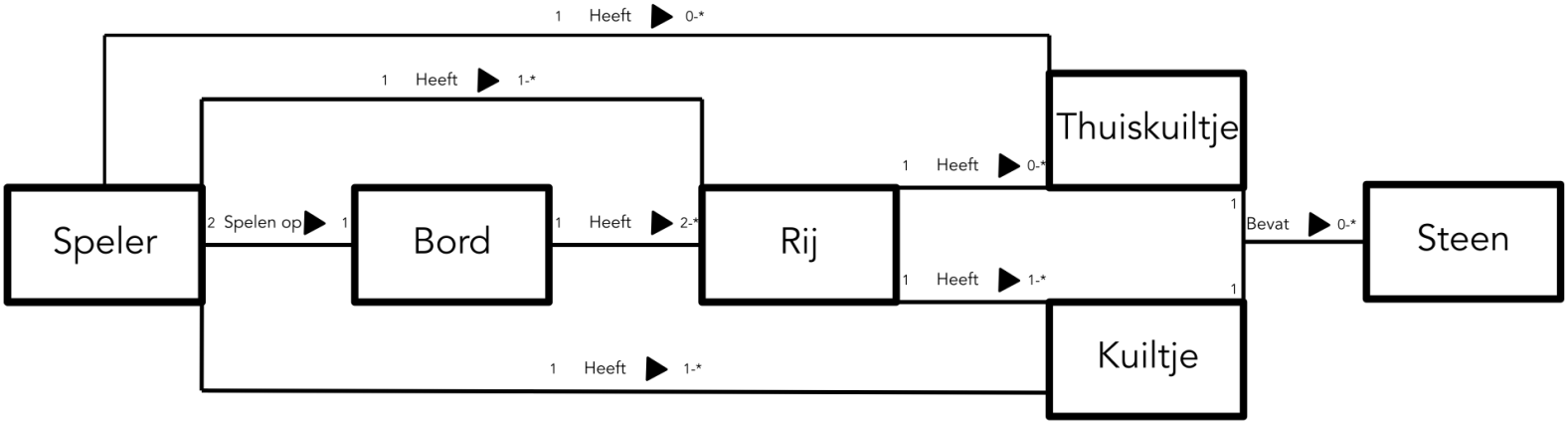
MSO Lab 2 - Imbert Dam & Luka de Vrij

Domain model



CVA

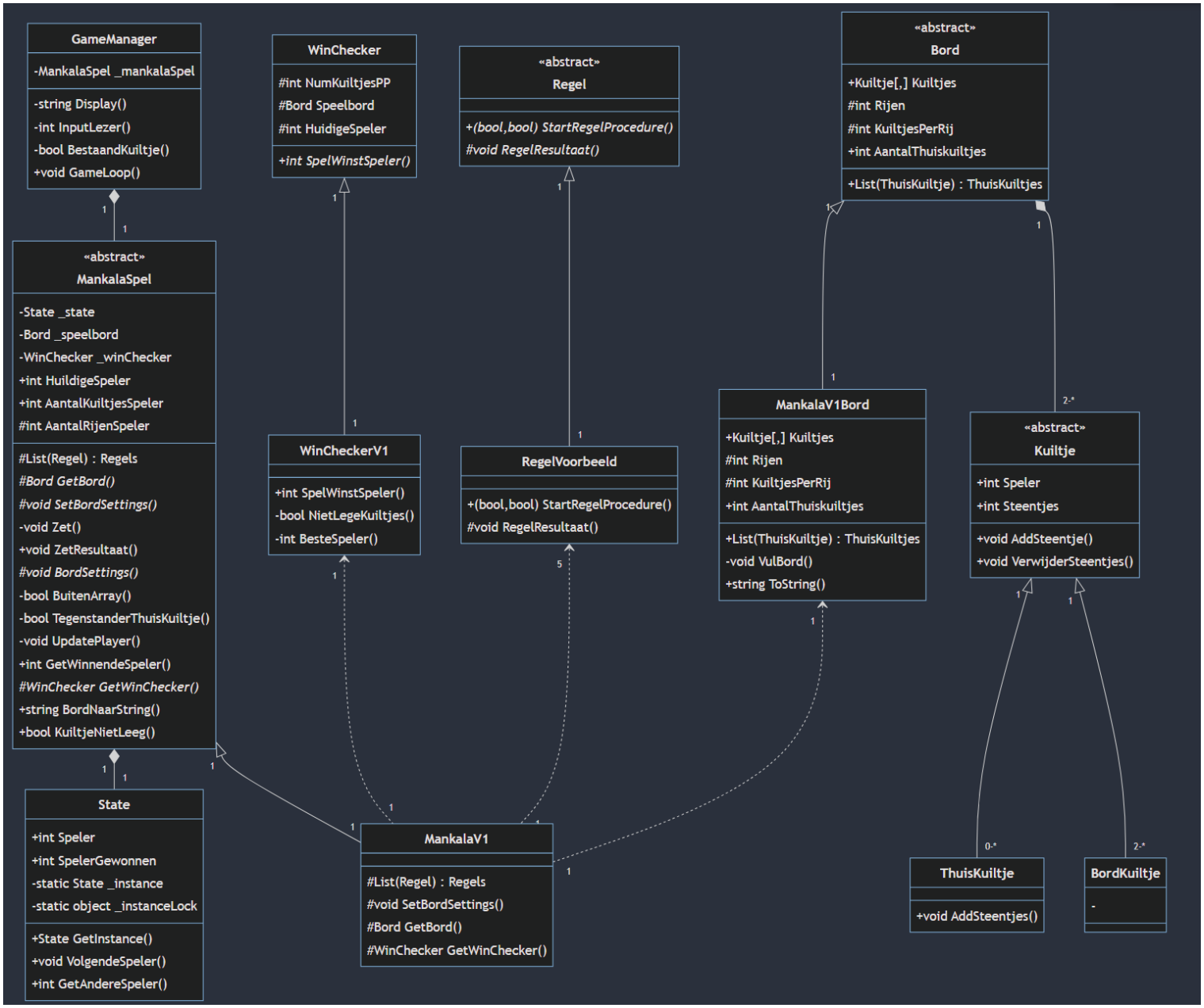
Commonalities	*Variations*
Mankala Spel	V1 V2 …
Kuiltjes	Thuiskuiltje Standaard kuiltje
Steentjes	V1-based V2-based …
Regels	V1-based V2-based …

Analysis matrix

	Mankala	**Other**
Aantal kuiltjes per speler	6	1-*
Aantal steentjes per kuiltje	4	1-*
Aantal thuiskuiltjes per speler	1	0-*
Aantal rijen per speler	1	1-*
Regels voor resultaat	De laatste steen komt terecht in het thuiskuiltje van de speler. De speler mag nu een nieuwe zet doen. Er is geen maximum aan het aantal keer dat een speler achter elkaar aan de beurt is. De laatste steen komt in een ander kuiltje dan het thuiskuiltje van de speler, en dat kuiltje was niet leeg. De speler pakt alle stenen in het kuiltje op, en gaat verder met de beurt. Er is geen maximum aan het aantal keren dat in één beurt stenen opgepakt kunnen worden. De laatste steen komt in een leeg kuiltje van de tegenspeler. De zet is over, en de beurt is over: de tegenspeler is aan de beurt. De laatste steen komt in een leeg kuiltje van de speler. Het kuiltje van de tegen- speler daar tegenover is leeg. De