KEYBARRICADE

Ontwikkeldocument

Aarnoutse, M.F.A.; Dufour, J.E.R.; Vermeij, E.H.J.
15044688, 15036278, 14101327

Begeleider: Okan Zor
Klas: SE-OO-2
Groep: 5

INHOUDSOPGAVE

1. INLEIDING

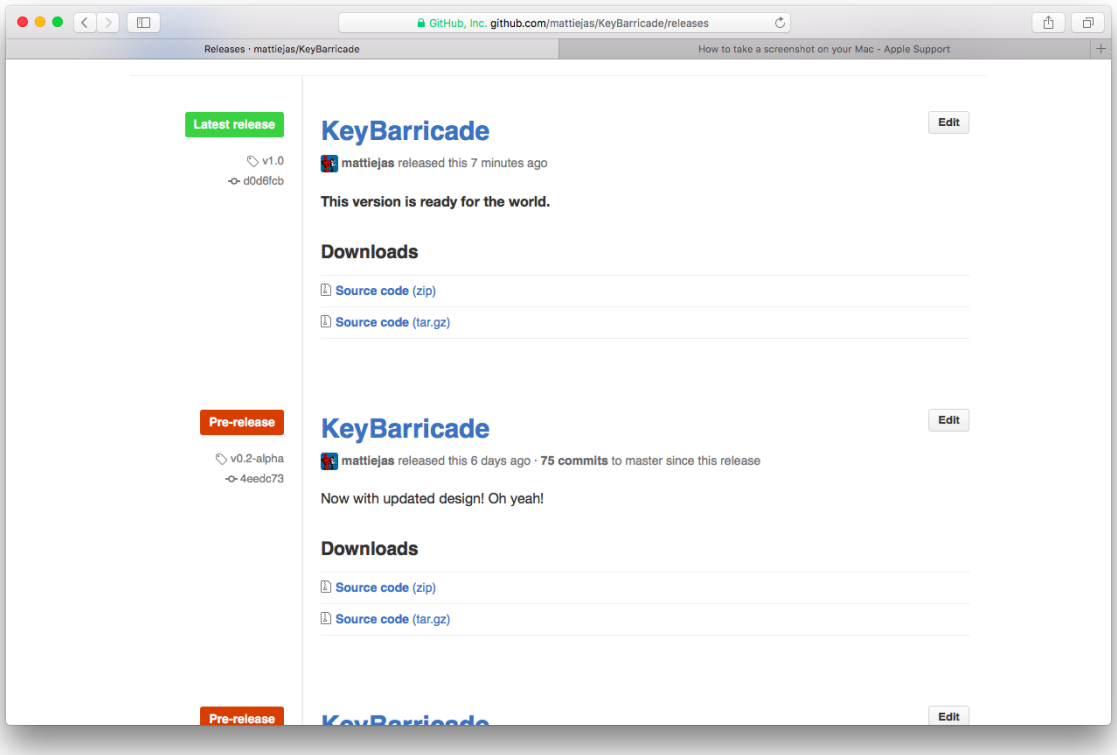
In opdracht van de opdrachtgever is er een spel ontwikkeld. In dit document is de ontwikkeling in verschillende stappen beschreven. Allereerst wordt er vastgesteld wat de exacte eisen zijn van het programma.

Daarnaast is er een analyse klassendiagram opgesteld voor de business.

Ten slotte is er tevens een design klassendiagram opgesteld voor de ontwikkelaars en treft u een aantal JUnit testen aan. Uiteraard heeft iedere hoofdstuk een uitgebreide uitleg.

Verder moet er nog vermeld worden dat dit project met de waterfall aanpak is ontwikkeld.

Qua versiebeheer hebben we gebruik gemaakt van GitHub. Dit was een goede oplossing om tegelijkertijd met elkaar het spel te kunnen ontwikkelen. In onze repository hielden we ook onze taken, eventuele vragen en interessante bronnen bij.



