|  |
| --- |
|  |
| Othello-Net i Java |
| - En design- og implementeringsproces |
|  |
|  |
|  |

Roskilde Universitet

Datalogi, Hus 43.2

B1-modulet

6. semester



**Christian Casper Hofma** – cchofma@ruc.dk

Vejleder:

**Keld Helsgaun –** keld@ruc.dk

14. december 2011

[1 Indledning 4](#_Toc311894263)

[1.1 Læsevejledning 5](#_Toc311894264)

[1.2 Målgruppe 5](#_Toc311894265)

[1.3 Ordforklaring 5](#_Toc311894266)

[2 Installationsvejledning 6](#_Toc311894267)

[2.1 Systemkrav 6](#_Toc311894268)

[2.1.1 Software 6](#_Toc311894269)

[2.1.2 Hardware 6](#_Toc311894270)

[2.2 Initiering af Othello-Nets server og klient 6](#_Toc311894271)

[3 Brugervejledning 7](#_Toc311894272)

[3.1 Beskrivelse af brugergrænsefladens elementer 7](#_Toc311894273)

[3.1.1 Serveren 7](#_Toc311894274)

[3.1.2 Klienten 8](#_Toc311894275)

[3.2 Start af et spil 8](#_Toc311894276)

[3.3 Spillereglerne 10](#_Toc311894277)

[3.4 Et spilscenarie 10](#_Toc311894278)

[3.5 Fejlmeddelelser 11](#_Toc311894279)

[3.5.1 Serveren 11](#_Toc311894280)

[3.5.2 Klienten 11](#_Toc311894281)

[4 Problemanalyse 13](#_Toc311894282)

[4.1 Problemstillingen 14](#_Toc311894283)

[4.2 Kravspecifikation 14](#_Toc311894284)

[4.2.1 Funktionelle krav 14](#_Toc311894285)

[4.2.2 Tekniske og stabilitets krav 15](#_Toc311894286)

[5 Projektforløb 16](#_Toc311894287)

[5.1 Den objektorienterede designproces 16](#_Toc311894288)

[5.1.1 Iterativ udvikling 16](#_Toc311894289)

[5.1.2 Throwaway-prototyping 17](#_Toc311894290)

[5.2 Projektforløbets faser 17](#_Toc311894291)

[5.2.1 Indledende fase – Udarbejdelse af en kravspecifikation 17](#_Toc311894292)

[5.2.2 Mellemliggende fase – Revidering af kravspecifikation 18](#_Toc311894293)

[5.2.3 Afsluttende fase – Implementering af Othello-Net 18](#_Toc311894294)

[5.3 Delkonklusion 18](#_Toc311894295)

[6 Teori 19](#_Toc311894296)

[6.1 Designmønstre 19](#_Toc311894297)

[6.1.1 Model-View-Controller 19](#_Toc311894298)

[6.1.2 En MVC-variation: Observer-mønsteret 20](#_Toc311894299)

[6.1.3 MVC i Othello-Net 21](#_Toc311894300)

[6.2 Server-Klient netværksstrukturer 22](#_Toc311894301)

[6.2.1 Tynde og tykke klienter/servere 22](#_Toc311894302)

[6.2.2 RMI og Sockets i Java 23](#_Toc311894303)

[6.3 Netværksstrukturen i Othello-Net 24](#_Toc311894304)

[6.3.1 Sockets 25](#_Toc311894305)

[6.4 Delkonklusion 25](#_Toc311894306)

[7 Programdokumentation 26](#_Toc311894307)

[7.1 Othello-Nets anvendelse af NetOthello 26](#_Toc311894308)

[7.2 Klasserne i Othello-Net 26](#_Toc311894309)

[7.2.1 View/Controller (Observer) 27](#_Toc311894310)

[7.2.2 Controller/Klient 27](#_Toc311894311)

[7.2.3 Model (Observable) 27](#_Toc311894312)

[7.2.4 Serveren 27](#_Toc311894313)

[7.3 Klassernes interaktion 27](#_Toc311894314)

[7.4 View og Controller-klasserne 28](#_Toc311894315)

[7.4.1 BoardView-klassen 28](#_Toc311894316)

[7.4.2 GUI-klassen 31](#_Toc311894317)

[7.4.3 Styrker og svagheder 32](#_Toc311894318)

[7.5 Controller og Klient-klassen 33](#_Toc311894319)

[7.5.1 OthelloClient-klassen 33](#_Toc311894320)

[7.5.2 Styrker og svagheder 34](#_Toc311894321)

[7.6 Model 35](#_Toc311894322)

[7.6.1 BoardModel-klassen 35](#_Toc311894323)

[7.6.2 Styrker og svagheder 37](#_Toc311894324)

[7.7 Serveren 38](#_Toc311894325)

[7.7.1 Server-klassen 38](#_Toc311894326)

[7.7.2 Connect-klassen 38](#_Toc311894327)

[7.7.3 ConnectThread-klassen 39](#_Toc311894328)

[7.7.4 Styrker og svagheder 40](#_Toc311894329)

[8 Afprøvning af program 41](#_Toc311894330)

[8.1 Test af programmellets struktur og funktionaliteter 41](#_Toc311894331)

[8.2 Resultatet af afprøvningerne 41](#_Toc311894332)

[9 Diskussion 43](#_Toc311894333)

[9.1.1 Projektforløbet 43](#_Toc311894334)

[9.1.2 Brugergrænsefladen 44](#_Toc311894335)

[9.1.3 Designet 44](#_Toc311894336)

[9.1.4 Netværksstrukturen 44](#_Toc311894337)

[10 Konklusion 46](#_Toc311894338)

# Indledning

Det følgende afsnit indeholder en kort introduktion til emnet, herunder hvad spillet og rapporten grundlæggende omhandler, dets status samt de centrale konklusioner der er blevet erfaret i forlængelse heraf.

Dette projekt har omhandlet spillet *Othello-Net*. *Othello-Net* er et spil eller programmel som har til formål at gøre to normalbrugere i stand til at spille brætspillet Othello over et netværk.

I denne rapport vil jeg prøve at gøre rede for designet og den efterfølgende implementering af *Othello-Net*. Mere præcist vil fokusset ligge på de beslutninger der ligger bag valget af specielt det overordnede design samt de følgevirkninger det har haft på såvel opfyldelsen af det overordnede mål og på selve implementeringen af dette projektforløbs endelige prototype af *Othello-Net*. Rapporten vil derfor hovedsageligt have fokus på at klarlægge processen bag dette projekts endelige implementering af *Othello-Net*, altså hvorfor og hvordan jeg er nået frem til netop dette programmel og om hvorvidt det har opfyldt målet om at gøre to normalbrugere i stand til at spille Othello over et netværk. I forlængelse af dette vil jeg tillige berører mere programmeringsspecifikke emner, herunder de generelle principper for objektorienteret design og i hvor høj grad de er blevet overholdt under implementeringen af spillet.

Overordnet set opfylder *Othello-Net* de funktionaliteter som der er blevet specificeret i starten og løbende igennem projektforløbet. Dette indebærer, som nævnt, at to normalbrugere kan spille Othello på to forskellige computere over et netværk, i det tilfælde de anviste instruktioner fra brugervejledninger er blevet fulgt.

Der skal dog knyttes visse forbehold til denne konklusion. Brugervejledningen opstiller nogle forholdsvist kunstige og for normalbrugere til tider uforståelige krav specielt i opstartsfasen hvor det er nødvendigt at initiere serveren *før* klienten.I denne fase er der tillige en risiko for at serveren ikke vil være i stand til at lukke på normalvis. Denne fejl vil opstå i tilfælde af at brugeren vil lukke serveren inden der er tilkoblet nogle spillere til den. Der er altså en risiko for at spillet kan gøre systemet, hvorpå spillet kører, ustabilt og derfor påvirke det i unødig grad.

Opsummerende dokumentere denne rapport design- og implementeringsprocessen af spillet *Othello-Net* som overordnet set har resulteret i en opfyldelse af de specificerede krav der et blevet opstillet for dette projektforløb.

## Læsevejledning

Det anbefales at rapporten læses i den kronologiske rækkefølge som anvist i indholdsfortegnelsen, da den overordnede konklusion og rationalet bag afhænger af de enkelte kapitler. Hvis ens intention med denne rapport er at undersøge hvordan man afvikler spillet, kan man nøjes med at læse installations- og brugervejledningen.

## Målgruppe

Målgruppen for denne rapport er fortrinsvist rettet mod personer med viden inden for det datalogiske fagområde, da mange fagbegreber anvendes. En undtagelse af det førnævnte er brugervejledningen hvis indhold hovedsageligt er tilsigtet personer som alene har til formål at afvikle spillet *Othello-Net.*

## Ordforklaring

***Othello-Net* –** Navnet på dette projekts endelige prototype (Bilag A: Othello-Net).

***NetOthello* –** Den tidligere version af Othello fra 1996 (Bilag B: *NetOthello*).

**Normalbrugere –** Brugere som besidder normale computerkompetencer.

# Installationsvejledning

## Systemkrav

For at være i stand til at kunne køre programmet skal den pågældende hardwareenhed, eksempelvis en PC, opfylde en række minimumskrav i henhold til soft- og hardware. Disse er som følgende:

### Software

* OS: Windows, Linux eller lignende.
* Java Runtime Environment (JRE) 1.5 eller højere.

### Hardware

* Mus eller touch-input til navigation i menuen samt valg og flytning af spillebrikker.
* Skærm.
* 128 MB RAM
* 98 MB fri diskplads

Programmet er af formatet JAR og kan derfor startes uafhængigt af en compiler. (Bilag G: *Cd-rom*)

## Initiering af Othello-Nets server og klient

Programmet består overordnet set af to dele, en server- og en klientdel. Disse kan startes ved henholdsvist at vælge JAR-filerne *”Othello-Net-Server.jar”* og *”Othello-Net-Client.jar”* enten på normalvis eller igennem en konsol som for eksempel Kommandoprompt i Windows. For at vælge filerne ved brug af sidstnævnte kan man gøre brug af følgende fremgangsmåde:

1. Vælg stien hvor JAR-filerne er placeret ved brug af kommandoen ”*cd”*. For eksempel: ”*C:\Users\Administrator> cd C:\Othello-Net*”.
2. For at starte den givende fil, brug kommandoen ”*java jar-”*. For eksempel: *”C:\Othello-Net> java jar- Othello-Net-Server.jar”*.

