|  |
| --- |
|  |

|  |
| --- |
|  |
| Rapport Chess3D |
| Een beschrijvend rapport over het ontwerp- en realisatieproces van Chess3D |



Instelling Hogeschool Zeeland

Opleiding: B-ICT

Docent: D. de Waard

Course: JEDI (XXXX) 2009-06-14

Auteurs: Jeffrey Krist (00042708)  
Christophe Hesters (00042709)  
Casper van Dijk (000XXXX)

Datum: 18-6-2009

**Rapport Chess3D**

Een beschrijvend rapport over het ontwerp- en realisatieprocess van Chess3D

Instelling Hogeschool Zeeland

Opleiding: B-ICT

Docent: D. de Waard

Course: JEDI (XXXX) 2009-06-14

Auteurs: Jeffrey Krist (00042708)  
Christophe Hesters (00042709)  
Casper van Dijk (000XXXX)

Datum: 18-6-2009

Inhoudsopgave

[1 Inleiding 1](#_Toc233105355)

[2 Ontwikkelmethode 2](#_Toc233105356)

[3 GRASP Patterns 5](#_Toc233105357)

[4 Design patterns 7](#_Toc233105358)

[5 Logboek 8](#_Toc233105359)

[5.1 Week 1 (14 april 2009) 8](#_Toc233105360)

[5.2 Week 2 (21 april 2009) 8](#_Toc233105361)

[5.3 Week 3 (28 april 2009) 8](#_Toc233105362)

[5.4 Week 6 (19 mei 2009) 9](#_Toc233105363)

[5.5 Week 7 9](#_Toc233105364)

[5.6 Week 8 (2 juni 2009) 9](#_Toc233105365)

[5.7 Week 9 (9 juni 2009) 10](#_Toc233105366)

[5.8 Week 10 (16 juni 2009) 10](#_Toc233105367)

[Bijlage I Use case diagrammen 11](#_Toc233105368)

[Bijlage II Use cases 12](#_Toc233105369)

[Bijlage III Operation Contracts 17](#_Toc233105370)

# Inleiding

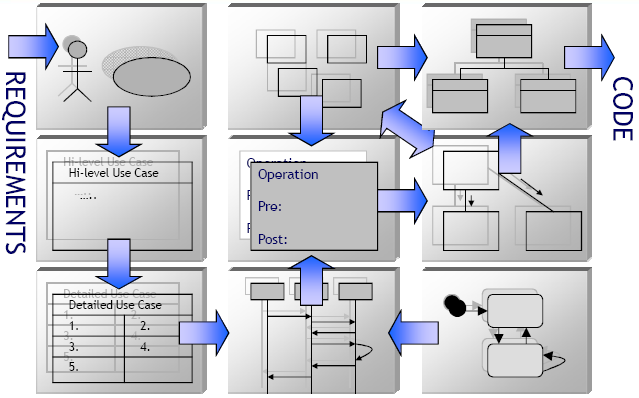
Dit document ontstaat naar aanleiding van de course JEDI (2009), gegeven door D. de waard. Tijdens de tweede helft van het jaarsemester werken de studenten in groepjes aan een zelfverzonnen softwareapplicatie. Dit gebeurd door een door de docent vastgestelde ontwikkeltraject te doorlopen. Tijdens het ontwikkelproces en de realisatie wordt in dit document bijgehouden - ondersteund door tussentijdse resultaten – en kan daardoor gezien worden als logboek.

Het doel van dit document is de desbetreffende docent of andere geïnteresseerden inzicht te geven in het projectverloop en de daardoor tussentijds ontstaande tussenresultaten. Vanaf het moment de course tot ten einde dreigt te komen, en de applicatie hopelijk succesvol is gerealiseerd, beoordeeld de docent aan de hand van dit document het project.

Dit document is chronologisch opgebouwd, per week is de opdracht uitgereikt om de volgende stap in het ontwikkeltraject te doorlopen. De resultaten van die opdrachten is in dit document vastgelegd.