zet is over, de beurt is over: de tegenspeler is aan de beurt. De laatste steen komt in een leeg kuiltje van de speler. Het kuiltje van de tegen- speler daartegenover is niet leeg. De speler pakt de laatst uitgestrooide steen plus de stenen in het kuiltje van de tegenspeler ertegenover, en voegt ze toe aan zijn thuiskuiltje. De zet is over, de beurt is over, de tegenspeler is aan de beurt.	other
Regels voor einde spel	Het spel eindigt op het moment dat een speler niet kan spelen omdat hij geen speelkuiltjes met stenen meer heeft. De speler met de meeste stenen in het thuiskuiltje is de winnaar.	other

Voor het aantal steentjes per kuiltje is er vanuit gegaan van een uniforme verdeling. Bij de tweede regel voor het resultaat is er van uitgegaan dat eind kuiltje zijn stapeltje stenen opgepakt wordt en er een zet over wordt gedaan.

Software design (UML)



(https://gist.github.com/LukaDeVrij/cafc0d5d4e8c261ff11500c32a2e8763)

Uitleg over UML diagram

GameManager

GameManager is een static klasse die verantwoordelijk is voor de hoogste laag van controle. Het controleren van de input, en het besturen van de GameLoop(), welke de InputLezer() en winst controle regelt.