Det er *vigtigt* at server-delen startes og initieres før klient-delen.

# Brugervejledning

Brugervejledningen indleder med at redegøre for brugergrænsefladen hvorefter en beskrivelse af hvordan spillet startes vil følge. I forlængelse heraf, vil reglerne for Othello og et spilscenarie blive fremstillet. Afslutningsvist bliver eventuelle fejlmeddelelser som brugeren kan risikere at støde på ved starten af et spil blive gennemgået.

## Beskrivelse af brugergrænsefladens elementer

*Othello-Net* består af to dele, en server- og klientdel og har derfor to brugergrænseflader. Begge dele skal startes før det vil være muligt at afvikle et spil, hvorfor deres brugergrænseflader nu vil blive beskrevet.

### Serveren

Serverens grænseflade består af tre interaktive elementer mere specifikt, et input-felt og to knapper kaldt henholdsvis *Start Server* og *Close Server* (se figur 3.1)*.*

I inputfeltet kan angives hvilken port serveren skal lytte på. Standard porten er angivet til at være port 4444 da klienten og dermed spilleren forbinder sig til serveren igennem denne port

*Start Server*-knappen kan først anvendes når et port-nummer er angivet. Når serveren er initieret vil hele vinduet forblive inaktivt indtil to spillere er tilsluttet serveren, hvorefter *Start Server*-knappen ikke længere kan benyttes.

*Close Server* knappen bruges når serveren skal lukkes. Den er ligeledes inaktiv indtil to spillere er tilsluttet serveren.

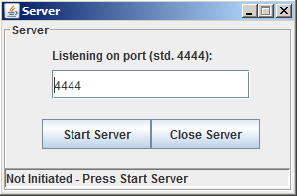
### Klienten

Klient-delen er inddelt i fire området hvoraf to, spillebrættet og chat-delen, er interaktive og to er statiske, *Game State* og *Moves Made*. Derudover befinder der sig ligeledes en Fil-menu samt et status område som løbende opdaterer netværks-forbindelsen status. Inde under fil-menuen kan brugeren vælge mellem: *New Game, Resign* og *Exit Othello*. (se figur 3.4)

## Start af et spil

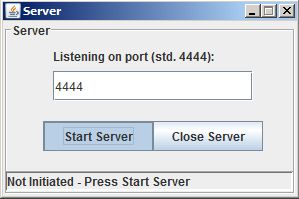
Et spil startes ved at initiere en server som spillerne kan tilslutte sig til, hvorfor serveren skal startes først. Dette gøres på følgende vis:

1. Indtast et port nummer. Det anbefales at anvende port 4444, da denne er standard for *Othello-Net*-klienterne. Hvis en port allerede er angivet kan dette trin springes over.



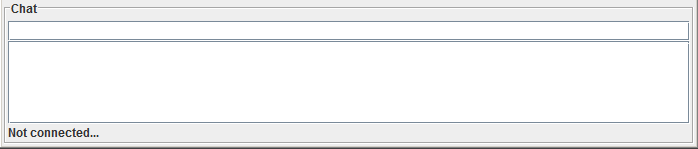
*Figur 3.1: Server-vindue.*

1. Tryk på knappen, *Start Server*. Hvis server knappen ser ud som illustreret i figuren nedenunder, kan *Othello-Net*-klienten nu startes ved dobbeltklik.

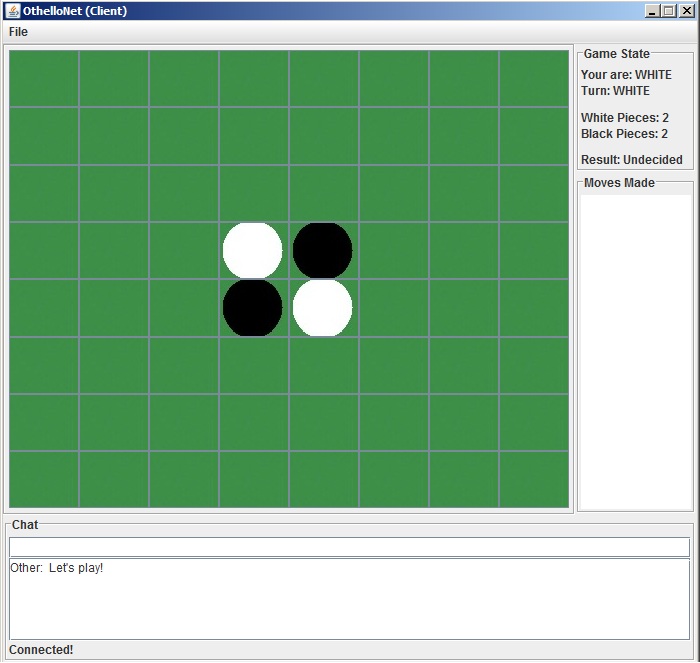


*Figur 3.2: Klienten kan nu startes.*

1. Klienten, hvis startet korrekt, skulle nu kunne ses på skærmen og vise ”*Not Connected…”* i statusbaren nederst i venstre hjørne (se figur 3.3).



*Figur 3.3: Billede af spillet når det ikke er tilsluttet en server eller mangler en anden spiller.*

1. For at starte et nyt spil vælg *File*, øverst i venstre hjørne, og dernæst *New Game*.
2. Spillet er nu startet og vinduet viser nu fire spillebrikker i midten af spillepladen. 

*Figur 3.4: Starten af et spil...*

Hvis opstarten ikke forløber som angivet ovenfor henvises til afsnittet om fejlmeddelelser.

## Spillereglerne

I Othello, eller Reversi, som det også bliver kaldt, spiller man normalt med en spilleplade som har 8x8 felter og de respektive brikker har henholdsvis en sort og hvid side. Når spillerne er blevet tildelt en farve startes spillet ved at lægge fire brikker, to af hver farve, i midten af spillepladen.

For spillerne gælder det nu om at placere ens brik således at der ved siden af, eller diagonalt for den, befinder sig en brik af modsatte farve samtidig med, at der for enden af rækken er en brik i ens egen farve. Det gælder derfor om at omringe modstanderens farve med ens egen farve og hvis det i så fald lykkes vil de omringede brikker blive vendt om og skifte farve. Spillerne skiftes således til at anbringe deres respektive brikker på spillepladen indtil de førnævnte betingelser ikke længere kan overholdes. Vinderen er den spiller som ender med at have flest brikker på spillepladen. Er der lige mange brikker ender det uafgjort.

## Et spilscenarie

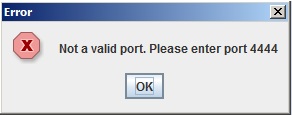
Et spil startes ved at én af brugerne initierer serveren, efter de fremstillede anvisninger tidligere i brugervejledningen. Serveren vil nu vente på at to klienter bliver tilsluttet på den port der er angivet. Klienterne tilsluttes ved at starte dem op mens serveren stadig lytter og hvis det i så fald lykkes vil klienterne nu være i stand til at starte et nyt spil og bruge chat-funktionen. Serveren vil tillige indikere en forbindelse mellem de to klienter ved at opdatere status-feltet med teksten *”Maksimum limit of players on server reached!”* for derefter at gøre *Start Server*-knappen inaktiv. Ved at vælge *New Game*-knappen, under menuen *File* vil begge spillepladerne vise fire spillebrikker og *Game State*-området samt netværksstatussen vil blive opdateret med deres respektive informationer. Herefter kan spillerne nu interagere med spillepladen i det omfang som de førnævnte regler tillader.

## Fejlmeddelelser

Under afviklingen af spillet kan brugerne risikerer at støde ind i fejlmeddelelser. Risikoen for fejl vil som regel være størst i forbindelse med opstarten af serveren og spillet (klienten) eller nærmere bestemt når de skal synkroniseres med hinanden. Disse fejlmeddelelser og løsningen til dem vil blive forsøgt beskrevet og uddybet nærmere i følgende afsnit.

### Serveren

1. Hvis brugeren forsøger at angive en anden port end standard porten 4444, vil man blive meddelt: *”Not a valid port. Please enter port 4444.”* (se figur 3.5.)



*Figur 3.5: Fejl når der indtastes bogstaver i stedet for tal.*

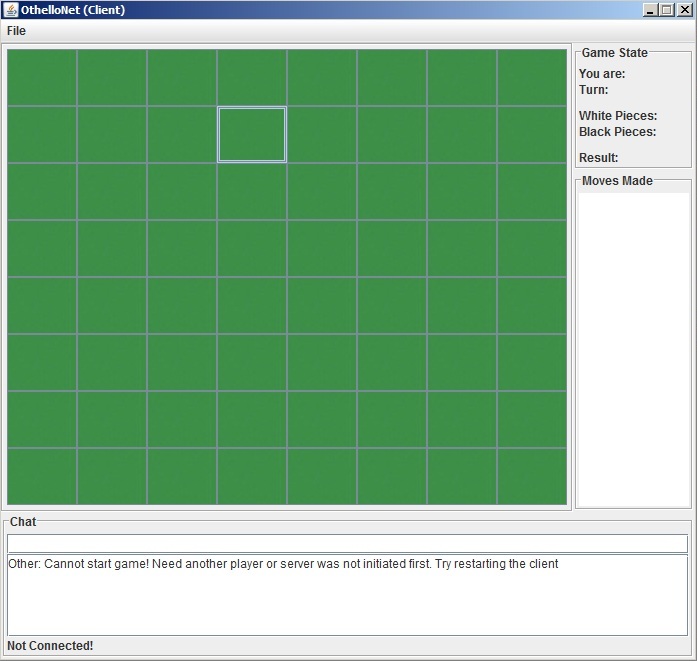
Løsningen vil typisk være at angive et gyldigt port-nummer. Det anbefales at man bruger port-nummeret 4444, da der er igennem denne klienterne normalt vil forbinde sig til serveren.