# Ontwikkelmethode

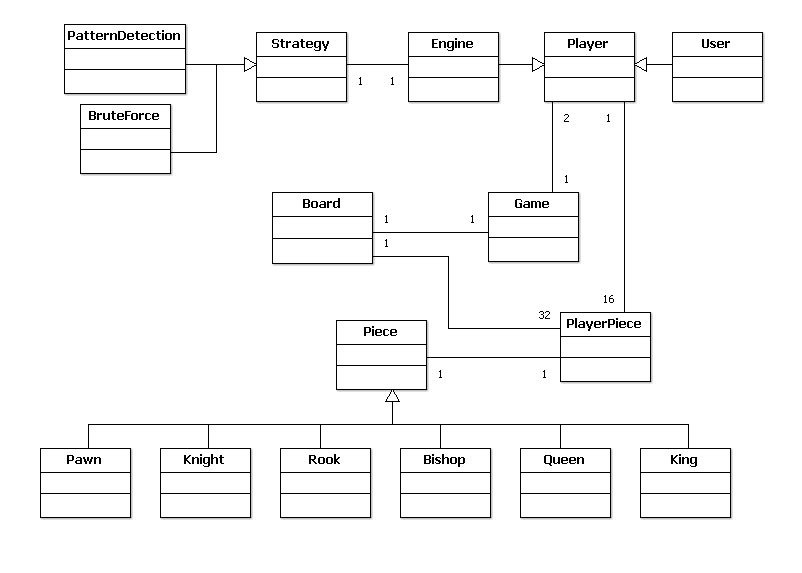
Om dit project in goede banen te leiden hebben we de Unified Processing ontwikkelmethodiek gebruik gemaakt. Deze methodiek is iteratief, dat betekent dat enkel fases meerder keren herhaald worden, en heeft als voordeel dat de specificatie van de software gaandeweg kan veranderen. Dat kan bijvoorbeeld komen door wijzigende eisen van de opdrachtgever of complicaties tijdens het ontwikkelen van de software. Figuur 1 geeft schematisch weer welke stappen er gevolgd worden tijdens dit proces.



Figuur UP Diagram

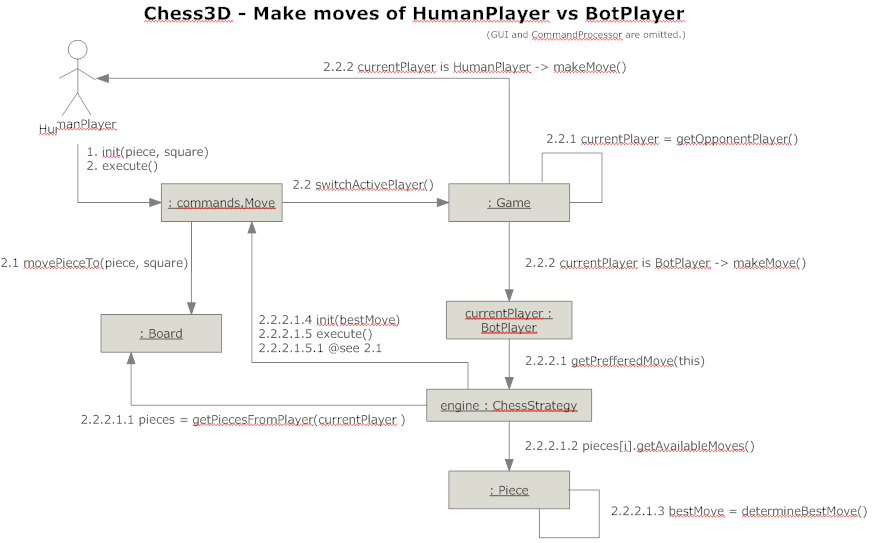
De eerste fase bestaat uit het maken van use case diagrammen. Hierin wordt grafisch vastgelegd welke functionaliteit het systeem moet hebben en welke actoren daarvan gebruikmaken. In bijlage I zijn alle use case diagrammen opgenomen die door ons zijn ontworpen. Direct daarop volgen hi-level use cases. Hierin staat globaal beschreven welke stappen er moeten worden doorlopen tijdens een taak of bij een bepaalde functionaliteit, deze zijn opgenomen in bijlage II. Dit is nog niet gedetailleerd genoeg om software van te kunnen ontwerpen, daarom zijn er ook nog detailed use cases. Dit zijn hele gedetailleerde beschrijvingen waarin stap voor stap staat wat er moet gebeuren, inclusief de alternatieve paden. Dit beschrijft de hele linkse rij in figuur 1.

Sequence diagrammen beschrijven hoe processen eruit zien door de interacties tussen verschillende objecten tijdens dat proces, inclusief de volgorde waarin ze gebeuren, weer te geven. Tijdens het opstellen van deze sequence diagrammen wordt gebruik gemaakt van de detailed use cases. Ook wordt nagedacht over het domein model. Een domein model is een conceptueel model van het systeem waarin de entiteiten van het systeem zijn opgenomen, inclusief de onderlinge relaties tussen deze entiteiten. Figuur 2 geeft het domein model weer wat ontwikkeld is voor dit project. Als de entiteiten en interacties bekend zijn kunnen deze als ‘operaties’ beschreven worden in operation contracts, deze specificeren de voorwaarden voor en na het uitvoeren van de operatie. Als dit vooraf is vastgesteld kan het werk makkelijker verdeeld worden over meerdere mensen. Ieder kan dan zijn deel ontwikkelen uitgaande van de vastgestelde voorwaarden. Bijlage III bevat een codevoorbeeld waarin de operation contracts zijn verwerkt zoals de docent dit heeft aangegeven. In de broncode van het spel zijn nog meer operation contracts opgenomen. Dit beschrijft de middelste rij in figuur 1.