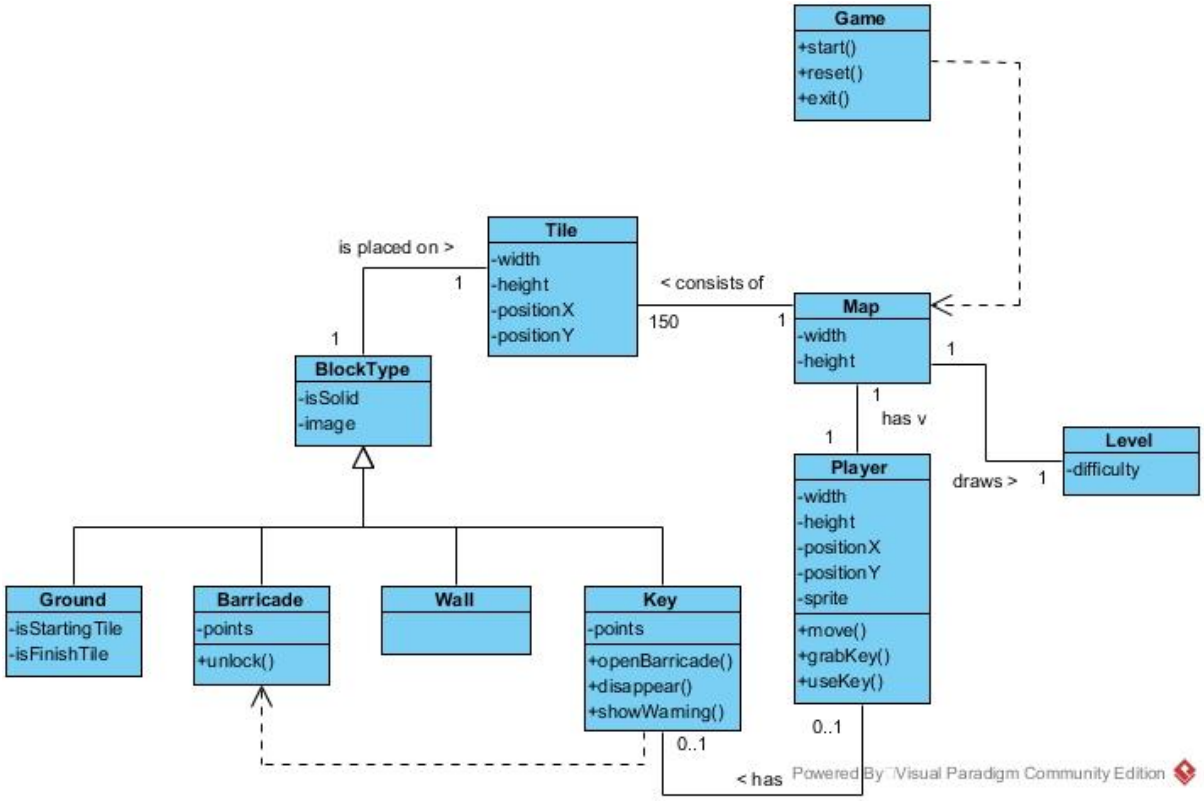
2. EISEN AAN HET SYSTEEM

Om het spel op een correcte wijze te ontwikkelen is het van belang om de eisen van de opdrachtgever bij elkaar te verzamelen. Uit die eisen moet een compleet beeld ontstaan over hoe het spel moet werken en hoe de gebruiker het spel in gebruik zal nemen. Om voor een degelijk, overzichtelijk en compleet beeld te zorgen zult u hieronder alle eisen nauwkeurig geformuleerd terugvinden.

- Het spel bestaat uit een vierkant speelveld.
- Het doel van het spel is om van punt a naar punt b te lopen.
- In het speelveld bevinden zich muren.
- In het speelveld bevinden zich barricades die elk een aantal punten bezit.
- In het speelveld bevinden zich sleutels die elk een aantal punten bezit.
- Een sleutel kan een barricade openen als het dezelfde aantal punten bezit als de barricade.
- Bij het openen van een barricade zal deze verdwijnen van het speelveld.
- De speler kan een sleutel oppakken.
- De speler mag maar één sleutel bij zich houden.
- Als de speler een sleutel oppakt, verdwijnt deze van het speelveld.
- Als de speler een sleutel pakt terwijl de speler een sleutel bij zich had, verdwijnt deze. De speler bezit nu de nieuw opgepakte sleutel.
- De speler kan de sleutel niet terugzetten op het speelveld.
- Sleutels kunnen oneindig op barricades gebruikt worden, mits het aantal punten van de sleutel en barricade overeenkomen.
- Bij het niet passen van een sleutel verschijnt een melding op het scherm.
- De speler beweegt zich door het speelveld doormiddel van de pijltjestoetsen.
- Het moet mogelijk zijn voor de gebruiker om exact hetzelfde speelveld opnieuw te spelen.
- Als de speler het eindpunt heeft bereikt, komt er een melding in het scherm dat de gebruiker het spel heeft gewonnen.
- Muren, barricades en sleutels zullen willekeurig in het speelveld worden gegenereerd.
- Niet ieder speelveld heeft een oplossing.
- Het spel moet verschillende moeilijkheidsgraden bevatten.
- Het ontwerp van het spel moet geschikt zijn voor uitbreidingen.