1. Hvis brugeren indtaster et andet nummer end standard-porten, 4444, vil den blive ved med at lytte indtil to spillere er tilsluttet.

For at stoppe serveren bliver man nødt til at lukke den ved hjælp at *force close* for eksempel ved at bruge *ctrl + alt + delete* eller noget lignende for andre operativsystemer.

### Klienten

1. Hvis brugeren ikke har startet en server eller ikke kan finde en på klientens specificerede port, normalt port 4444, og samtidig prøver at starte et spil, vil man igennem chat-vinduet få oplyst: *”Cannot start game! Need another player or server was not initiated first. Try restarting the client”* (se figur 3.6).



*Figur 3.6: Screenshot når man ikke har forbindelse til en server eller lignende.*

Løsningen vil typisk være at genstarte klienten og vente på at man er sikker på at der startet en server på standard porten, 4444.

# Problemanalyse

I dette afsnit vil jeg gøre rede for den endelige udformning af projektets problemstilling. Til at forstå rationalet bag denne, vil jeg først præsenterer de centrale og overordnede til- og fravalg der har ført til problemstillingens nuværende form hvorefter en fremstilling af den endelige problemstilling vil følge.

Overvejelserne omkring dette projekt begyndte indledningsvist med den erkendelse at jeg ville prøve at designe og implementerer et program i Java i henhold til de specificerede krav for datalogis bachelor-modul, B1:

*”Modulet skal give den studerende erfaring i at planlægge, implementere, afprøve og dokumentere middelstore programmeringsopgaver ved anvendelse af et højere, generelt programmeringssprog. ”* (Internetkilde 2)

Denne forholdsvist abstrakte formulering til de officielle krav gav mig rig mulighed for selvstændigt at vælge en programmeringsopgave hvis emne og sværhedsgrad ville tiltale og motiverer mig i tilstrækkelig grad. Jeg endte med at vælge at programmere en netværksversion af spillet Othello eller Reversi som det også bliver kaldt. Årsagen til dette var at jeg forholdsvist tidligt i projektforløbet fik inspiration af en ældre Java-version af Othello ved navn *NetOthello* fra 1996 som også indeholdte en netværksfunktionalitet (Fan, 1996, p. 619ff & Bilag B: *NetOthello*) . Med inspiration fra denne ældre version af Othello har fokusset for dette projekt hovedsageligt været på netværksdelen samt designet af programmellet i al almindelighed. Implikationen af dette er at jeg i mindre grad har beskæftiget mig med selve spildelen, nærmere bestemt spilalgoritmerne samt brugergrænsefladen. Mere konkret vil det betyde at jeg ikke vil inddrage større teoretiske refleksioner vedrørende de dele af spillet men blot forholde mig til dem i et mere praktisk programmerings øjemed. Derfor genanvender spillet også dele af søge-algoritmerne fra det tidligere spil, idet fokusset for dette projekt i mindre grad har været på spildelen hvilket også skal ses i lyset min manglende erfaring med udviklingen af programmel generelt hvorfor denne afgræsning har været nødvendig i et tidsmæssigt henseende, jf. Forord. En udspecificering af hvilke dele der er blevet genanvendt vil blive uddybet i kapitlet vedrørende Programdokumentationen.

## Problemstillingen

Projektets centrale problemstilling(er) er at:

*”… designe og implementerer en version af spillet Othello i en sådan grad at det vil være muligt for to normalbrugere at kunne afvikle et spil i henhold til de gældende regler for Othello.”*

For tilnærme sig en bedre idé af projektets problemstilling er der på baggrund af den ovenstående problemstilling udformet en uddybende kravspecifikation som nu vil efterfølge.

## Kravspecifikation

I kravspecifikationen vil den ovenstående problemstilling af projektet blive uddybet. Kravspecifikationen er under projektforløbet blevet revideret løbende. Den vil så vidt muligt kun omhandle de funktionelle krav som spillet vil have som mål at overholde, det indebærer nærmere, *hvad* der skal være med og ikke *hvordan* de skal implementeres (Bell, 2005, p. 37f). Specifikationen vil tillige indbefatte nogle mere eksplicitte krav til spillets stabilitet såvel som nogle få tekniske begrænsninger (Bell, 2005, p. 42ff). Kravene er vurderet

### Funktionelle krav

* Brugerne af spillet skal kunne spillet et spil efter at have læst brugervejledningen.
* Brugergrænsefladen skal, i forlængelse af det forgående krav, være så intuitiv som mulig, sådan så normalbrugere vil være i stand til at kunne starte og afvikle et nyt spil.
* Brugergrænsefladen vise spillets status, herunder hvis tur det er, point, træk foretaget og det endelige resultat.
* Spillet skal være i stand til at håndhæve de gældende regler for Othello.
* Spillet skal kunne afvikles på to forskellige computere så to forskellige normalbrugere kan spille imod hinanden over et netværk.
* Spillerne skal, hvis forbindelsen imellem dem er etableret, kunne kommunikere med hinanden før, under og efter et spil er startet.
* Spillet skal være i stand til at kunne imødekomme og oplyse brugeren om adfærd der afviger fra det som er spillets intention.

### Tekniske og stabilitets krav

* Hvis spillet er startet, i henhold til det anviste i brugervejledningen, skal spillet være i stand til at kører stabilt. Det indebærer at man som spiller ikke skal opleve kritiske fejl der kan påvirke såvel spillets som hardwareenhedens overordnede stabilitet (Bell, 2005, p. 36f).
* Spillet skal programmers i Java og dermed, så vidt muligt, overholde de principper som stilles til objektorienterede programmeringssprog om *indkapsling* (eng. *encapsulation*), *nedarving* (eng. *inheritance*) og *polymorfi* (eng. *polymorphism*) (Bell, 2005, p.139ff & Internetkilde 3).

# Projektforløb

Dette kapitel har til formål at gøre rede for og skitsere projektforløbet samt de dertilhørende metodiske overvejelser og værktøjer der er blevet benyttet under dette forløb. Indledningsvist vil de anvendte teorier for designprocessen blive præsenteret, hvorefter en beskrivelse af projektforløbet vil efterfølge. Herunder vil projektforløbets designproces blive analyseret ud fra de netop præsenterede teorier. De dertilhørende konkrete metodiske værktøjer vil blive præsenteret i forlængelse heraf.

## Den objektorienterede designproces

Den objektorienterede designproces indebærer at man som udvikler i mindre grad tilstræber en skarp adskillelse af design og implementeringsforløbet - et princip som man traditionelt har anset som værende vigtig indenfor de mere traditionelle udviklingsmodeller (Bell, 2005, p.37f & p.146). Dette paradigmeskift er sket i erkendelsen af at udfordringerne samt aktiviteterne ved anvendelsen af objektorienteret programmering typisk vil være de samme under design- og i implementeringsforløbet:

*“…activities* [such] *as recognizing the class to reflect some newly recognized abstraction are just as likely to take place during the implementation of a prototype as during the design.”* (Bell, 2005, p.146)

Derfor kan objektorienteret design, i forhold til mere traditionelle udviklingsmodeller, i højere grad karakteriseres som værende præget af mere udforskende tendenser i design- og implementeringsfasen hvorfor længere sigtet planlægning typisk vil spille en mindre rolle.

### Iterativ udvikling

Iterativ udvikling er som regel en vital del af objektorienteret design, idet man anerkender at forståelsen for en given problemstilling typisk vil være meget dårlig i starten af et udviklingsforløb hvorfor en længere sigtet planlægning ikke altid vil være brugbar. Der er for eksempel igen garanti for at man i starten af et udviklingsforløb vil finde alle klasser, funktionaliteter eller det rette design for et givent program. Derfor anser man en gentagene og løbende revidering af design- og implementeringsproblematikker som essentiel og sommetider nødvendig i et givent udviklingsforløb. Et konkret værktøj til understøttelsen af denne metodik kan udmønte sig i udviklingen af tidlige prototyper for derigennem at skabe et holistisk billed af design- og implementeringsproblematikker. Igennem denne gentagene revidering af såvel implementeringen og designet opnås derfor en mere fleksibel udviklingsproces hvori det bliver lettere at inkorporere og tilpasse sig eventuelle ændringer. (Bell, 2005, p.146ff)

### Throwaway-prototyping

Idéen med *Throwaway-prototyping* er at implementere eller tage udgangspunkt i en prototype af det ønskede program så tidligt som muligt i en given iterativ udviklingsproces. Fordelen med implementeringen eller inddragelsen af en prototype er at man som udvikler vil have større mulighed for at identificere og italesætte eventuelle problematikker som man vælger at fokuserer på ved implementeringen af denne. For eksempel kan sådan et fokus være på de funktionelle kravspecifikationer eller måske mere designmæssige problemstillinger – problemstillinger som man ellers kunne have overset. Når man har vurderet at kravspecifikationerne, funktionelle eller ikke-funktionelle, er tilstrækkelige vil prototypen typisk blive kasseret hvorefter udviklingsforløbet den endelige version af et givent program vil starte (Bell, 2005, p.304ff).

## Projektforløbets faser

Projektforløbet for udviklingen af *Othello-Net* kan overordnet set opdeles i tre faser; en indledende, mellemliggende og afsluttende fase. Årsagen til netop denne inddeling skyldes at der i disse faser blev anvendt eller udarbejdet prototyper.

### Indledende fase – Udarbejdelse af en kravspecifikation

Udformningen af spillet tog udgangspunkt i en allerede eksisterende version af Othello, *NetOthello* (Bilag B: *NetOthello*)*,* hvorfor denne indledningsvist fungerede som en prototype. Målet med denne prototype var at udarbejde en *use-case* for derigennem at identificere problemstillinger og udforme en kravspecifikation (Bilag E: *Use-case*). Kravspecifikationen forholdte sig fortrinsvist til de funktionelle samt tekniske problemstillinger ved designet samt implementeringen af et sådan projekt. Mere specifikt blev der overvejet hvilke klasser der kunne genbruges fra såvel *NetOthello* samt fra API’et generelt.

