computer speler aangemaakt, daarna wordt de game gereset. De ComputerPlayerTest test twee dingen, hij test of de computer speler de valid moves doet en of hij niet de niet valid moves doet. Bij beide gevallen tiles aangemaakt en daarmee worden nieuwe Moves gemaakt. Deze moves worden dan uitgevoerd door de computer en indien valid moet de tile geplaatst zijn, als het geen valid move is moet de computerspeler hem niet geplaatst hebben en moet hij nieuwe tiles gepakt hebben voor in zijn hand.

Met deze Test wordt de ComputerPlayer class 100% gecoverd.

Game:

Tot slot is er de GameTest, deze maakt een nieuwe game aan en reset deze. Hij maakt ook een HumanPlayer aan. In de eerste test wordt gekeken of de game constructor een juiste game aanmaakt, door spelers op te roepen van de zo juist gemaakte game. Als tweede wordt de

CurrentPlayer getest, door steeds de newCurrentPlayer aan te roepen en te kijken of de nieuwe currentPlayer de juiste speler is.

De ValidMove methode van de Game wordt getest door de players constant Moves te laten doen waarvan we op voorhand hebben gekeken of ze valid zijn of niet en dan te kijken of de validMove methode de juiste boolean teruggeeft.

De GameOver methode wordt getest door eerst de speler en de stak validMoves te geven in dit geval moet de test zeggen dat het spel niet game over is. Daarna moeten er geen valid moves meer in de stack zitten maar wel in de hand van de spelers, hier moet het resultaat van game over ook false zijn. Daarna moet het omgedraaide gebeuren, dus de stack heeft validMoves, maar de speler niet, ook in dit geval moet de game over test false zijn. Tot slot moet de stack en de hand van de speler geen valid moves meer bevat en dan moet het spel game over zijn.

Bij de Score test wordt gekeken of de score Methode de juiste score geeft van de juiste speler en in de WinnerTest wordt naar de score gekeken en moet de isWinnaar de juiste speler geven, dit is de speler met de hoogste hoeveelheid punten.

Tot slot bevat de game de methodes swapTile en drawTile. Deze worden ook allebei getest in de Gametest. De swaptile wordt getest door te kijken of de speler de zojuist geswapte tile niet meer in zijn hand heeft na een swap. Bij de drawTile wordt er één tile in de stack gestopt en als de speler dan een tile wil pakken moet dat deze tile dus zijn. Of dit ook echt zo is wordt getest.

De totale coverage van de test over game bevat 66,5%. Dit is lager dan onze andere testen, maar dit komt omdat er een aantal methodes niet getest kunnen worden, omdat er bij deze methodes menselijke invoer nodig is. Daarom zijn we tevreden met 66,5%

Metrics report

Tijdens deze module hebben we de volgende software metrics geleerd :

- McCabe Cyclomatic Complexity
- Weighted methods per Class
- Afferent Coupling
- Efferent Coupling
- Lack of cohesion of methods

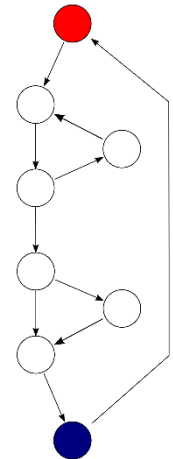
Ze zullen hieronder een voor een besproken worden en wat de waarden waren van de applicatie en wat dat betekent voor de kwaliteit van onze code.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
> McCabe Cyclomatic Complexity (avg/max per method)		3,009	4,5	35	/Final Project/src/Controller/Game.java	play
> Weighted methods per Class (avg/max per type)	653	21,065	36,32	147	/Final Project/src/qwirkleTest/testGame.java	
> Afferent Coupling (avg/max per packageFragment)		6,667	6,774	15	/Final Project/src/Model	
> Efferent Coupling (avg/max per packageFragment)		3,333	3,859	11	/Final Project/src/qwirkleTest	
> Lack of Cohesion of Methods (avg/max per type)		0,318	0,368	0,917	/Final Project/src/Model/Tile.java	