3. HET ANALYSE KLASSENDIAGRAM

Om voor de business een beeld te schetsen hoe het spel in elkaar steekt en hoe “objecten” met elkaar om gaan hebben wij een analyse klassendiagram opgesteld. Figuur 1 laat een afbeelding zien van het analyse klassendiagram. Op de volgende bladzijde wordt uitleg gegeven over alle klassen, attributen, operaties, associaties en dependencies.



Figuur 1

Wij hebben ervoor gekozen op om het speelveld te laten bestaan uit 150 vakjes. De klasse Map omvat het speelveld. De klasse Tile omvat het vakje. Dit verklaart ook de associatie tussen Map en Tile. De Map “consist of” (bestaat uit) 150 Tiles. Een Tile zal zich altijd bevinden op een Map. Zowel Map als Tile heeft een breedte (`width`) en een hoogte (`height`). Tile heeft echter ook een `positionX` en een `positionY` om de locatie op de map te bepalen.

In het spel komen muren, barricades, sleutels en grond voor om op de lopen. Deze potentiele klassen hebben allemaal de overeenkomst dat ze op een Tile komen te liggen, maar hebben verschillende eigenschappen. Daarom hebben wij ervoor gekozen om de klassen Ground, Barricade, Wall en Key subklassen te maken van de klasse Blocktype. Alle klassen die overerven van Blocktype zijn wel of niet solid (vast). Hiermee wordt bepaald of de speler er doorheen kan lopen. Alle klassen die overerven van Blocktype hebben een afbeelding waarmee ze goed herkent kunnen worden.

Ground heeft als unieke attributen `isStartingTile` en `isFinishTile`. Hiermee wordt bepaald waar een speler begint en waar de speler het spel kan eindigen.

Barricade heeft als uniek attribuut `points` en als unieke operatie `unlock()`. Points zorgen ervoor dat de Barricade alleen geopend (unlocked) kan worden met een in punten overeenkomende sleutel. De methode `unlock()` spreekt voor zichzelf.

Wall heeft geen unieke attributen of operaties. Toch hebben wij besloten er een klasse van te maken. Een Wall heeft namelijk een uniek afbeelding en zal altijd solid zijn. Wall overerft dit van Blocktype. Vandaar dat het niet wordt aangegeven in de klasse Wall zelf.

Key heeft als uniek attribuut `points` en als unieke operaties `openBarricade()`, `disappear()` en `showWarning()`. De points dienen voor het openen van barricades met een overeenkomend aantal punten. Met de operatie `openBarricade()` kan een barricade worden geopend, mits de punten overeenkomen. Komen de punten niet overeen dan zal de operatie `showWarning()` een melding geven dat de sleutel niet past. Als de speler de sleutel oppakt zit deze in zijn zak en zal de operatie `disappear()` ervoor zorgen dat de sleutel van het veld verdwijnt. Key heeft een dependency met Barricade, omdat de sleutel bepaalde barricades kan openen. De sleutel hoeft niet te weten welke barricade het is, alleen het aantal punten. Vandaar de keuze van de dependency.

Vanwege het feit dat Ground, Barricade, Wall en Key overerven van Blocktype, heeft Tile alleen een associatie met Blocktype nodig. Voor een Tile is het namelijk belangrijk te weten of de Blocktype solid is en welke afbeelding hij mee krijgt. Een Tile bevat ook altijd één Blocktype en een Blocktype is altijd geplaatst op één Tile.

De klasse Player heeft de attributen `width`, `height`, `positionX`, `positionY` en `sprite`. `Width` en `height` bepalen de hoogte en de breedte van de speler. `positionX` en `positionY` geven aan waar de speler zich in de Map bevindt. `sprite` is de afbeelding van de speler waarmee hij op het speelveld getekend wordt.