### Mellemliggende fase – Revidering af kravspecifikation

I denne fase blev der, på baggrund af *NetOthello* og et Chat-program (Bilag D: *Chat Server/Client* & Internetkilde 12), foretaget overvejelser om designet og implementeringen af netværksstrukturen samt hvilke funktionaliteter dette projektforløbs endelige prototype skulle indeholde.

I den mellemliggende fase blev der implementeret en fungerende chat-server og klient på baggrund af det tidligere refererede program. Denne blev inddraget for at klarlægge hvilke tekniske krav der skulle til for at implementere en sådan funktion. På baggrund af disse erfaringer blev der udarbejdet en fungerende prototype af *Othello-Net* (Bilag C: *Othello-Net*). Denne version kunne initieres men var dog ikke i stand til at indbefatte de funktionelle krav, herunder den føromtalte chat-funktionalitet. Derudover indbefattede denne prototype mere grundlæggende fejl i reglerne af spillet og i spillets overordnede program.

### Afsluttende fase – Implementering af Othello-Net

I den afsluttende fase af dette design- og implementeringsforløb af *Othello-Net* blev der på baggrund af *UML*-diagrammer og generelle erfaringer fra den anden prototype af *Othello-Net* (Bilag C: *Othello-Net*), designet og implementeret en ny version hvis overordnede designstruktur havde taget udgangspunkt i et designmønster. Anvendelsen af designmønstret muliggjorde den nuværende version af *Othello-Net* (Bilag A: *Othello-Net*) og dermed en opfyldelse af de opdaterede kravspecifikationer fra den mellemliggende fase.

Afsluttende i denne fase blev der tillige udført afprøvninger af programmets funktionelle krav samt dens stabilitet.

## Delkonklusion

Der kan i forlængelse af de netop fremlagte faser konkluderes at *Othello-Net* er forsøgt udviklet på baggrund af objektorienterede designprincipper. Dette er sket efter analyse og udarbejdelse af tre prototyper.

# Teori

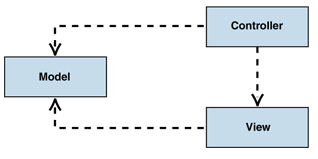
I dette kapitel vil de overordnede teoretiske overvejelser der har været i forbindelse med designprocessen blive taget op. Konkret omhandler disse overvejelser det anvendte designmønster og netværksstrukturen. Efter hver præsentation af teorierne vil der følge en analyse af hvordan de er blevet implementeret i *Othello-Net*.

## Designmønstre

I Java og generelt når det gælder objektorienterede programmeringssprog opererer man ofte med såkaldte designmønstre. Designmønstre kan kort beskrives som værende løsninger på hyppigt opstående designproblemer. Ved anvendelsen af et givent designmønster enten i starten af eller under en udviklingsproces vil man være i stand til helt at undgå eller i hvert fald nedsætte risikoen for problemstillinger som normalt ikke anses som værende attråværdige inden for objektorienteret programmering (Bell, 2005, p. 151 & Internetkilde 3).

### Model-View-Controller

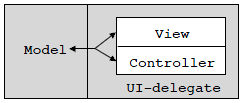
Model-View-Controller strukturen er et designmønster som anerkender at de fleste, om end ikke mange systemer, overordnet set består af tre centrale dele: en simulation, en input-del og en output-del. Simulationen bliver som oftest kategoriseret som *modellen*, input-delen som c*ontrolleren* og output-delen som *viewet* (Internetkilde 7).



*Figur 6.1: Illustration af MVC-mønstret (Internetkilde 7).*

Det overordnede formål ved denne strukturering af klasserne imellem er at gøre modellen (eng. *model*) uafhængig af kontrolløren (eng. *controlleren*) og brugergrænsefladen (eng. *view*). Med indførslen af dette design bliver det muligt at lave ændringer i brugerænsefladen eller udskifte den helt uden at tage stilling til modellen, hvilket er essentielt idet brugergrænsefladen som oftest udskiftes eller ændres langt hyppigere end modellen. Princippet om lav afhængighed (eng. *loose coupling*) imellem klasser er et designprincip som man tillige anser som vigtigt indenfor objektorienteret programmering generelt (Internetkilde 3 & 6).

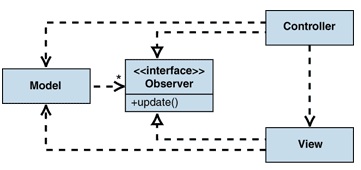
Som illustreret i figur 6.1 fordrer dette designmønster også lav afhængighed af brugergrænsefladen og kontrolløren, men det er dog ikke hovedformålet med dette mønster og heller ikke ønsket i alle afarter af programmer med *MVC*-strukturen. Årsagen til dette, det vil sige ønsket om *øget* afhængighed og integration imellem kontrolløren og brugergrænsefladen, kan være forskellige men kan grunde i ønsket om mindre kompleksitet eller at den modellerede entitet eller koncept typisk vil give bedre mening (eng. *cohesion*) uden denne opdeling af brugergrænsefladen og kontrolløren. Konkrete eksempler på dette tilfælde kan ses i Java-klassen *Swing*, hvori en sådan opdeling ikke findes og i programmer hvor man opererer med såkaldte tykke-klienter vil denne struktur også være mindre fordelagtig. Denne variation af *MVC*-mønstret, hvor brugergrænsefladen (eng. *view*) og kontrolløren (eng. *controlleren*) er integreret med hinanden, kaldes også *model-delegate-*mønstret (Internetkilde 4)*.*



*Figur 6.2: Illustration af Model-delegate mønstret* (Internetkilde 4)

### En MVC-variation: Observer-mønsteret

*Observer*-mønstret er en specialisering af *MVC*-mønstret. Dette mønster vil man med fordel kunne anvende i programmer som har, eller hvor man ønsker, en *aktiv*-model. En aktiv model bliver, i modsætning til den passive variation, ikke kun ændret af kontrolløren men også af andre kilder for eksempel en klient (Internetkilde 7). Introduktionen af denne øgede afhængighed vil dog være i modstrid med det netop beskrevne princip i *MVC*-mønstret. Derfor introducerer man typisk en mekanisme som informerer andre objekter om ændringer i modellen *uden* at øge afhængigheden klasserne imellem. Konkret gøres dette ved at definere en eller flere *Observere* og *Observables*. *Observere* vil typisk være de dele af programmet som har brug for at vide modellens status og *Observables* er modellen selv. Som *Observer* vil man være i stand til at tilmelde (eng. *subscribe*) sig modellen og derigennem få information om eventuelle ændringer i modellens tilstand. Dette tillader at modellen, den *observante*, ikke behøver at vide noget om de dele af programmet som er afhængig af den, hvorfor man undgår introduktion af den øgede afhængighed. *Observer*-modellen er en afart af *publish/subsribe*-mønstret og er en del af Javas API.

**

*Figur 6.3: Illustration af Observer/Observable-mønstret (Internetkilde 7).*

### MVC i Othello-Net

Under udviklingen af dette projektforløbs endelig prototype af *Othello-Net* er det overordnede design løbende blevet overvejet, da der på baggrund af de tidligere prototyper et blevet erfaret nødvendigheden af dette, jf. kapitlet *Projektforløbet*.

På baggrund af denne erkendelse blev der, under udviklingen af den sidste prototype af *Othello-Net* i dette projektforløb, fundet frem til hvad der kunne defineres som værende input (eng. *view*), output (eng. *controller*) og modellen (eng. *model*). Under den efterfølgende implementering og kategorisering af klasserne, blev problematikkerne ved *ikke* at integrere kontrolløren og brugergrænsefladen, som *MVC*-mønstret i nogle tilfælde anbefaler, blottelagt. Adskillelsen af brugergrænsefladen og kontrolløren besværliggjorde nemlig implementeringen af en brugergrænseflade som defineret i kravspecifikationen. Dette grunder i at brugergrænsefladen blev implementeret igennem brugen af klassen *Swing* som bygger på *MVC*-variationen, *model-delegate*, der fordrer en integration af kontrolløren og brugergrænsefladen, hvorfor det blev nødvendigt at integrerer de nævnte dele til trods for øget afhængighed (Internetkilde 4). Til at afhjælpe denne øgede afhængighed imellem brugergrænsefladen og kontrolløren (eng. controlleren) blev det derfor nødvendigt at gøre brug af *Observer*-klassen, der netop kunne afhjælpe problemet idet den gør brug af *publish/subscribe*-mønstret.

Designet af *Othello-Net* er derfor overordnet set baseret på princippet om lav afhængighed mellem de forskellige klasser, hvilket er forsøgt implementeret igennem en løbende tilpasning af *MVC*-mønstret.

## Server-Klient netværksstrukturer

Den mest anvendte netværksstruktur er den såkaldte *Server-klient* model. Server-klient modellen består typisk af en server som tilbyder nogle tjenester. Klienter har så mulighed for at tilkoble sig denne server og gøre brug af de tjenester som bliver udbudt af serveren. Denne model kan illustreres ved brugen af internetsurfing. Her agerer internetbrowseren som en klient og de enkelte hjemmesider kan betegnes som serverer eller repræsentationer af disse. Man forbinder sig så til en given server ved at angive en internetadresse i søgefeltet.

### Tynde og tykke klienter/servere

Servere og klienter kan overordnet set designes som værende enten *tykke* eller *tynde*. En tynd server eller klient implicerer at computeren eller softwaren som bliver anvendt i disse roller typisk vil være meget afhængig dens modpart i udførslen af dens, traditionelt set, grundlæggende opgaver. I tilfælde hvor den respektive enhed er tyk vil afhængigheden af hinanden falde. Denne opdeling fordrer visse ulemper såvel som fordele hvorfor valget mellem de to modeller vil være afhængig af konteksten hvori serveren og klienten benyttes.

For eksempel, det at have en tyk server som skal vide alt om givent spil gør at kravene til de tynde klienter vil være mindre i henhold til ydelse da al behandlingen af spillet sker hos serveren. Dette vil også øge kravene til serveren idet den skal kunne behandle de enkelte klienters forespørgsler i højere grad end hvad ellers ville være tilfældet. Denne øgede integration vil desuden gøre klienterne mere afhængige af serveren hvorfor klienten i tilfælde af eventuelle nedbrud af serveren ikke vil være i stand til at kører.