MankalaSpel

Deze klasse bestaat uit een groot aantal methoden die allen verantwoordelijk zijn voor een laag dieper, namelijk een oppervlakkige representatie van het spel. Deze klasse wordt

gerealiseerd door een subklasse, namelijk MankalaV1. MankalaSpel heeft controle over hoe de zetten gedaan worden, en wanneer. Daarnaast wordt er van spelers gewisseld en zijn er een groot aantal private hulpmethoden die nodig zijn in bijv. Zet() of ZetResultaat(), om enorme if-statements te voorkomen.

MankalaV1
Deze subklasse van MankalaSpel bevat variaties die in Mankala voor zouden kunnen komen. In de toekomst zouden meer methoden vanuit MankalaSpel zich naar MankalaV1 kunnen verplaatsen, omdat er meer verschillen optreden tussen Mankala varianten. Deze klasse heeft connecties met de verschillende *Products* die met factory methods worden gemaakt (zie design patterns).

State
De state houdt bij welke speler er aan de beurt is en of er al iemand gewonnen heeft. Ook bevat hij methoden die handig zijn voor het switchen van spelers. Hoewel State officieel geen State pattern is, gebruiken we hem wel op die manier. In de toekomst zou de State uitgebreid kunnen worden tot een echt design pattern. State is wel een singleton, waarmee we in ieder geval zeker weten dat er maar 1 enkele bestaat, waardoor de State altijd dezelfde is.

WinChecker & WinCheckerV1
Een concrete ‘product’ klasse die subklassen heeft op basis van de Mankala variant. Deze WinChecker wordt gebruikt in de MankalaV1/MankalaSpel klasse om na een zet te checken of er een winnaar aangewezen kan worden.

Regel & Regelvoorbeeld
De abstracte klasse Regel heeft in het UML diagram een enkel connectie, i.e. Regelvoorbeeld. Het is de bedoeling dat elke versie van Mankala een aantal Regels kent, waar hij zetten aan toetst. MankalaV1 heeft bijvoorbeeld 5 verschillende regels in onze C# implementatie. Elke Regel zal dezelfde methoden met andere implementaties hebben, en deze worden dus polymorf aangeroepen. Je kunt dit daarom zien als een strategy pattern (zie design patterns). De gameplay features zijn het meest aanwezig in deze klassen, de subklassen van Regels hebben namelijk een enorm grote invloed op de werking van het spel.

Bord & MankalaV1Bord
Bord bevat methoden die het bord initializeren, en een string representatie maken om te gebruiken in de console output. Ook het *Product* MankalaV1Bord wordt gemaakt met een factory method zodat deze direct gekoppeld staat aan MankalaV1, en dit maar een enkele keer gespecificeerd hoeft te worden. Het Bord bestaat onder andere uit een 2d array met de hoeveelheid rijen maal het aantal kuiltjes per rij aan Kuiltjes objecten. Deze variabelen hoeven maar op een enkele plek gewijzigd te worden. Een directe lijst met Thuiskuiltjes wordt gebruikt om deze snel te kunnen vinden, zonder door de gehele 2d array heen te hoeven itereren.

Kuiltje & ThuisKuiltje & BordKuiltje
BordKuiltje en ThuisKuiltje zijn in essentie niet echt verschillend in implementatie, maar het verschil in type is wel van belang voor verscheidene regels in onze standaard Mankala

variant. Ze bevatten methoden die het makkelijk maken om snel steentjes te verwijderen en toe te voegen. Mocht er in de toekomst meer moeten gebeuren in dit geval, dan hoeft dit enkel hier aangepast te worden.

Design Patterns

Factory Method:

We hebben getwijfeld of de abstract factory pattern voor deze game handig is bij het aanmaken van een bord, winst condities en zet resultaten dat kan variëren bij mankalaspellen. Omdat we er vanuit zijn gegaan dat er bij sommige varianten dezelfde regels en/of bord layout zijn, hadden we het gevoel dat een abstracte factory pattern overbodig was. We gebruiken dus een factory method pattern. Onze factory methodes zijn GetBord(), GetWinChecker() en in de constructor van MankalaV1 wordt een lijst van regels aangemaakt. Voor Bord, Regel en WinChecker zijn abstracte producten aangemaakt en concrete producten ervan geïmplementeerd. Dit zorgt ervoor dat een nieuwe variant van Mankala andere borden, regels en win-condities kan aanmaken, maar ook de al bestaande kan hergebruiken. Dit is erg vergelijkbaar met de method factory van DialogBox en WindowsButton uit de slides.