Daarnaast bevat de klasse Player de operaties `move()`, `grabKey()` en `useKey()`. Met de operatie `move()` kan de speler zich voortbewegen in het speelveld. De operatie `grabKey()` zorgt ervoor dat een sleutel in het speelveld op kan worden gepakt. De speler kan een sleutel gebruiken op een barricade door middel van de operatie `useKey()`.

De klasse Player heeft één associatie, namelijk met de klasse Key. Dit is omdat de sleutel dan kan worden opgepakt en de speler altijd moet weten welke sleutel hij in zijn zak heeft. Een sleutel bevindt zich op het veld, maar de speler kan hem ook vasthouden. De speler heeft niet altijd een sleutel vast.

Player heeft ook een associatie met de klasse Map. Een speelveld bevat altijd één speler, die speler moet zich altijd op het speelveld bevinden. Doordat de speler zich voor een langere tijd op het speelveld bevindt voldoet een dependency niet.

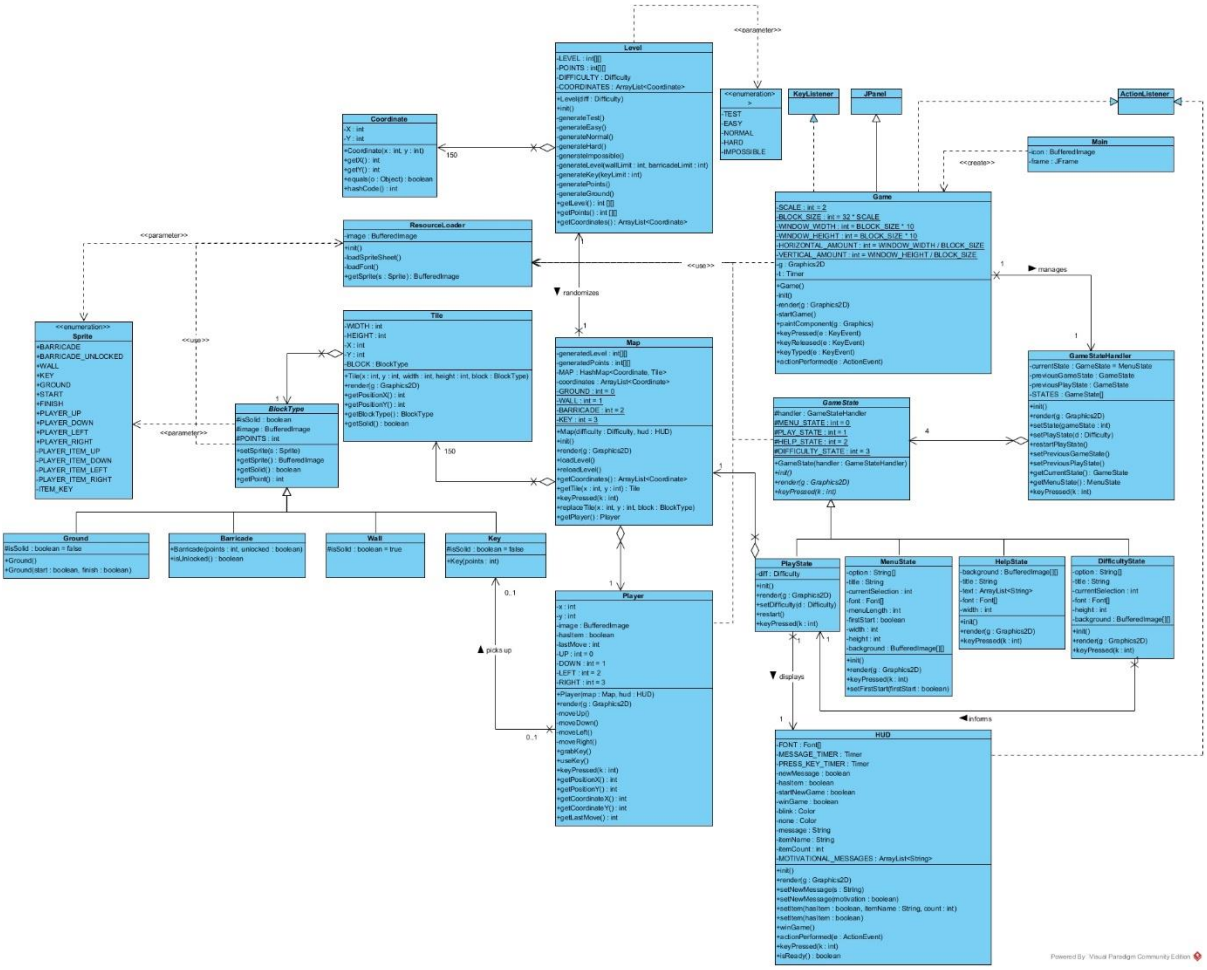
De klasse Level bevat als enige attribuut `difficulty` en een associatie met de klasse Map. Met de `difficulty` wordt de moeilijkheidsgraad van een Level bepaald. Doordat Level een associatie met Map heeft kan de map met de `difficulty` van level bepalen hoe alle Tiles met Blocktypes moeten worden neergezet.