Begge strukturer har derfor deres styrker og svagheder som man må vurderer i forhold til konteksten. Traditionelt set kan der dog identificeres visse tendenser, idet programmel såsom spil eller lignende grafisk krævende programmer er blevet designet med tykke klienter og tynde servere. Årsagen til dette designvalg skal ses i lyset af at grafisk krævende programmer kræver relativt store mængder båndbredde for at fungere optimalt, hvilket traditionelt set har været mindre tilgængeligt. Tillige vil et spil, hvis bestanddel af programmel lå på en server, ikke være i stand til at fungere uafhængigt af en server, hvorfor det ville indskrænke funktionaliteten af spillet betydeligt idet muligheden for at for eksempel at spille offline ikke ville være til stede.

### RMI og Sockets i Java

Hvis man vil implementere et programmel med en server-klient struktur i Java, kan det gøres ved hjælp af enten Sockets eller RMI(Remote Method Invocation).

Sockets er objekter som gør brug af TCP/IP protokollen. Igennem denne protokol vil kommunikationen mellem henholdsvis et serverSocket og Socket,typisk implementeret i klienten, finde sted. Serverens Socket vil så sættes til at lytte på en given port og vente på indkommende klienter. Når en klient så er modtaget vil det være muligt for klienten og serveren at kommunikere.

Kommunikationen imellem en server og klient vil med anvendelse af RMI foregå anderledes. Mere specifikt kommunikerer man ved hjælp af fjerne (eng. *remote*) metodekald. Fjerne betyder i denne sammenhæng at den ene klient kalder metoder fra programmel, typisk beliggende på en server eller klient, igennem et netværk. Klienterne interagerer derfor med hinanden ligesom klasser interagerer med hinanden i et lokalt programmel generelt.

Der er fordele og ulemper ved anvendelsen af begge strukturer. RMI kan karakterisereres som værende en abstraktion af Sockets, idet RMI ligeledes benytter sig af Socket-strukturen men denne er dog indkapslet så man vil i mindre grad beskæftige sig med de grundlæggende netværks-mekanismer som man gør ved brugen af Sockets. Grundet denne abstraktion og indkapsling er man bundet til at sende objekter over et netværk som alt afhængigt af konteksten kan have sine fordele og ulemper. I Sockets er det tillige muligt at sende objekter, men man er ikke bundet til denne omstændighed da man kan vælge at benytte sig af primitive datatyper såsom integers. Dette giver en hvis fleksibilitet, men kræver også et større udviklingsarbejde.

En ulempe ved Sockets fremfor brugen af RMI, er at man relativt hurtigt støder ind i begrænsninger ved brugen af Sockets alene, specielt i henhold til designet af serveren. Servere med kun en tråd eller slet ingen er som oftest enklere men også mere ineffektive idet de er nødsaget til at acceptere og behandle de indkommende klienter i én enkelt tråd/klasse. Det at disse to processer befinder sig i samme tråd/klasse, medfører at serveren bruger mindre tid i dens accept-løkke hvorfor risikoen for at afvise eventuelle indkommende klinter bliver væsentlig større. Derfor vil man typisk behandle de accepterede klienters forespørgsler til serveren/andre klienter i en ny tråd eller klasse (eng. *client handler*) og dermed gøre brug af flere tråde, heraf navnet m*ultithreading*. Brugen af flere tråde vil dog typisk øge kompleksiteten og dermed chancen for komplikationer som for eksempel *deadlocks* og *race-conditions.*

Sockets og tråde kan derfor med fordel benyttes i situationer hvor brugeren ønsker en højere grad for tilpasning og i nogle tilfælde en enklere implementering alt afhængigt af kravene til ens server. Tillige vil den manglende abstraktion gøre at Sockets i udgangspunkt yder bedre, men man udnytter ikke muligheden for genbrug som man ellers anser for et vigtigt designprincip inden for objektorienteret udvikling (Internetkilde 3).

## Netværksstrukturen i Othello-Net

*Othello-Net* er blevet bygget op omkring princippet om tykke-klienter og tynde serverer. Dette indebærer, som nævnt, at de fleste af *Othello-Net*s grundlæggende funktioner bliver udført i klientdelen af spillet. Dette er gjort på baggrund af blandt andet erfaringerne med den første prototype, *NetOthello*, som ligeledes har implementeret Othello som en tyk klient (Bilag B: *NetOthello*). I forlængelse af de specificerede krav til *Othello-Net*, har muligheden for at udvikle en applikation frem for en applet tillige været en vigtig faktor idet tykke klienter i højere grad vil være uafhængig af en server.

### Sockets

Netværksstrukturen i *Othello-Net* er blevet designet og implementeret igennem brugen af Sockets. Grunden til at valget netop er faldet på Sockets frem for RMI skyldes hovedsageligt at den allerede eksisterende prototype, *NetOthello*, netop også tog udgangspunkt i denne metode. Kravet om at klienterne skulle være i stand til at chatte med hinanden, altså kommunikere uafhængigt af hinanden, var desuden ikke med i overvejelserne på dette tidspunkt, hvorfor RMI ikke blev anset som værende nødvendigt, jf. kapitel *Projektforløb*.

Efterhånden som kravspecifikationen udviklede sig blev brugen af flere tråde (eng. m*ultithreading*) dog en nødvendighed. Valget af Sockets har derfor givet mulighed for en hvis fleksibilitet i udviklingen af *Othello-Net* men har grundet indførslen af flere tråde ikke gjort det enklere end RMI at designe og implementerer.

## Delkonklusion

Opsummerende kan der ud fra det ovenstående kapitel konkluderes at *Othello-Net* i udviklingen af det overordnede design har benyttet sig af designmønstret *MVC* og netværksstrukturen benytter sig af Sockets i målet om at opfylde kravspecifikationerne. Begge designbeslutninger har desuden i høj grad været præget af de erkendelser og erfaringer der blev gjort i design- og implementeringsfasen af de tidligere prototyper.

# Programdokumentation

Formålet med dette kapitel er at gøre rede for programmets virkemåde. Inden denne beskrivelse vil kapitlet først redegøre for hvordan *Othello-Net* har anvendt *NetOthello* i implementeringsfasen. Efter denne udredning vil programmellets klassestruktur og interaktionen imellem disse blive beskrevet. På baggrund af dette vil en uddybende redegørelse af de dele af klasserne som er relevante for forståelsen af programmellets virkemåde følge. I forlængelse heraf vil der, i tråd med principperne for objektorienteret programmering, blive foretaget en vurdering af de enkelte klassers styrker såvel som svagheder.

## Othello-Nets anvendelse af NetOthello

*Othello-Net* har i dette projektforløb gjort brug af *NetOthello* som prototype. Mere præcist blev *NetOthello*, hovedsageligt i starten af dette projekt, anvendttil at identificere de funktionelle krav som specificeret i kapitlet herom. I et programmeringsøjemed er der udover søge-algoritmerne og de dertilhørende metoder ikke blevet genbrugt andre dele af *Othello-Net*, helt eller delvist*.* Dette skal ses i lyset af de opstillede kravspecifikationer. Derudover anvender NetOthello desuden en ældre version af Java hvis lighed efterhånden er begrænset hvorfor en yderligere inspiration/genanvendelse af *NetOthello* ikke var relevant. Konkret ses det i for eksempel brugergrænsefladen hvor *NetOthello* benytter sig af Java-klassen AWT fremfor den nyere Java-klasse Swing der anvendes i *Othello-Net* (Bilag A: *Othello-Net &* Bilag B: *NetOthello*). For yderligere uddybning af genanvendelsen af *NetOthello* jf. underafsnit om BoardModel-klassen.

## Klasserne i Othello-Net

*Othello-Net* er, som beskrevet i det forudgående kapitel, designet og implementeret på baggrund af principperne om *MVC*-mønstret. Overordnet set kan *Othello-Net*s klasser derfor kategoriseres som værende en del af enten modellen, viewet eller controlleren.

*UML*-diagrammet (figur 7.1), på næste side, illustrerer en skitsering af programmets overordnede design og struktur, for en mere detaljeret version jf. Bilag F. I dette diagram kan det anvises at der er blevet forsøgt anvendt princippet om *MVC*. Her ses det at *Othello-Net* i alt består af 7 klasser. Disse klasser er kategoriseret som følgende:

### View/Controller (Observer)

* *GUI.java*
* *BoardView.java*

### Controller/Klient

* *OthelloClient.java*

### Model (Observable)

* *BoardModel.java*

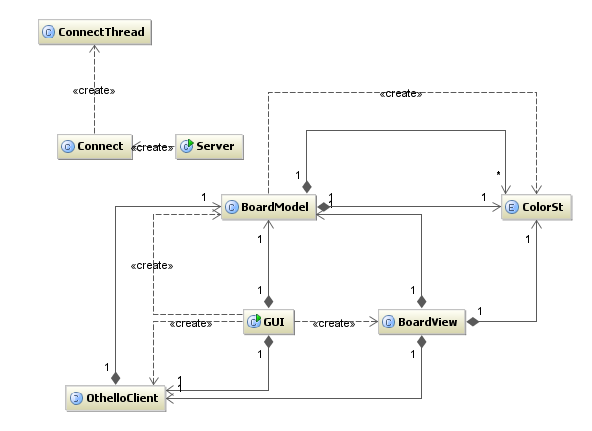
### Serveren

* *”Server.java”*
* *”Connect.java”*
* *”ConnectThread.java”*

*Othello-Net* indeholder udover de 7 klasser også en enum-klasse kaldet *”ColorSt.java”*.

## Klassernes interaktion

Ud fra det nedenstående *UML*-diagram kan det anvises at klasserne interagerer på baggrund af de roller som *MVC*-mønstret anbefaler.



*Figur 7.1: UML-diagram over Othello-Net*

Dette indebærer nærmere at klasserne, GUI, BoardView og OthelloClient benytter model-klassen, BoardModel ved enten at refererer til den eller ved at kalde dens getter- og setter-metoder.