Strategy Pattern:

We hebben een vorm van een strategy pattern gebruikt voor de regels: We hebben een selectie van regels in een regellijst gezet en daar dezelfde methode aangeroepen. Afhankelijk van welke regel geselecteerd is (polymorfisme) wordt het bijbehorende algoritme gebruikt, net zoals in de slides. We hadden de resultaatregels ook in een methode kunnen zetten, echter als er bij een variant een gelijke regel zou worden gebruikt zou je die dan opnieuw moeten implementeren. Verder voor in het geval dat er in de toekomst andere soorten kuiltjes komen kunnen deze makkelijk geschaard worden onder de Kuiltjes, en deze gebruikt worden in het vullen van het Bord.

Singleton Pattern:

In het geval dat we een online multiplayer variant van dit spel hebben en om zeker te zijn dat er maar 1 gamestate tegelijkertijd kan bestaan, hebben we een singleton pattern geïmplementeerd, zoals in de slides. Dit ook om onszelf het niet mogelijk te maken toch per ongeluk een nieuwe instantie van State te maken. We hadden ook voor een klassiek state pattern kunnen kiezen, maar onze manier van state representeren is vergelijkbaar en de state pattern vonden we net te uitgebreid voor hoe wij state in ons ontwerp gebruiken.

Assumptions

- “Al of niet” geïnterpreteerd als “wel of niet”. We zijn er dus vanuit gegaan dat er een Mankala variant bestaat waar geen thuis kuiltjes in het spel zitten
- We zijn er vanuit gegaan dat er maar 1 spel tegelijkertijd kan worden gespeeld.
- Ook de regels waren ambigu en we hebben geïnterpreteerd dat de regels exclusive or waren, dus dat er geen regels tegelijkertijd waar kunnen zijn.
- Verder hebben we aangenomen dat in toekomstige versies van Mankala het mogelijk is dat regels hetzelfde kunnen blijven.

Evaluatie
Om het huidige ontwerp uit te breiden zijn er veel verschillende uitbreidingen waar van tevoren rekening mee zijn gehouden. We hebben geprobeerd de coupling zo laag mogelijk te houden zodat als een onderdeel veranderd wordt er relatief weinig aanpassingen moeten worden gemaakt.

We hebben erg ons best gedaan om het ontwerp van de Regels zo adaptief mogelijk te maken, dat wil zeggen dat het erg makkelijk moet zijn om een aantal nieuwe regels op te stellen, en deze te kunnen ‘mix en matchen’ met de oude regels, in een nieuwe Mankala variant. Dit systeem is daardoor erg adaptief en kan makkelijk in de toekomst gebruikt worden.

Stel dat er andere win-condities worden geïntroduceerd en/of nieuwe bord lay-out kan er een uitbreiding worden gemaakt van de abstracte klasse Bord en/of WinChecker waar je deze aanpast naar de nieuwe regels.
Om een nieuwe regelset te kunnen gebruiken kan er een uitbreiding van MankalaSpel worden gemaakt waar je het Bord kan kiezen, de zet resultaat regels, en de WinChecker. Nieuwe versies van Mankala kunnen dus worden gekoppeld met nieuwe (of bestaande) subklassen van Bord, WinChecker en verschillende Regels. Bord heeft controls om de hoeveelheid steentjes, kuiltjes en rijen te variëren.

Taakverdeling
- **Opdracht 1**
    - We hebben eerst alle onderdelen besproken en daarna samen CVA en de matrix samen uitgewerkt. Imbert heeft daarna het besproken domain model uitgewerkt.
- **Opdracht 2**
    - We hebben samen het design van het UML diagram gemaakt en Luka heeft die vervolgens uitgewerkt. De beschrijving van de klassen is door Luka gedaan en Imbert heeft de design patterns uitgewerkt.
- **Opdracht 3**
    - De evaluatie is samen besproken en uitgewerkt door Imbert
- **Opdracht 4**
    - We hebben de implementatie eerlijk verdeeld en tegelijkertijd stappen besproken. Luka heeft de code opgeknapt en comments geplaatst (ook voor onszelf)