Als laatste is de klasse Game aan de beurt, deze heeft de operaties `start()`, `reset()` en `exit()` en heeft een dependency naar Map. De operaties van Game geven aan Map door of er een spel moet stoppen, opnieuw moet starten of dat het spel moet worden verlaten.

Nu het analyse klassendiagram is behandeld zal de volgende stap het design klassendiagram zijn. Hierbij worden alle technische benodigdheden voor het spel gemodelleerd. Dit zult u aantreffen in het volgende hoofdstuk.

4. HET DESIGN KLASSENDIAGRAM

Het design klassendiagram is bedoeld om de technische structuur van het spel overzichtelijk weer te geven.



Figuur 2

In de Main klasse wordt een nieuwe instantie van de klasse Game aangemaakt. Dit is een relatie van korte duur, oftewel een dependency met het stereotype '`create`'. De klasse Game heeft één overerving en twee implementaties.

Game overerft de klasse JPanel, zodat het frame in de Main klasse een instantie van Game kan aanmaken en toevoegen als component. De tweede reden is dat JPanel de mogelijkheid geeft om met `paintComponent()` objecten te visualiseren.

Verder maakt Game ook gebruik van KeyListener, op deze manier is het mogelijk om input te geven via het toetsenbord. Zoals te zien in het design zijn alle operaties van KeyListener geïmplementeerd. Tevens implementeert Game de ActionListener interface. Door het gebruik van de bijbehorende methode `actionPerformed()` kunnen we met een timer de gameloop realiseren. Om de zoveel tijd worden alle componenten binnen de JPanel opnieuw getekend.

Vervolgens heeft Game een dependency naar ResourceLoader. De Game initialiseert namelijk de ResourceLoader wanneer het spel wordt gecreëerd. Op deze manier kan de ResourceLoader de benodigde middelen zoals lettertype en afbeeldingen ophalen. Tot slot heeft de Game een associatie

met GameStateHandler, omdat de GameStateHandler ervoor zorgt dat de juiste staat het van spel op het scherm wordt weergegeven. De associatie is uni-directioneel, omdat Game de GameStateHandler beheert en niet andersom. Hierdoor hoeft GameStateHandler niet af te weten van de Game klasse.

Zoals eerder gezegd zorgt GameStateHandler ervoor dat er vloeiend gewisseld kan worden tussen de verschillende staten in het spel. Vervolgens heeft GameStateHandler een aggregatie met GameState, omdat hij alle staten van het spel beheert. Elke GameState heeft zijn eigen verantwoordelijkheden. Deze aggregatie is bi-directioneel, omdat GameState aan de GameStateHandler moet kunnen doorgeven dat er gewisseld wordt tussen de verschillende staten. Doordat alle staten van de game worden overerft van GameState, voldoet een aggregatie tussen de GameStateHandler en GameState. Dit zorgt ook voor een lage koppeling.

GameState is een abstracte klasse, hierdoor worden de overervende klasse verplicht om de benodigde methodes te implementeren. Verder worden er attributen geïnitialiseerd die verwijzen naar de verschillende staten in de GameStateHandler. Tot slot heeft GameState een dependency naar ResourceLoader. Deze is nodig om de benodigde middelen op te halen voor het generen van een staat.