McCabe Cyclomatic Complexity

De McCabe Cyclomatic Complexity berekent de complexiteit van de code. Dit gebeurt aan de hand van het aantal paden die bestaan door de code. Bijvoorbeeld op het moment dat er in de code een conditie zit die waar kan zijn, of niet waar dan zijn er 2 mogelijkheden hoe het programma door kan gaan. Dit betekent dus dat er vanaf dan 2 verschillende paden zijn. De McCabe Cyclomatic Complexity wordt berekend door het aantal binaire keuzes te tellen en hier 1 bij op te tellen.

De waarde van de McCabe Cyclomatic Complexity in ons project is gemiddeld 3,009 per max method. Gemiddel genomen is dus onze applicatie redelijk makkelijk te onderhouden gezien de McCabe Cyclomatic Complexity. Echter als we wat specifiek naar de classes kijken zien we dat de class Game een gemiddelde heeft van 6,042. Dit betekent dat die class toch wat moeilijker is en daarmee ook moeilijk te onderhouden. We hadden hierin de methodes meer kunnen opsplitsen zodat ze simpeler bleven.



Weighted methods per Class

De Weighted methods per Class is alle opgetelde complexiteiten bij elkaar. Dit wordt vaak berekend door alle McCabe cyclomatic complexities bij elkaar op te tellen. Hierdoor kun je zien hoe ingewikkeld een class is en of je deze beter had kunnen opsplitsen.

De waarde van de Weighted methods per Class in onze applicatie is gemiddeld genomen 653. Hierin is de class Game weer een van de hoogste met 145. Dit is eigenlijk te hoog waardoor deze class te moeilijk wordt om te onderhouden. Het is moeilijk om een referentiewaarde te vinden maar een hoogte van 50 per class is toch wel ongeveer de maximum. De classes Board en Player zitten wel rond die hoogte en de rest van de classes zitten rond de 20. Deze zijn dus wel makkelijk bij te houden. Wij kunnen de class Game opsplitsen om de Weighted methods per Class om laag te brengen.

Afferent Coupling

De afferent coupling betekent eigenlijk hoe veel verantwoordelijkheid de package heeft. Dit wordt bepaald door hoeveel andere classes in andere packages afhankelijk zijn van de classes in de huidige package.

De waarde van de afferent coupling van onze applicatie is gemiddeld genomen 6,7. Deze wil je eigenlijk zo laag mogelijk houden zodat de mogelijkheid van het testen van classes zo simpel mogelijk blijft. Het kan dat bij ons de waarde zo hoog is omdat we ook de testClasses meenemen. Deze zijn allemaal afhankelijk van de classes in het Model package.

Efferent Coupling

De efferent coupling is precies het tegenovergestelde van afferent coupling. Het berekent van hoeveel classes in andere packages de huidige package afhankelijk is.

In onze applicatie is de waarde van efferent coupling gemiddeld genomen 3,33. Deze waarde wil je ook zo laag mogelijk houden. Nu is 3,33 niet extreem hoog maar het kan lager. Door deze waarde is de code moeilijker te onderhouden maar is hij ook een stuk gevoeliger voor verandering van een stuk code.

Lack of cohesion of methods

De Lack of cohesion of methods meet de cohesie van een bepaalde class. Een class heeft veel cohesie als veel methoden dezelfde fields gebruiken.

De Lack of cohesion of methods in onze applicatie heeft de gemiddelde waarde van 0,318. Dit is teveel. Ideaal zou het zijn om een Lack of cohesion of methods van 0 te hebben. Dit betekent dat de class veel cohesie heeft. De waarde moet dus zo dicht mogelijk bij 0 komen. Het is mogelijk om de waarde te verkleinen door verschillende classes te splitten en zo te zorgen dat elke kleinere class wel veel cohesie heeft.