Figuur domein model

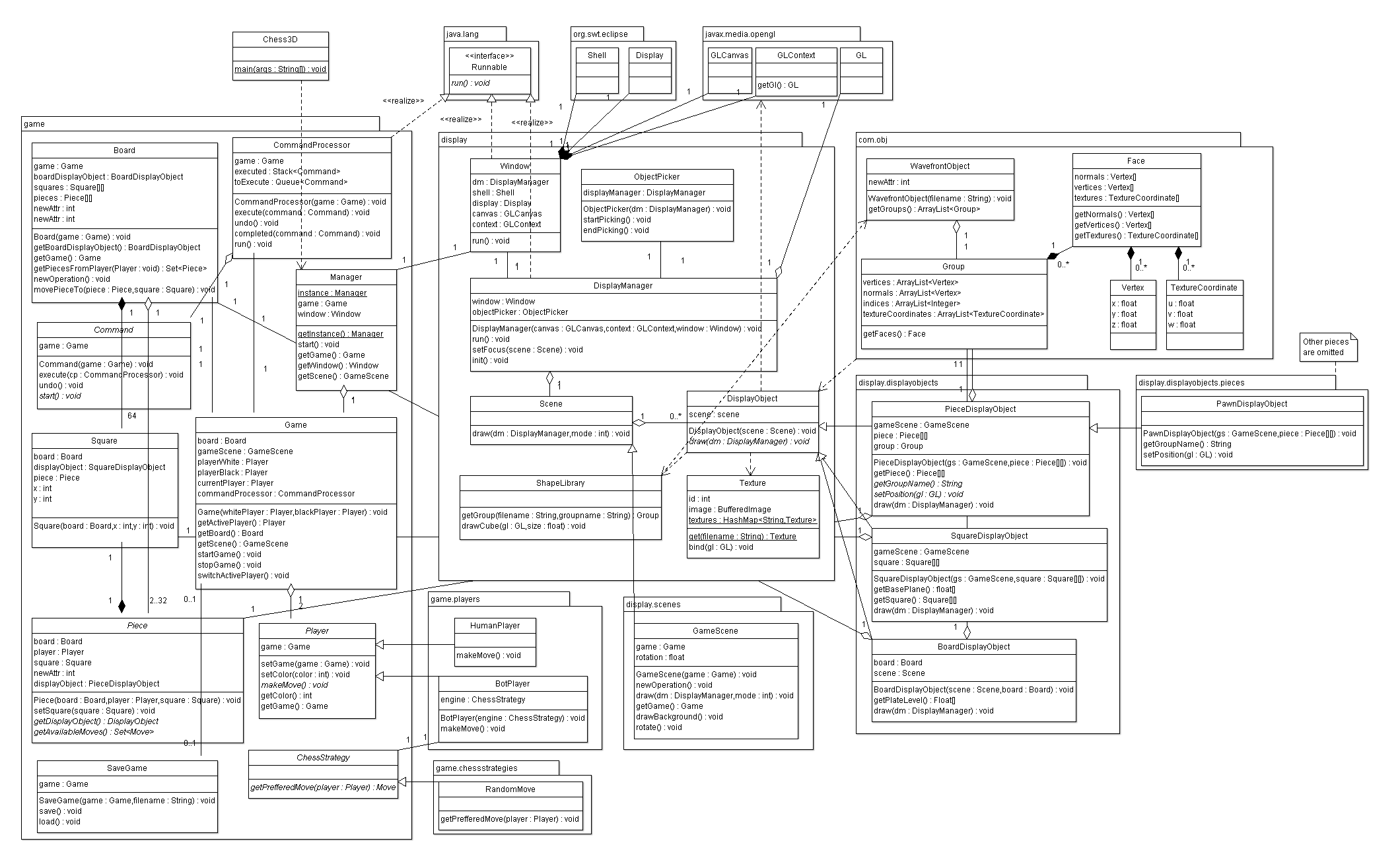
Activity diagrammen zijn niet behandeld in tijdens de cursus omdat het weinig gebruikt worden en ook weinig meerwaarde hebben voor dit project. Wat er nog overblijft zijn communicatiediagrammen en klasse diagrammen. Een communicatie modelleert de interactie tussen objecten in sequentiële ´berichten´. Omdat de objecten in een communicatie diagram, in tegenstelling tot sequence diagrammen, vrij geplaatst kunnen worden gebruikt men een nummeringschema om de volgorde aan te geven. Het communicatiediagram wat voor dit project is ontwikkeld is weergegeven in figuur 3.