PlayState, MenuState, HelpState en DifficultyState zijn de staten die overerven van GameState. Elke staat heeft zijn eigen implementatie en functie, dit zorgt voor een hoge cohesie. Er is een onderlinge associatie tussen de DifficultyState en PlayState, omdat PlayState van DifficultyState moet weten wat de moeilijkheidsgraad is van het level dat gecreëerd moet worden. De associatie is uni-directioneel, omdat DifficultyState alleen de moeilijkheidsgraad hoeft door te geven.

PlayState heeft een associatie met de klasse HUD. Deze zorgt ervoor dat er berichten op het scherm worden getoond als de gebruiker het spel speelt. Deze associatie is uni-directioneel, omdat HUD alleen de berichten verwerkt. Zodra de gebruiker het spel heeft gewonnen, wordt bij de PlayState via de HUD een eindscherm getoond. Na het indrukken van de spatie knop wordt je teruggeleid naar het menu.

Verder heeft PlayState een aggregatie met Map, omdat PlayState afhankelijk is van het speelveld die Map voor hem tekent. PlayState geeft via deze aggregatie de moeilijkheidsgraad en de HUD mee aan de klasse Map. De aggregatie is uni-directioneel, omdat Map hoeft niks te doen met PlayState.

Zoals eerder vermeldt zorgt Map ervoor dat het speelveld getekend wordt op het scherm. Deze klasse krijgt via zijn constructor twee argumenten mee van PlayState, namelijk: Difficulty en HUD. Map gebruikt Difficulty om via de associatie met Level de structuur van het speelveld te bepalen aan de hand van de gekozen moeilijkheidsgraad. Verder stuurt Map de HUD door naar Player, zodat zijn acties kunnen worden doorgegeven aan HUD. Vervolgens kan de klasse HUD het juiste bericht op het scherm tonen.

De klasse Level genereert een speelveld. In totaal zijn er 150 coördinaten waarvoor een specifiek blok type wordt gegenereerd aan de hand van de gekozen moeilijkheidsgraad. Deze relatie wordt weergegeven met een dependency naar de Difficulty enumeratie. Het gegenereerde speelveld wordt opgeslagen, die weer wordt opgehaald door de klasse Map die het speelveld vervolgens tekent.

De klasse Player wordt aangemaakt door Map. De speler loopt op een speelveld. Dit is een bi-directionele associatie. Ze weten van elkaar dat ze bestaan, want Player moet een Tile kunnen ophalen uit de klasse Map. Dit om te weten of de speler daarnaartoe mag lopen. Verder krijgt de speler ook een HUD-object mee via de klasse Map. Met bepaalde handelingen stuurt Player informatie door naar de HUD die vervolgens bepaald welk bericht op het scherm wordt weergegeven. Verder heeft Player een uni-directionele relatie met de Key klasse, omdat het object

van de sleutel niet hoeft te weten dat het is opgepakt door de speler. Ten slotte heeft de speler niet altijd een sleutel bij zich. De sleutel kan zowel op het speelveld liggen of van de speler zijn. De speler kan zelf maar één sleutel in zijn zak hebben. Dit verklaart de multipliciteit '`0..1`' aan beide kanten.

Teruggekomen bij Map. Deze heeft een aggregatie met de klasse Tile, aangezien het speelveld bestaat uit 150 vakken. Dit is uiteraard terug te zien in de multipliciteit. De aggregatie is uni-directioneel, omdat een vak niet hoeft te weten op welke map hij ligt.

Tile is zoals eerder gezegd een vak in het speelveld. In de constructor wordt er naast de coördinaten en afmeting ook een BlockType meegegeven. Doordat Tile een aggregatie heeft met de klasse BlockType kunnen alle verschillende subklassen van BlockType op de Tile worden geplaatst.

BlockType is een abstracte klasse die vier subklasse onder zich heeft. Dit zijn de klasse Ground, Barricade, Wall en Key. Door de overerving hebben alle subklassen de methodes en attributen van BlockType. Verder heeft BlockType een dependency naar ResourceLoader en gebruikt de parameter Sprite om de juiste resources in te laden. Dit verklaart de dependency naar de klasse Sprite.

5. JUNIT TEST

We hebben twee klassen getest met JUnit. Beide testen hebben code coverage én decision coverage behandeld. Hieronder tref je meer informatie.