De conclusie is dus dat onze applicatie wat betreft software metrics nog een stuk beter kon. Hierdoor zou de kwaliteit van onze code ook een stuk beter zijn en het onderhoud aan de code was een stuk makkelijker geweest. Op een paar onderdelen zaten we niet zo ver af van de gewenste waarde zoals de Lack of cohesion of methods en de cyclomatic complexity. Maar bij de rest van de onderdelen weken onze waarden toch wel echt te veel af van het gewenste en als wij extra tijd hadden, konden we dit nog aanpassen om de kwaliteit van onze code te verbeteren.

Reflectie op de planning

Invloed van design planning

Allereerst waren we door de ervaring met het plannen van het design al vroeg begonnen met een planning voor het project. We hadden al van veel studenten gehoord dat het enorm veel werk zou zijn dus we wilden op tijd beginnen en alle mogelijke tijd er in steken. Daarom hadden wij voor de kerstvakantie al een planning gemaakt om te beginnen aan het project. Het design project ging ons echter redelijk makkelijk af en we hadden genoeg tijd dus wat betreft de planning, konden we deze makkelijk volgen. Hierdoor hebben we ook niet echt valkuilen ontdekt tijdens het designproject wat we konden toepassen op de planning van het programmeerproject.

De planning

Datum	Onderdeel
20-12	Deadline tiles + test
02-01	Deadline board + test
06-01	Deadline game af
08-01	Deadline Player af
09-01	Testen + interface
10-01	Testen + interface
11-01	Server maken
12-01	Server maken
13-01	Server maken
14-01	Server af + testen
15-01	Uitloopdag
16-01	Verslag
17-01	Verslag
18-01	Verslag
19-01	Verslag
20-01	Verslag
21-01	Verslag
22-01	Strategy
23-01	Strategy
24-01	Strategy
25-01	Finishing touch
26-01	Finishing touch
27-01	Finishing touch

De afspraak erbij was dat wij om het andere onderwijs heen van 10 tot 5 uur aan het project zouden werken.

Correctheid van de planning

In het begin leek ons dit een goede planning. Tijdens de kerstvakantie zouden we al beginnen en dit was redelijk gelukt. Alleen zaten we met board nog op een paar puntjes vast. Hierdoor begonnen we al met uitloop in week 7. Wij hadden in week 7 en week 8 overschat hoeveel tijd we zouden hebben voor het project. Ondertussen was natuurlijk ook nog het wiskunde-tentamen en in week 9 het programmeertentamen. In week 7 moesten we ook nog de laatste programmeeropdrachten laten aftekenen. Hierdoor zijn wij in die weken minder aan het project toegekomen. Vooral in week 8, 9 en 10 zijn we veel aan het project toegekomen. Ook liep het in week 7 en 8 niet zo lekker omdat wij het moeilijk vonden waar we moesten beginnen en hoe we het allemaal moesten maken. Onze kennis van programmeren was niet zo veel dat we heel makkelijk konden beginnen dus hebben we

hier ook nog heel veel vertraging opgelopen. Dit betekende uiteindelijk dat wij in het weekend voor week 10 pas klaar waren met de lokale game.

De conclusie was dus dat de planning helemaal niet correct was en dat wij hadden overschat hoeveel tijd we hadden voor het project in het begin en hoe snel wij konden werken.

Maatregelen om planning te compenseren

Doordat onze planning al vrij snel niet meer klopte hebben wij heel mat maatregelen moeten nemen om te zorgen dat wij aan het einde toch een applicatie en verslag hadden om in te leveren.