Figuur Communicatie diagram

Het communicatiediagram in figuur 3 beschrijft de interacties die plaatsvinden als de speler een stuk verzet en de schaak engine daarop reageert. Figuur 4 geeft het conceptueel klassendiagram weer, welke het allereerste prototype betreft.

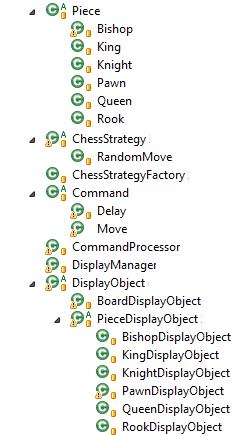
Figuur klassendiagram



# GRASP Patterns

Tijdens het ontwikkeltraject hebben we GRASP patterns gebruikt om de verantwoordelijkheden en methoden toe te kennen aan klassen. GRASP voor: General Responsibility Assignment Software Patterns/Principles en bestaat uit 10 regels die je helpen bepalen de verantwoordelijkheden en methoden toe te kennen aan bepaalde klassen, de regels zijn hieronder opgesomd:

1. Information expert
   * Ken de verantwoordelijkheid toe aan de klasse die alle essentiële informatie bezit om het uit te voeren.
2. Creator
   * Ken de verantwoordelijkheid voor het instantiëren van klasse A toe aan de klasse die A bevat, de informatie nodig heeft om A te instantiëren, een verzameling bijhoud van A, veel gebruik maakt van A of A Bewaard
3. Controller
   * Wat is het eerste object onder de user interface die systeem handelingen ontvangt en coördineert. Ken die verantwoordelijkheid toe aan een object die:
     1. Het gehele systeem representeert, het ‘root object’, het ‘apparaat’ waar de software op draait of een groot subsysteem
     2. Een use case scenario representeert waarbinnen systeemhandelingen voorkomen
4. Low coupling
   * Verdeel de verantwoordelijkheid zo dat onnodige koppeling minimaal blijft. Dit kan goed gebruikt worden om alternatieven tegen elkaar af te wegen.
5. High cehesion
   * Zorg dat de objecten gefocust blijven, zodat ze alleen doen wat ze moeten doen en niet meer. Een hoge cohesie gaat vaak gepaard met low coupling. Verdeel de verantwoordelijkheden zodanig dat de cohesie hoog is.
6. Polymorphism
   * Als het gedrag per type varieert maak je gebruik van subclassing. De verantwoordelijkheid wordt dan toegekend aan de superklasse maar elke subklasse bepaald dan het gedrag zelf.
7. Pure fabrication
   * Als je de verantwoordelijkheid niet logischerwijs kan toekennen aan een klasse omdat het buiten het probleemdomein valt en het de low coupling en high cohesion patterns zou schaden, ken dan de verantwoordelijkheden toe aan een nieuwe klasse die niets te maken heeft met het probleemdomein.
8. Indirection
   * Hoe ken je verantwoordelijkheden toe zodat je directe koppeling met objecten die aan verandering onderhevig zijn zoveel mogelijk voorkomt? Ken de verantwoordelijkheden toe aan object dat zich tussen het object bevind dat aan wijzigingen onderhevig is en de overige klassen.
9. Protected variation
   * Hoe ontwerp je componenten zo dat variaties binnen deze componenten niet leiden tot ongewenste wijzigingen daarbuiten? Bepaal waar en welke variaties in het component kunnen voorkomen. Stel de verantwoordelijkheden zo op dat er wel een stabiele interface ontstaat (de variaties in je component leiden niet tot wijzigingen in je interface.
10. Don’t talk to strangers
    * Dit is eigenlijk een regel die niet bij alle definities van GRASP aanwezig is, maar volgens de docent wel relevant!
    * Als een object B kennis heeft van de interne structuur van andere objecten, dus bij voorbeeld een methode wilt aanroepen op een object A waarmee het niet direct is geassocieerd, is dat high coupling. Hoe voorkom je dat? Ken die verantwoordelijkheid toe aan object C waarmee B wel is geassocieerd, en die zelf wel direct met A is verbonden en de verantwoordelijkheid kan delegeren aan A.

Omdat we in ons project een groot aantal klassen en verantwoordelijkheden hebben kwam dit GRASP principe goed van pas. Zo hebben we veel klassen geïntroduceerd die elk een specifieke coherente taak hebben. Zoals de spelstukken, het bord, de selectiemanager, het venster, het spelobject, etc. Hiervoor hebben we de Low Coupling en High cohesion patterns gebruikt.