BoardModel er ikke afhængig af nogle af de andre klasser. Dens interaktion med de andre klasser foregår som nævnt igennem getter- og setter-metoder. Igennem *Observer*-klassen tillader den tillige at andre klasser kan tilmelde (eng. *subscribe*) sig til denne, for derigennem at få information om eventuelle ændringer i dens datastrukturer. I dette tilfælde bliver det benyttet af BoardView-klassen som igennem en metode får besked om eventuelle ændringer i BoardModels datastrukturer.

Serverdelen er ikke afhængig af nogle af de andre klasser i *MVC*-mønstret, idet dens interaktion alene foregår ved brugen af Sockets, nærmere bestemt igennem klassen OthelloClient. Serverdelen består af i alt tre klasser, Server, Connect og ConnectThread, hvis afhængighedsforhold er defineret igennem brugen af referencer.

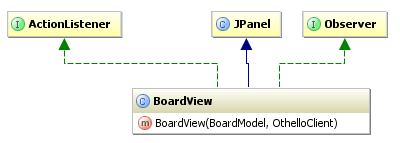
Enum-klassen, ColorSt, bliver refereret af BoardModel og BoardView.

## View og Controller-klasserne

De klasser som står for at vise *Othello-Net*s brugergrænseflade samt håndtere dens input er: GUI-, BoardView- og OthelloClient-klassen. Disse klasser vil i de følgende underafsnit blive beskrevet med henblik på at give et overblik over deres virkemåde, hvorfor ikke al kode vil indgå. Hvis der ønskes et overblik over hele *Othello-Net*s kode, henvises der til Bilag A.

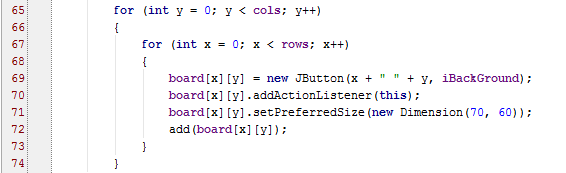
### BoardView-klassen

BoardView-klassen har som hovedformål at repræsentere selve spillepladen, hvorfor den henholdsvis har nedarvet og implementeret JPanel-klassen, en actionListener samt en *Observer*. BoardView-konstruktøren tager, i tråd med det anvendte designmønster, desuden programmellets model og ene controller som parametre, altså BoardModel og OthelloClient, se figur nedenunder.



*Figur 7.2: Diagram af BoardView-klassens nedarvning og implementering*

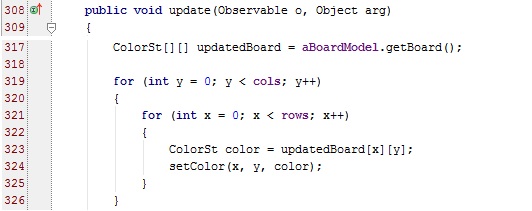
Idet klassens hovedformål er at repræsentere en spilleplade, bliver spillepladen, som er implementeret igennem brugen af JButton-klassen, initieret i konstruktøren. Dette bliver gjort ved hjælp af en for-løkke som placerer de enkelte JButtons på spillepladen i otte rækker og kolonner. I denne løkke bliver de tillige tildelt et koordinat, et ikon og klassens actionListener. Koordinatet bruges til at identificere hver enkel knap på spillepladen når den skal sendes til modstanderen og modellen. Til at starte med initieres ikonerne som selve spillepladen (se figur 7.3), men de repræsentere ligeledes de enkelte brikker ved at skifte ikonet ud når der bliver trykket på knapperne.



*Figur 7.3: Knapperne bliver initieret.*

actionListener()-metoden lytter på eventuelle events som sker i klassens brugergrænseflade om end det er i tekst-feltet eller på selve spillebrættet. Hvis et sådan event opstår tjekkes der først om klienten har forbindelse til en modspiller ved at anvende OthelloClients metode, isConnected(). I tilfælde af at der er forbindelse tillades det at sende enten en besked eller et træk til modstanderens klient.

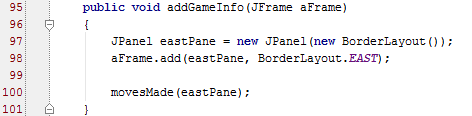
I tillæg til dette har BoardView-klassen, grundet dens rolle som *view*, implementeret *Observer*-klassen, hvilket gør den i stand til at *Observer*e eventuelle ændringer i den observante klasse, BoardModel. Beskederne om ændringer i modellen får den igennem update()-metoden som kræves implementeret når *Observer*-klassen anvendes i en given klasse. I dette tilfælde bliver update()-metoden benyttet til at holde den grafiske brugergrænsefladet opdateret med spillets gang og status som løbende bliver ændret under spillet, se figur nedenunder.



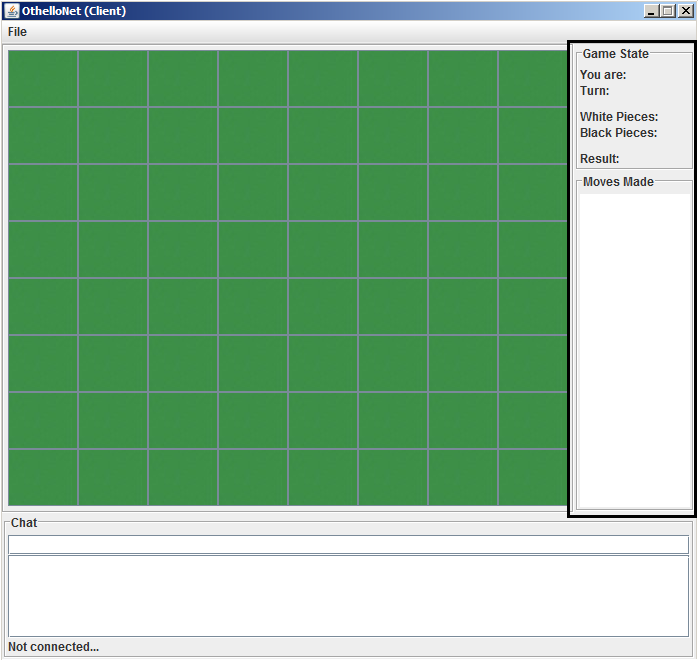
*Figur 7.4: Update-metoden som er tilmeldt Observable klassen BoardModel.*

Ændringerne i modellen, BoardModel, bliver løbende foretaget af BoardViews actionListener()-metode og af de kommandoer som bliver modtaget af OthelloClient-klassen, jf. afsnit om OthelloClient-klassen for yderligere uddybning heraf.

BoardView-klassen viser som nævnt også spillets status som den får fra modellen blandt andet. Disse informationer bliver implementeret igennem metoderne addGameInfo() og addChatAndLabels(), som deler spilinformationerne op alt afhængig af hvor de er placeret i JFramen. For eksempel placeres movesMade()-metoden, i henhold til layoutklassen BorderLayout, i den østlige del af JFramet (se figur 7.5 og 7.6).

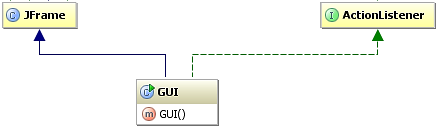


*Figur 7.5: movesMade-metode placeres i den valgte JFrame.*



*Figur 7.6: MovesMade()-metoden (markeret) implementeret i den østlige del af JFramet.*

### GUI-klassen



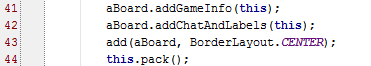
*Figur 7.7: UML-diagram der viser nedarving og implementering i GUI-klassen.*

GUI-klassen nedarver (eng. extends), som anvist i figuren ovenfor, JFrame-klassen. Dette er gjort med den hensigt at vise og samle BoardModel, OthelloClient og BoardView i én JFrame og klasse.



*Figur 7.8: Objekter af de nævnte klasser.*

Objekterne aModel og aClient bliver efter initieringen tilføjet til BoardView-objektet, aBoard. Dette objekt bliver efterfølgende tilføjet til GUI-klassens JFrame (jf. underafsnit om BoardView-klassen for uddybning af linje 41 og 42).

**

*Figur 7.9: Objekterne bliver sat ind i GUIs JFrame.*

GUI-klassen indbefatter, udover de tre klasser, funktioner som ikke direkte kan moduleres fra virkeligheden, men stadig er nødvendige i forbindelse med afviklingen af et spil, herunder muligheden for at starte, genstarte og afslutte et spil. Disse er implementeret igennem brugen af Swing-klassen, JMenu.

### Styrker og svagheder

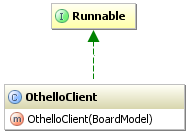
BoardView-klassens design holder sig så vidt muligt inden for rammerne af *MVC*-princippet, hvilket giver fordele som man anser som attråværdige i objektorienteret programmering, heriblandt muligheden for genbrug og let udskiftning af brugergrænsefladen. Dette forhindres dog til en vis grad idet BoardView-klassen besidder en hvis afhængighed da der er erklæret en del variabler som er statiske i netop denne klasse. Derudover har den også to variabler, movesMade og textArea der bliver brugt af andre klasser hvorfor afhængigheden øges. I henhold til samhørigheden (eng. *cohesion*) af BoardView-klassen, kunne det have været mere hensigtsmæssigt at have adskilt koncepter som brikker og plader, hvilket til eksempel kunne have gjort det muligt at gøre BoardView lettere at generalisere igennem brug af interfaces.

## Controller og Klient-klassen

I det følgende afsnit vil virkemåden af klassen OthelloClient blive kortlagt.

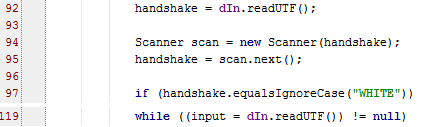
### OthelloClient-klassen

OthelloClient-klassen kan overordnet set defineres som havende to roller nemlig klient- og kontrollørrollen (eng. *controller*) da den både modtager og sender informationer igennem brugen af en server og samtidig interagerer med modellen, BoardModel. OthelloClient implementere, som anvist i figuren nedenfor, Runnable-klassen og tager BoardModel som parameter i dens konstruktør.