De eerste was de beslissing om elke ochtend om kwart voor 9 te beginnen en tot half 6 door te werken en in de avonden zelf nog door te werken. Hierdoor hadden we meer tijd om in het project te steken. De tweede maatregel die wij namen was de beslissing om het project pas vrijdag in te leveren in plaats van woensdag. Hierdoor hadden we nog 2 dagen meer om aan het project te werken en het was voor ons vooral belangrijk om het te halen en richtte wij ons niet meer op het hoogste cijfer. Toen uiteindelijk al die maatregelen waren genomen kwamen we er in week 10 achter dat de server totaal niet lukte. Dus als laatste en grootste maatregel hebben wij besloten geen energie meer te steken in de client/server en ons te focussen op een mooie lokale game en een goed verslag. Hierdoor hebben wij onze uitgelopen planning gecompenseerd met als doel een mooi eindproduct.

Deze maatregelen hebben wel veel impact gehad op ons project en de kwaliteit ervan. We hebben namelijk geen server/client wat ons veel punten zal kosten. Echter pakte het positief uit op de kwaliteit van de lokale game want deze hebben we nog een stuk beter kunnen maken en het verslag nog een stukje uitgebreider.

Voor de volgende keer

Wat ik heb geleerd van deze planning tijdens het project is dat het belangrijk is om rekening te houden met de andere activiteiten tijdens deze weken en dit ook in te plannen. Hierdoor krijg je een veel realistischer beeld van hoeveel tijd je echt hebt tijdens deze weken. Ook is het handig om van tevoren uit te zoeken hoe snel je kan werken. In sommigen gevallen is dit niet mogelijk, echter soms ook wel. We hadden deze keer bijvoorbeeld eens kunnen timen hoe lang wij deden over het schrijven van een bepaalde class en kunnen toepassen op de rest van de planning. Wat ook handig kan zijn voor de volgende keer, dat als je weet dat er een moeilijk onderdeel aan komt, deze keer was dat dan de server/client, om dan in de rest van de weken de theorie al te lezen en wat dingen alvast te proberen. Dan kan je er veel sneller achter komen als het niet lukt en nog wat maatregelen instellen zodat het nog steeds lukt.

Ik heb nog wel een paar tips voor de studenten van volgend jaar.

Do's:

- Zoek een studentassistent die goed kan uitleggen en jou ook goed begrijpt en waar je vast zit. Wij konden met sommigen studentassistenten beter overleggen en daar snapten wij de uitleg ook sneller van. Dit verschilt heel erg per persoon maar ook per studie(BIT/TI).
- Als je al een tijd zit te staren op een fout in je code en kan hem niet vinden, gun jezelf dan even 5 minuten pauze of ga met iets anders verder. Als je dan terugkomt bij dat stukje code heb je weer een frisse kijk en heb je een grotere kans om de fout te vinden.
- Werk altijd met z'n tweeën samen aan de code. Hierdoor heb je altijd een partner om mee te sparren over je code en deze kan helpen bij dingen waar je niet uit komt en andersom. Apart van elkaar classes bouwen kost uiteindelijk meer tijd en als je vast zit kan je ook niet verder.

Don'ts:

- Maak niet al je testclasses aan het einde als je alle classes al hebt gebouwd. Een hele dag tests schrijven is en heel saai en als je dan een fout tegenkomt is het vaak veel moeilijker om die eruit te halen omdat bijvoorbeeld andere methodes die methode gebruiken.
- Als je probeert iets te maken en het lukt niet, verwijt het dan niet jezelf. Dit levert alleen veel stress en boosheid op. Realiseer je dan op zo'n moment hoeveel je al kan en hebt geleerd door het project en zorg ervoor dat je hulp krijgt van iemand erbij.

De conclusie is dat uiteindelijk onze planning ontzettend is uitgelopen en eigenlijk aan het einde niet meer klopte, maar doordat we veel maatregelen hebben genomen is het ons toch gelukt een werkende game te maken en een mooi eindproduct in te leveren. Wij zijn ook heel blij met hoeveel we hebben geleerd tijdens dit project en wat we nu allemaal kunnen vergeleken met het begin van deze module.