Figuur Type hierarchy

Er zijn veel klassen die information expert zijn. Zo weet elk individueel schaakstuk welke zetten hij kan doen omdat hij de informatie heeft van de locatie waarop hij staat en welke zetten hij van daaruit kan maken. Hiervoor vraagt hij overigens wel informatie op aan bijvoorbeeld het bord, deze weet weer welke andere stukken op die bepaalde locatie staan.

Polymorphism hebben we ook regelmatig gebruikt in situaties waar het gedrag per type verschilt. Dit is bijvoorbeeld het geval bij schaakstukken. Elk schaakstuk vertoond ander gedrag, een koning gedraagt zich beduidend anders als een knight. Zo heeft elk schaakstuk ook andere bewegingen die hij kan maken. Dit was ook het geval bij de schaakengines. Elke schaak engine is anders in het maken van keuzes. Zie figuur 5 voor een overzicht van de type hiërarchie van alle klassen in het schaakspel, alleen de meest relevante klassen zijn afgebeeld.

Pure fabrication is toegepast bij een klasse die simpele 3D objecten, zoals vierkantjes en planes[[1]](#footnote-1), tekent. Deze verantwoordelijkheid was aan geen enkele klasse logisch toe te kennen. Hiervoor hebben we de klasse *ShapeUtil* gemaakt en dient als utility klasse voor grafische objecten.

De klasse *ObjectPicker* is een toepassing van het Controller pattern. Deze klasse vangt de invoer handelingen af en coördineert deze. Onder deze klasse zit ook nog een structuur welke bepaald wat de huidige status van het spel is en filtert als het ware invoer.

# Design patterns

Ondanks design patterns nog niet zijn besproken, hebben we er wel een paar toegepast. Dit onderdeel van het rapport legt kort de situaties uit waarvoor we de bepaalde patterns hebben gebruikt:

1. State pattern
   * Dit pattern hebben we gebruikt om de status van het spel bij te houden. In een instantievariabele van *ObjectPicker* zit een object dat als superklasse *SelectionState* heeft. Een bepaalde selectionstate kijkt naar geldige invoer bij bepaalde situaties. Als de gebruiker bijvoorbeeld een zet moet kiezen dan wordt eerst de selection state *SelectPieceState* ingeladen. Zodra de gebruiker daadwerkelijk een object selecteert, vervang SelectPiece zichzelf met een andere status, namelijk *SelectSquareState*. Als de gebruiker iets anders dan een Schaakstuk van zijn kleur kiest negeert *SelectPieceState* dat gewoon. *SelectSquareState* reageert alleen maar op selecties van squares. Als de Computer aan zet is wordt *SelectIgnore* state ingeladen. De gebruiker kan dan geen selecties doen. Deze manier maakt het heel flexibel om de complexiteit van selectie verdelen over in meerdere klassen zonder dat de *ObjectPicker* daar weet van heeft. Ook kunnen er later andere states worden toegevoegd zonder code wijzigingen.
2. Command Pattern
   * De commando’s zijn geïmplementeerd volgens dit pattern. Bijvoorbeeld alle schaakstukken kunnen een set van commando’s teruggeven die op dat moment geldig zijn voor hun. Dit commando kan later uitgevoerd worden. Na het uitvoeren van een commando wordt hij op een stack geplaatst. Bij het undo’en van een bepaald commando wordt hij van de stack gehaald word de methode Undo op hem aangeroepen. Hij bevat dus alle informatie die nodig is om het commando uit te voeren.
3. Singleton
   * De manager die het *Game* object en het venster bevat is een singleton. Er is maar één instantie van dit object. Door gebruik te maken van *Manager*.*getInstance*() kunnen clients de instantie opvragen.
4. Strategy
   * Om verschillende schaakstrategieën te implementeren binnen het spel hebben we het Strategy pattern gebruikt. De bot speler kan een strategie toegekend krijgen welke voor hem de beste zet bepaald. De speler zelf weet niks van het algoritme zelf af, en het is interessant en leuk als er meerdere verschillende strategieën zijn waaruit de speler kan kiezen.
5. Proxy Pattern
   * Hoewel we niet zelf een proxy hebben geschreven, maken we er wel gebruik van. Zo kan het bord een set teruggeven van alle schaakstukken van een bepaalde speler op het bord. Het is niet gewenst dat de gebruiker deze collectie aan zou kunnen passen. Hiervoor hebben we de volgende oplossing, wat in feite een proxy is, gebruikt:
     1. *Collections*.*unmodifiableSet(pieces);*

# Logboek

In dit hoofdstuk zijn de activiteiten die door ons verricht zijn, summier beschreven. Het betreft hier zowel de verkregen lesstof als het verrichten van werkzaamheden.