5.1 ResourceLoader

De ResourceLoader klasse is een belangrijk onderdeel van het spel. Deze klasse zorgt ervoor dat niet alleen objecten gevuld kunnen worden met een BufferedImage, maar het laadt ook de spritesheet en het font in.

Door het gebruik van de methode `getSubImage()` hoeft er maar één spritesheet gebruikt te worden als resource. Dit zorgt ervoor dat het spel minder zwaar wordt, omdat er maar één afbeelding ingeladen hoeft te worden.

Voor deze klasse zijn uiteindelijk 16 tests geschreven. Deze test controleert bij de gekozen sprite op positie en afmeting. De afmeting en positie mag niet aangepast worden. Daarom staat er in de test de gewenste waardes van de gekozen sprite met de berekende positie en afmeting van de BufferedImage. Door deze test dekken we zowel code als decision coverage, omdat we binnen het switch statement alle mogelijke opties langsgaan.

Het voordeel van deze test is dat als we later de spritesheet willen uitbreiden, dat we de test vaker kunnen gebruiken om te controlerem of alle sprites de gewenste positie en afmeting hebben.

5.2 Player movement

In de test werd gekeken naar de volgende methodes:

- `moveUp()`
- `moveDown()`
- `moveLeft()`
- `moveRight()`

Deze methodes zijn belangrijk voor het voortbewegen van de speler. Ze bevatten allemaal methodes met een `if`-statement waarin gekeken wordt of de positie van de speler voldoet om een bepaalde richting op te lopen. Zie het volgende voorbeeld:

```
getCoordinateX() >= 0 && getCoordinateX() <= Game.HORIZONTAL_AMOUNT – 1
```

Waarin '`getCoordinateX()`' de x-coördinaat retourneert van de speler. Daarnaast is '`Game.HORIZONTAL_AMOUNT`' het aantal tiles wat zich in de breedte bevindt van het speelveld.

In de eerste conditie wordt gekeken of de x-coördinaat groter is of gelijk aan nul. Dit wil zeggen dat de speler in het speelveld moet staan. Dit is ook zo bij de tweede conditie, alleen beginnen de coördinaten bij nul, dus moet er één tile van de breedte van het speelveld afgetrokken worden. Hetzelfde gebeurd ook voor de volgende twee condities, maar dan voor de y-coördinaat.

Als de speler zich niet in het veld zou bevinden, kan deze zich nooit voortbewegen, omdat de eerste `if` altijd `false` zal retourneren. Mocht de speler zich wel in het veld bevinden, dan zal hij aan de klasse Map vragen of hij zich mag verplaatsen naar het volgende vakje waar hij naar toe wilt bewegen. Map zal doormiddel van de `getSolid()` methode van Tile aan Player doorgeven of hij zich naar de desbetreffende Tile mag verplaatsen.

Als laatste wordt bijgehouden wat de laatste ingedrukte toets van de gebruiker is. Hiermee kan worden bepaald in welke richting de speler een barricade wil openen.

In de eerste vijf testen in de PlayerMovementTest klasse is getest of de speler zich op de juiste manier voortbeweegt. Met de eerste vier testen wordt met elke test gekeken of alle richtingen op de juiste manier worden uitgevoerd door de coördinaten te controleren. In de andere test doet de speler een aantal verschillende bewegingen om te kijken of deze bewegingen worden uitgevoerd. Doordat er ook altijd wordt gekeken of de `lastMove` overeenkomt met de verwachte `lastMove`, spreken we van **code coverage.** Iedere regel code wordt namelijk afgegaan.

In de laatste vier testen van PlayerMovementTest wordt gekeken of de `lastMove` en coördinaten overeenkomen met de verwachte `lastMove` en coördinaten, als de speler uit het veld probeert te lopen. Hier spreken we van **decision coverage**, omdat bij een aantal moves de `if`-statements niet wordt uitgevoerd.