*Figur 7.10: OthelloClient-klassen.*

OthelloClients konstruktør har til opgave at etablere en forbindelse med en server. Dette gøres ved at lave et objekt af klassen Socket som søger efter en server på adressen ”localHost” og port 4444. Hvis dette i så fald lykkes initieres et input- og outputstream af typen DataStream samt en tråd. Klassen implementerer som nævnt Runnable- klassen der indeholder en read-løkke og en dertilhørende protokol i dens run-metode. Read-løkken nås først når serveren har sendt et håndtryk (eng. handshake) af typen String, indeholdene teksten ”WHITE” eller ”BLACK”.



*Figur 7.11: Håndtryk sendt fra serveren indeholdende teksten WHITE eller BLACK og den efterfølgende read-løkke.*

Hvis håndtrykket bliver accepteret nås læse-løkken hvorefter den starter med at læse eventuelle chat-beskeder eller kommandoer fra den anden klient igennem brugen af en protokol. Input med præfikset ”msg” bliver tolket som chat-beskeder. Derudover tillader protokollen brugen af kommandoerne: ”move”, ”init”, ”resign” og ”exit”.



*Figur 7.12: OthelloClients metoder.*

### Styrker og svagheder

Grundet dens rolle som både kontrollør og klient har den en øget afhængighed til klasserne GUI, BoardModel og BoardView. Dette kommer konkret til udtryk igennem brugen af blandt andet de respektive klassers statiske-variabler. En bedre løsning kunne have været at indkapsle adgangen til variablerne ved brug af setter og getter-metoder.

Brugen af Strings og de dertilhørende streams som kommunikationsmedie imellem klienterne giver den fordel at det umiddelbart bliver lettere at implementere en chat-funktion i henhold til kravet i kravspecifikationen. Til trods for dette har der tillige været problemer med anvendelsen af Strings for eksempel er det ikke muligt at udføre de samme operationer på dem i for eksempel if-betingelser.

## Model

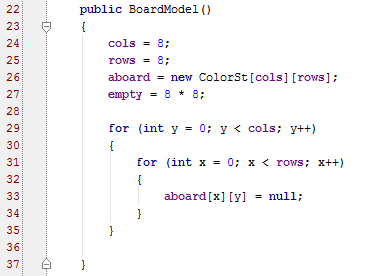
*Othello-Net* indeholder BoardModel-klassen som er en model eller simulation af spillet Othello. Ikke alle dele af denne klasse vil blive beskrevet men blot de dele som har til hensigt at give et overblik og klassens virkemåde og intentioner generelt. Dele af denne klasses metoder bygger på en allerede eksisterende version af Othello, nemlig *NetOthello* som blev brugt som prototype indledningsvist i dette projektforløb jf. kapitel om problemfelt og projektforløbet for yderligere uddybning.

De dele af programmet som bygger på *NetOthello* er metoderne: isLegalMove()(hedder checkMove() i *NetOthello*), doMove(), validMoves()og isEndOfGame(). Grundet nogle grundlæggende forskelle på *Othello-Net* og *NetOthello* blev de føromtalte dog tilpasset før de kunne benyttes i denne klasse. Se Bilag B for originalkoden.

*Figur 7.13: Metoder i BoardModel-klassen*

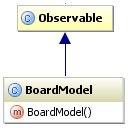
### BoardModel-klassen

Boardmodel-klassen har som nævnt til formål modellerer selve spillet Othello, hvilket konstruktøren giver udtryk for idet den initierer en simulation af et spil igennem brugen af et 2-dimensionelt array af enum-typen ColorSt.



*Figur 7.14: Konstruktøren i BoardModel.*

Denne rolle fordrer desuden, efter princippet i *MVC*-mønstret, en uafhængighed af såvel brugergrænsefladen (eng. *view*) og dens kontrollør(e) (eng. *controller*). Det er opnået igennem nedarving af *Observable*-klassen, hvilket gør det muligt for relevante klasser at blive opdateret om eventuelle ændringer i denne model.



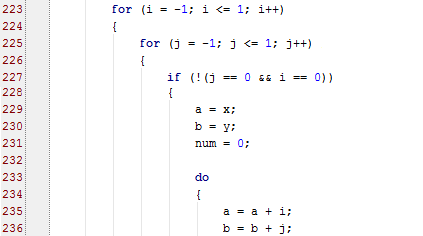
*Figur 7.15: Nedarving af klassen Observable*

De dele eller metoder af BoardModel-klassen som observeres er: addPiece()-, doMove()-, setWinner()-, initGame()- og clearBoard()-metoden. Mere specifikt er dette gjort ved hjælp af *Observer*-klassens metoder, setChanged() og notify*Observers*(), som er placeret inden i de førnævnte metoders kroppe. Årsagen til at det netop er disse metoder er, at de er nødvendige for at holde brugergrænsefladen. Metoderne isLegalMove()og doMove(), som er baseret på tilsvarende metoder i *NetOthello*, er centrale i opfyldelsen af denne klasses rolle da de står for hovedparten af den udførte kontrol af et givent spils regler. De to metoder er overordnet set meget ens, jf. figur nedenfor.



*Figur 7.16: Metoderne isLegalMove og doMove.*

isLegalMove()og doMove()har begge til formål at tjekke om hvorvidt et træk er lovligt. Dette gøres igennem brugen af en algoritme som ud fra en given position, angivet som parametrene x og y jf. figur ovenfor, søger såvel vandret lodret og diagonalt (se figur 7.17).



*Figur 7.17: Søge algorimeten i doMove()-metoden*

doMove()foretager derudover også selve trækket og ændre dermed i selve modellen eller simulationen af spillets data. I tillæg til disse funktioner har doMove()-metoden til opgave at kontrollere om farven, der ligeledes angives som parameter, har flere lovlige træk tilbage og i forlængelse heraf om spillet har en vinder eller det skal ende uafgjort. For at opfylde disse funktioner anvender den metoderne validMoves() og isEndOfGame().

### Styrker og svagheder

BoardModel-klassen har igennem anvendelsen af *Observer*-mønstret og dens getter og setter metode bevaret en afhængighed og dermed de fordele som følger heraf. Ulempen i dette konkrete tilfælde er at dens lytter, BoardView-klassen, ikke har en måde at tjekke om hvorvidt brugergrænsefladen i virkeligheden afspejler det som sker i BoardModel-klassen, i tilfælde af eventuelle fejl eller forsinkelser. Denne mulighed for fejl gør sig tillige gældende i forbindelse med OthelloClient-klassen, hvorfor man med fordel kunne have implementeret sikkerhedsforanstaltninger med dette problem for øje.

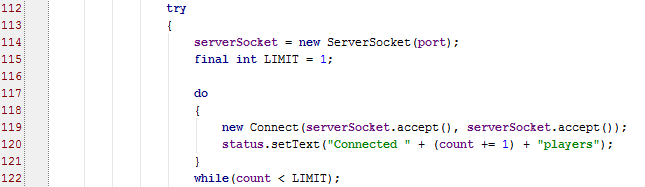
I henhold til modellens to metoder isLegalMove() og doMove()ville man med fordel have kunne sammenlagt dem idet de besidder nogle af de samme funktioner. Mere konkret gælder dette specielt søge-algoritmerne som er identiske.

## Serveren

Server-delen i *Othello-Net* består af i alt tre klasser: Server, Connect og ConnectThread. Deres roller er, sammen med klienten, at opfylde kravet om at kunne spille Othello over et netværk.

### Server-klassen

Server-klassen har som sagt til formål at agere som server for de indkommende klienter. Dette gør den ved at initiere et serverSocket som lytter på port 4444. De indkommende klienter bliver så løbende accepteret ved hjælp af server-klassens accept()-metode som returnerer to Sockets. Disse Sockets bliver efterfølgende anvendt som parametre i klassen Connect der håndtere og gruppere klienter to og to (eng. clienthandler). Serveren er af stabilitetsgrunde sat til kun at acceptere to par af klienter, jf. kodefragment nedenunder.



*Figur 7.18: Server-løkken accepterer og videresender klienternes Sockets til Connect-klassen.*

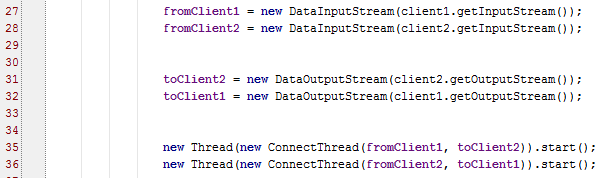
### Connect-klassen

Denne klasse håndterer de indkommende klienters Sockets som den får fra serverens accept()-metode, hvoraf den skal bruge to får at blive initieret.



*Figur 7.19: Connect-klassens parametre.*

Når den har modtaget klienternes Sockets laver den I/O-streams af typen DataStream (se figur 7.20). Hvis dette bliver gennemført uden nogle former for exceptions, anvendes de som parametre i ConnectThread-klassens konstruktør hvorefter de respektive tråde i ConnectThread-objekterne initieres, jf. næste afsnit for uddybning om ConnectThread for yderligere uddybning heraf.



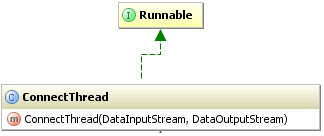
*Figur 7.20: Streams fra de to Sockets bliver lavet og sat ind i ConnecThread.*

Konstruktøren i Connect-klassen, afsluttes med at sende et håndtryk til de to klienter, af typen String indeholdene teksten, ”WHITE” eller ”BLACK”.

I tilfælde af at der skulle kastes en exception bliver den grebet af try/catch-blokken hvorefter metoden cleanUp()bliver kaldt for at lukke eventuelle åbnede Streams der ikke er lukkede.

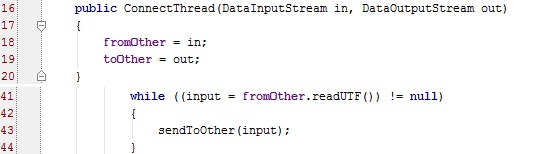
### ConnectThread-klassen

Sammen med Connect-klassen, har denne klasse til opgave at håndtere de indkommende klienter. Til forskel for den føromtalte klasse har implementeret en tråd, jf. figur nedenunder.