## Week 1 (14 april 2009)

Deze week zijn er groepen gevormd m.b.t. de case en is er een onderwerp gekozen. Christophe en Jeffrey hebben gekozen voor een Sudoku Solver: een door de docent voor gesteldt idee.

Sudoku Solver is een programma waarmee een gebruiker Sudoku puzzels, deels geautomatiseerd, kan oplossen. De gebruiker moet in staat zijn om handmatig getallen, of kandidaat-getallen in een vak in te vullen, of uit te sluiten. Het programma moet ook automatisch kandidaten kunnen weergeven, en volgens bepaalde strategieën kunnen zoeken naar getallen.

In deze week hebben Christophe en Jeffrey al nagedacht over de functionele specificatie van de Sudoku solver. Het gaat hierbij om hi-level use cases en GUI prototypes.

## Week 2 (21 april 2009)

Deze week hebben Christophe en Jeffrey de high-level use case en een GUI prototype aan de docenten laten zien. Echter waren we van mening dat een Sudoku Solver geen echte uitdaging is en hebben daarom gevraagd of het goed is of we mogen overschakelen naar een ander onderwerp, namelijk een 3D schaakspel d.m.v. OpenGL. De docenten vonden het een goed idee, waardoor we de ideeen voor de Sudoku Solver hebben laten vallen voor onze nieuwe uitdaging, Chess3D.

Deze week zijn we vervolgens gezamelijk het domeinmodel in de les gaan schetsen.

## Week 3 (28 april 2009)

Deze week hebben we tijdens de les het gehad over -contracten tussen GUI en domeinmodel op basis van de use cases. Deze moeten worden gemaakt voor een aantal methoden voor de volgende les. Deze dienen in week 6 (na de meivakantie) te worden reviewed door de docent.

Vervolgens hebben Christophe en Jeffrey zich bezig gehouden met het realiseren van diverse Proof-Of-Technologies en heeft Christophe het domeinmodel bijgeschaafd.

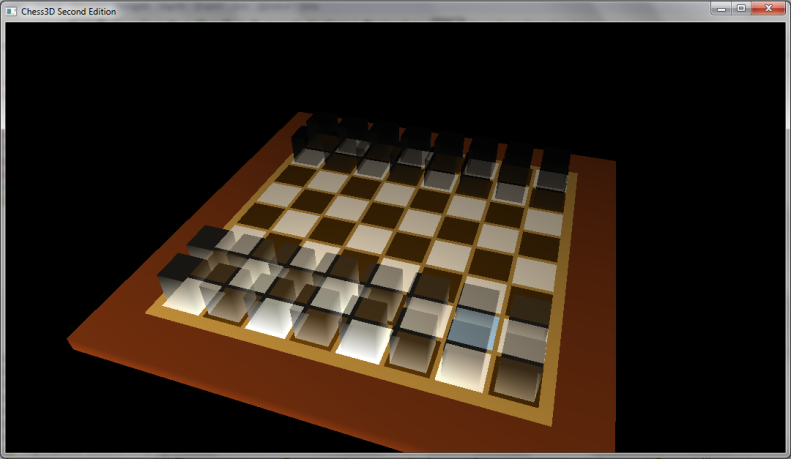
## Week 6 (19 mei 2009)

In deze week zijn er presentaties gehouden over de gekozen Proof-Of-Technology. Christophe en Jeffrey hebben aan de klas laten zien hoe de rendering in JOGL te werk gaat en hoe dit gebruikt zou kunnen worden in Chess3D.

In deze les is er verder gewerkt aan de operation-contracten en hebben Christophe en Jeffrey de Proof-of-Technologies verder gerealisserd.

## Week 7

In deze week was er geen sprake van les, maar vakantie. Christophe en Jeffrey hebben wel een eerste prototype ontworpen en gerealiseerd (zie ).



Figuur Impressie eerste prototype

## Week 8 (2 juni 2009)

In deze week is er tijdens de les stil gestaan bij Threads. Er is verteld wat de voordelen van threads zijn en hoe je deze kunt toepassen voor de case.

Tijdens deze les zijn we begonnen aan het ontwikkelen van communicatie diagrammen op basis van de operation-contracten en aan het uitbreiden van ons domeinmodel naar een detailontwerp.

De realisatie van het spel door Jeffrey en Christophe gaat vlot. Zie , welke echter het eindresultaat betreft, maar er nauwelijks verschil te zien is met het resultaat van deze week.