INSPIRATIEBRONNEN

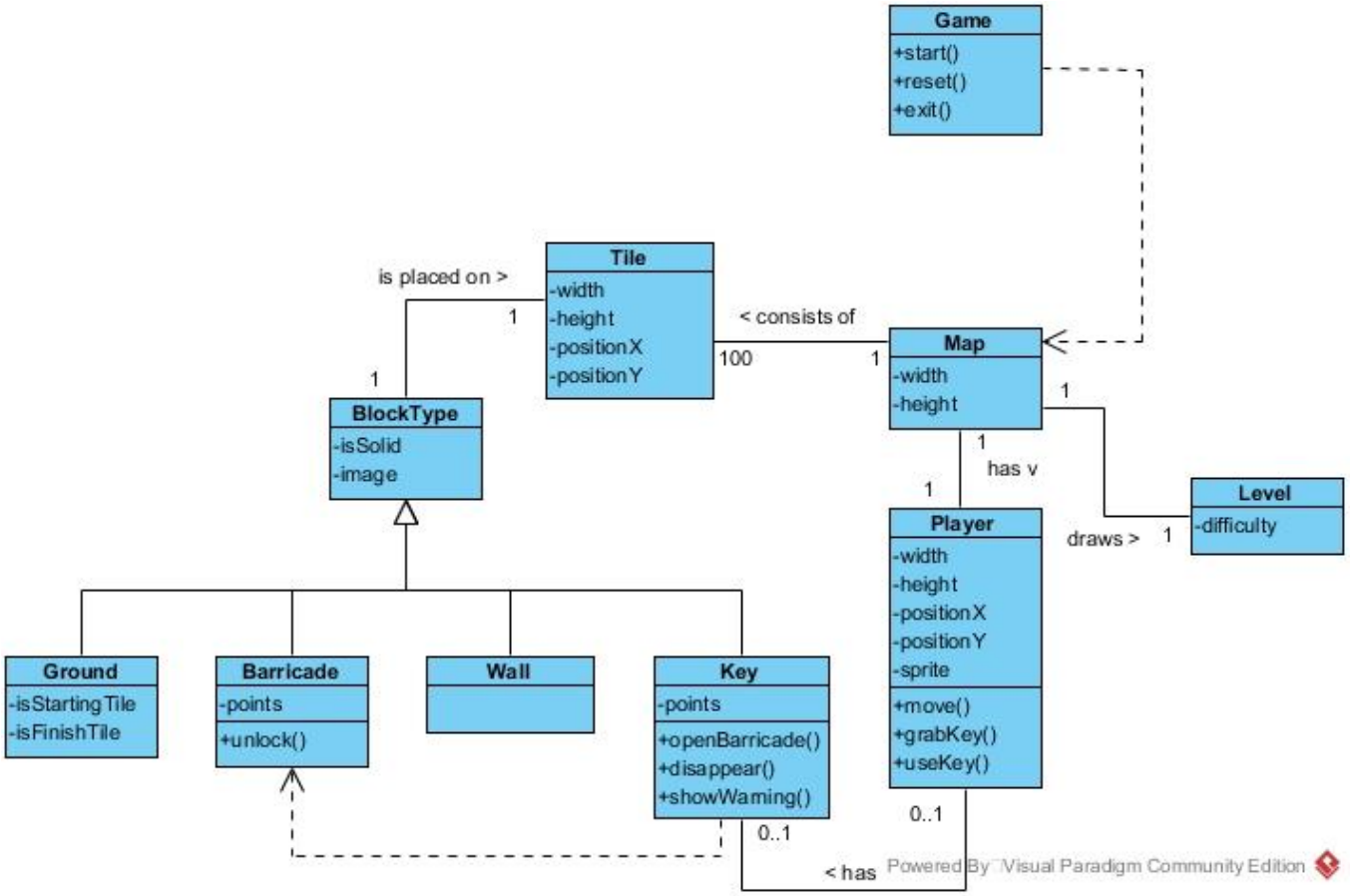
We hebben onze code zelf geschreven, maar deden hier en daar wel inspiratie op. We hebben ze voor de zekerheid bewaard. U kunt ze hieronder terugvinden.

corsiKa. (2011, augustus 22). *How to make a tile based map from an array in Java?* Opgehaald van StackOverflow: http://stackoverflow.com/questions/7150624/how-to-make-a-tile-based-map-from-an-array-in-java

Ewald, M. (sd). *Game State Management*. Opgehaald van Cygon's Blog: http://blog.nuclex-games.com/tutorials/cxx/game-state-management/

pj6444. (2014, februari 13). *Java 2D Platformer Tutorial #3 - The Game State Manager*. Opgehaald van YouTube: https://www.youtube.com/watch?v=OCcZUO4Zf6o

BIJLAGE: ANALYSIS



BIJLAGE: DESIGN