Figuur Impressie eind product

## Week 9 (9 juni 2009)

In deze week hebben we tijdens de les de methoden in GRASP geleerd. Daarnaast is er weer verder gewerkt aan de code, vooral aan de grafische schil. Er is ook begonnen aan het algoritme voor de computergestuurde speler en de strategieën die deze kan gebruiken.

## Week 10 (16 juni 2009)

Christophe heeft zich allereerst bezig gehouden met een aantal kleine puntjes op de i te zetten in de programmacode.

Jeffrey heeft zich bezig gehouden met een communicatiediagram en het conceptueel class diagram in de correct maar originele staat te verkrijgen.

Casper heeft de usecase diagrammen tesamen gemaakt met enkele communicatie diagrammen, een automatisch gegenereerde class diagram en een begin van het logboek.

# Bijlage I Use case diagrammen

|  |  |
| --- | --- |
|  |  |
|  |  |

# Bijlage II Use cases

**Use Case 1: *Speler stukken toevoegen op boord***

Betrokken actor(en):

Gebruiker

Beschrijving

De use case begint als een als er een nieuw-, oud spel of een ander boord wordt geladen. Hierbij dient het boord, de speler en de locatie van het speler stuk meegegeven te worden.

Details: main success scenario:

Als alles goed verloopt, zal het proces als volgt worden doorlopen:

1. Gebruiker kiest voor:

a. Nieuw / oud spel inladen.

b. Nieuw boord inladen.

1. Systeem reageert:

a. Er wordt een nieuw / oud spel ingeladen.

b. Er wordt een nieuw boord ingeladen.

1. Systeem reageert:

a. Initialiseren vierkanten.

b. Initialiseren speler stukken beide spelers.

c. Initialiseren grafische schil boord.

1. De gebruiker wordt op de hoogte gesteld dat de hele systeemoperatie voltooid is.

Systeem operaties, afgeleid van het main success scenario

1. a.

i. new Game(new Player(), new Player());

ii. new Game(settings.loadPlayer, settings.loadPlayer());

b. changeBoard(game.getScene(), this);

1. a. new Board(game);

b. new BoardDisplayObject(game.getScene(), this);

1. a. initSquares();

b. initPieces();

c. new BoardDisplayObject(game.getScene(), this);

1. updateGraphics();
2. MessageBox.Show(“Uw speler stukken zijn succesvol op het boord geplaatst.”, “Uw speler stukken zijn gereed”);

**Use Case 2: *Spel spelen***

Betrokken actor(en):

Gebruiker

Beschrijving

De use case begint als er een nieuw of oud spel geladen moet worden. Hierbij dienen de participanten / spelers meegegeven te worden.

Details: main success scenario:

Als alles goed verloopt, zal het proces als volgt worden doorlopen:

1. Gebruiker kiest voor:

a. Nieuw spel inladen.

b. Oud spel inladen.

1. Systeem reageert:

a. Er wordt een nieuw spel ingeladen.

b. Er wordt een oud spel ingeladen.

1. De gebruiker wordt op de hoogte gesteld dat de systeemoperatie voltooid is.

Systeem operaties, afgeleid van het main success scenario

1. a. menu.newGame();

b. menu.loadGame();

1. a. new Game(new Player(), new Player());

b. new Game(settings.loadPlayer, settings.loadPlayer());

1. MessageBox.Show(“Spel geladen, succes! “Spel geladen”);

**Use Case 3: *Zet terugdraaien***

Betrokken actor(en):

Gebruiker

Beschrijving

De use case begint als de speler een zet wilt terugdraaien.

Details: main success scenario:

Als alles goed verloopt, zal het proces als volgt worden doorlopen:

1. Gebruiker kiest voor de optie ‘zet terugdraaien’.
2. Systeem reageert:

a. Checken of er een voorgaande zet aanwezig is.

b. Terughalen vorige zet.

1. De gebruiker wordt op de hoogte gesteld dat de systeemoperatie voltooid is.

Systeem operaties, afgeleid van het main success scenario

1. btnRewindMove.click();
2. a. getOldSquare();

b. rewind();

1. MessageBox.Show(“Uw vorige zet is succesvol teruggedraaid.”, “Voorgaande zet teruggedraaid”);

**Use Case 4: *Zet uitvoeren***

Betrokken actor(en):

Gebruiker

Beschrijving

De use case begint als de speler een speler stuk wilt verzetten.

Details: main success scenario:

Als alles goed verloopt, zal het proces als volgt worden doorlopen:

1. Gebruiker kiest voor speler stuk verzetten.
2. Systeem reageert:

a. Checken of de zet al uitgevoerd is.

b. Uitvoeren zet.

1. De gebruiker wordt op de hoogte gesteld dat de systeemoperatie voltooid is.

Systeem operaties, afgeleid van het main success scenario

1. objectPicker.mouseDown(MouseEvent mouseEvent);
2. a. move.isExecuted();

b. setPiece(Piece piece);

1. MessageBox.Show(“Uw speler stuk is succesvol op het boord verplaatst.”, “Speler stuk verzet”);

**Use Case 5: *Spelerstuk zetten op vierkant***

Betrokken actor(en):

Gebruiker

Beschrijving

De use case begint als de speler een speler stuk wilt verzetten naar een ander vierkant op een boord.

Details: main success scenario:

Als alles goed verloopt, zal het proces als volgt worden doorlopen:

1. Gebruiker observeert de mogelijkheden om zijn / haar speler stuk naar een ander vierkant op het boord te verzetten.
2. Systeem reageert:

a. Berekent de mogelijke zetten van een bepaald speler stuk op het boord.

b. Notificeert de gebruiker daarover in grafisch aspect, door de mogelijke zetten van een spelerstuk op een vierkant te highlighten.

1. Gebruiker verplaats zijn / haar speler stuk naar één van de mogelijke verzetlocaties (vierkant) op het boord.
2. a. Systeem checkt of zet al eerder uitgevoerd is.

b. Systeem verplaatst het speler stuk naar de opgegeven locatie.

1. De gebruiker wordt op de hoogte gesteld dat systeemoperatie voltooid is.

Systeem operaties, afgeleid van het main success scenario

1. objectPicker.mouseIsOverAndFirstObject(DisplayObject displayObject);
2. a. piece.getAvailableMoves();

b. pieceDisplayObject.setColor(GL gl);

1. objectPicker.mouseDown(MouseEvent mouseEvent);
2. a. move.isExecuted();

b. setPiece(Piece piece);

1. MessageBox.Show(“Uw speler stuk is succesvol op het boord verplaatst.”, “Speler stuk verzet”);

# Bijlage III Operation Contracts

/\*\*

\* Deze klasse is verantwoordelijk voor het boord, de spelomgeving en beide spelers.

\* Het managed bovenstaande instanties in het spel.

\*/

public class Game

{

public static final int COLOR\_BLACK = 1;

public static final int COLOR\_WHITE = 2;

private Board board;

private GameScene scene;

private Player playerWhite;

private Player playerBlack;

/\*\*

\* Initialiseert het boord, de spelomgeving en beide spelers in het spel.

\*

\* @param playerWhite

\* de speler met een witte kleur.

\* @param playerBlack

\* de speler met een zwarte kleur.

\* @throws NullPointerException

\* indien playerWhite of playerBlack null is.

\* @throws IllegalStateException

\* indien playerWhite gelijk is aan playerBlack.

\*/

public Game(Player playerWhite, Player playerBlack)

{

this.playerWhite = playerWhite;

this.playerBlack = playerBlack;

playerWhite.setColor(COLOR\_WHITE);

playerBlack.setColor(COLOR\_BLACK);

init();

}

/\*\*

\* Initialiseert het boord en de spelomgeving in het spel.

\* Deze methode wordt expliciet alleen door de constructor aangeroepen.

\*/

private void init()

{

scene = new GameScene(this);

board = new Board(this);

}

/\*\*

\* Retourneer de spelomgeving waarvoor het spel geldt.

\*

\* @return de spelomgeving waarvoor het spel geldt.

\*/

public GameScene getScene()

{

return scene;

}

/\*\*

\* Retourneer de speler a.d.v. de opgegeven kleur, waarvoor het spel geldt.

\*

\* @param playerColor

\* de kleur die voor een speler geldt.

\* @throws IllegalArgumentException

\* indien playerColor niet van het datatype integer is.

\* @throws IllegalArgumentException

\* indien playerColor niet COLOR\_BLACK of COLOR\_WHITE is.

\* @return de speler, waarvoor het spel geldt.

\*/

public Player getPlayer(int playerColor)

{

if(playerColor == COLOR\_BLACK)

return playerBlack;

else if(playerColor == COLOR\_WHITE)

return playerWhite;

else

throw new IllegalArgumentException("Given playerColor is unsupported.");

}

/\*\*

\* Retourneert de tegenstander van de opgegeven speler.

\* Als de speler een ongeldige kleur heeft, retourneer dan null.

\*

\* @param player

\* de speler, waarvan de tegenstander opgevraagd dient te worden.

\* @return de tegenstander van de speler.

\*/

public Player getOpponentPlayer(Player player)

{

if(player == playerBlack)

return playerWhite;

else if (player == playerWhite)

return playerBlack;

else

return null;

}

}

1. Een vlak in een 3D ruimte dat geen breedte heeft [↑](#footnote-ref-1)