*Figur 7.21: Har implementeret en tråd igennem klassen Runnable.*

Det er gjort med den hensigt at kunne læse eventuelle beskeder eller kommandoer løbende fra den anden klients DataInputStream som den får fra Connect-klassen. Idet ConnectThread-klassens konstruktør også tager deres ’egne’ klienters Sockets DataOutputStream, er den tillige i stand til at sende beskeder til den anden klient ved brugen af sendToOther()-metoden, jf. figur nedenfor.



*Figur 7.22: Konstruktøren og læse-løkken i ConnectThread der gør brug af sendToOther()-metoden.*

### Styrker og svagheder

*Server*-klassen udfører funktionerne som specificeret i kravspecifikationen, nemlig at to spillere skal være i stand til at kunne spille mod hinanden over et netværk. Der forholder sig dog også visse problematikker i forbindelse med dens design. Mere konkret i henhold til accept-løkken som grundet client-handlerens krav om to klienter kan blokerer i længere tid ad gangen, hvilket kan begrænse anvendelsen af denne struktur i sammenhænge hvor høj effektivitet er en vigtig del af kravspecifikationen.



*Figur 7.23: Accept-løkken i Server-klassen.*

# Afprøvning af program

I dette kapitel vil jeg gøre rede for de interne afprøvninger der er blevet foretaget på den endelige prototype af *Othello-Net*. I henhold til projektets begrænsede tidsramme samt dets hovedfokus er der ikke foretaget dybdegående afprøvninger af programmellet. Til test af programmellets interne struktur er der blevet gjort brug af generel debugging, herunder brugen af testudskrifter (Bell, 2005, p. 253f). Der er tillige blevet udført tests på baggrund af principperne om *black-box* metoden (Bell, 2005, p. 269f), hvorfor denne del af afprøvningen ligeledes har haft til formål at teste om hvorvidt programmellet opfylder de opstillede funktionaliteter i kravspecifikationen.

## Test af programmellets struktur og funktionaliteter

Koden i den endelige prototype af *Othello-Net* er blevet debugget ved løbende at placere System.out.print()-metoder før såvel som efter udvalgte kodeblokke, hvorpå det forventede resultat såvel som det egentlige resultat blev sammenlignet efterfølgende. Denne fremgangsmåde er især blevet benyttet på klassen BoardModel, mere specifikt på dens søge-algoritmer såvel som på det samlede output der blev produceret af denne klasse. I tillæg til denne metode blev der desuden anvendt compilerens egne debugging-værktøjer. Senere i udviklingsforløbet af den endelige prototype blev disse test suppleret af *black-box* afprøvninger. Disse tests havde til formål at identificere såvel strukturelle fejl og funktionelle mangler, hvorfor de ligeledes fungerede som et supplement til de testudskrifter som specificeret i starten af dette afsnit.

## Resultatet af afprøvningerne

På baggrund af de ovenstående tests af den endelige prototype af *Othello-Net* blev der identificeret følgende:

* **Søge-algoritmen:** På baggrund af flere *black-box* tests blev der identificeret uregelmæssigheder i henhold til de gældende regler for Othello. Mere konkret opstod uregelmæssigheden i situationer hvor en af spillerne ikke havde mulighed for at udføre et træk men alligevel blev tildelt turen, hvorfor spilleren ikke havde mulighed for at placere sin brik. I sådanne situationer tilskriver reglerne af en given spiller kan få lov til at udføre træk indtil den anden spiller igen har mulighed for at udføre et lovligt træk. Løsningen på dette problem blev fundet igennem separate tests af BoardModel-klassen og brugen af System.out.print()-metoder.
* **Serverblokering:** Ved at udføre *black-box* tests af serveren fandt jeg frem til at i tilfælde af at den er startet og klienter er forhindret i at tilkoble sig serveren vil det ikke være muligt at lukke den ned på normal vis. Årsagen til dette skyldes designet af serverens eller mere præcist designet af accept-løkken. Denne blokerer så længe den ikke har modtaget to klienter, hvorfor brugeren er nødsaget til at *force-close* serveren igennem brugen af for eksempel ctrl+alt+delete i Windows. Grundet projektets begrænsede tidsramme blev dette problem dog ikke nået løst. En eventuel løsningen på dette problem kunne være at uddelegere denne funktion til en tråd der kører parallelt med af resten af servere.
* **Mislykket forbindelse mellem server klient:** Igennem brugen af en *black-box* test blev nødvendigheden af metoden isConnected() klarlagt, idet klienterne kunne sende beskeder via chat-funktionen samt andre funktioner der var afhængige af en forbindelse selvom forbindelsen var mislykket. Resultatet af dette var at klienterne fremprovokerede fejl der kunne gøre spillet ustabilt og til tider bryde dem ned. Det blev, på baggrund af denne test, derfor klart at der skulle testes om hvorvidt klienten i realiteten havde forbindelse til serveren igennem brugen af isConnected()-metoden *inden* brugen af disse funktioner blev tilgængelig for klienten.

På baggrund af disse netop beskrevne afprøvningsmetoder blev der lokaliseret og blottelag design og implementerings fejl som med stor sandsynlighed ellers ikke ville være opdaget. Disse metoder er dog ikke udtømmende og har derfor med sikkerhed ikke gjort programmellet fejlfrit. *Othello-Net* har dog, på baggrund af det opstillede krav, opnået et tilfredsstillende stabilitetsniveau som gør det muligt for to normalbrugere at afvikle et spil.

# Diskussion

I det følgende kapitel vil den endelige prototype af *Othello-Net* blive diskuteret. Dette indebærer nærmere en diskussion af om hvorvidt *Othello-Net* opfylder de fremsatte kravspecifikationer. I forlængelse af dette vil alternative løsninger på kravspecifikationen ligeledes blive diskuteret.

Projektets opstillede problemstilling har i dette forløb været at:

*”… designe og implementere en version af spillet Othello i en sådan grad at det vil være muligt for to normalbrugere at kunne afvikle et spil over et netværk i henhold til de gældende regler for Othello.”*

I forlængelse af dette kan det udledes at prototypen, *Othello-Net,* overordnet set har løst denne opgave idet programmellet tillader to normalbrugere at afvikle spillet Othello over et netværk. Den anvendte løsningsmodel har tillige resulteret i visse problematikker, hvorfor det kan diskuteres om det er den mest optimale løsning. Årsagerne til disse problematikker samt alternative løsningsmuligheder vil nu prøve at blive fremført i henhold til projektets problemstilling.

### Projektforløbet

Projektforløbet har i høj grad været præget af principperne som defineret i procesmodellen iterativ udvikling samt udviklingsmetoden *throwaway-prototyping*. Dette udviklingsmodel har været medvirkende årsag til en fleksibel udviklingsprocessom løbende har kunnet indbefatte ændringer specielt i henhold kravspecifikationen funktionelle aspekter.

En ulempe ved denne udviklingsmodel er manglen af fokus på ikke-funktionelle krav, idet formålet med de indledende prototyper, som nævnt, fortrinsvist er at identificere funktionelle krav. Denne tendens har tillige været problematisk i henhold til dette projektforløb idet prototyperne kun i mindre grad har haft fokus på ikke-funktionelle aspekter som for eksempel det overordnede design og brugergrænsefladen hvilket tillige har påvirket den endelige prototype af *Othello-Net*.

En anden fælde ved denne udviklingsmetodik og som tillige har været fremtrædende i dette projektforløb, er at man kan risikere at bruge for lang tid på udviklingen af de initele prototyper og dermed risikere en nedprioritering af ikke-funktionelle krav, hvorfor en effektiv udnyttelse af *throwaway-prototyping* fordrer en relativ hurtig udvikling af disse prototyper. Manglen af en hurtig udviklingsproces af de indledende prototyper har i dette udviklingsforløb bevirket relative sene beslutninger vedrørende *Othello-Net*s overordnede designmønster og dens brugergrænseflade.

### Brugergrænsefladen

Blandt de væsentligste problematikker ved denne prototype kan nævnes den manglende brugervenlighed for normalbrugere. Dette gør sig specielt gældende i opstarten af serveren og den efterfølgende synkronisering med klienterne. I denne fase kræves det nemlig at serveren startes før klienten. Dette er i sig selv ikke nødvendigvis mindre intuitivt, men grundet integrationen af klientdelen og spildelen, kan det virke ulogisk ikke at initiere spillet først for derigennem at initiere eller finde en server og/eller en klient igennem brugen af en samlet brugergrænseflade. En sådan integration af serveren, spillet og klientens brugergrænseflader ville i højere grad tilnærmer sig opfyldelsen af kravet om en intuitiv burgergrænseflade som specificeret i kravspecifikationen.

### Designet

Designet af det overordnede program blev som påpeget først besluttet relativt sent i dette projektforløb. Derfor har alternative overvejelser omkring dette emne begrænset sig til designmønstre som enten er afarter af *MVC*-mønstret eller tæt relaterede til dette. Ved at anvende alternative designmønstre end de førnævnte kunne klienten i højere grad implementeres på baggrund af principperne om objektorienteret programmering for eksempel mindre afhængighed af klasserne imellem. Et sådan designmønster kunne for eksempel være *layer*-designmønstret (Larman, 2002, p. 273f).

### Netværksstrukturen

Netværksstrukturen i *Othello-Net* kan i henhold til kravspecifikationerne anses som værende tilfredsstillende idet den udfører de funktionelle krav som er blevet opstillet, deriblandt muligheden for at kunne spille Othello og samtidig være i stand til at kommunikere med hinanden, igennem en chat-funktion, uafhængigt af spillets gang. Den sidstnævnte funktion er gjort mulig ved brugen af flere tråde (eng. *Multithreading*). I forlængelse af denne implementering er der gjort overvejelser om hvordan denne multitrådede server skulle designes, mere specifikt om serveren skulle gøre brug af et såkaldt *non-blocking* eller *blocking*-design i behandlingen af de enkelte klienter.

*Non-blocking* designet, som tilnærmelsesvist er anvendt i dette projektforløb, indebærer at man tildeler hver enkel indkommende klient en tråd hvorfor deres tråde forbliver aktive selvom de ikke modtager eller sender beskeder (eng. *one-thread-per-client*). *Blocking*-designet plæderer i højere grad for genbrugen af tråde igennem anvendelsen af *threadpools.* *Threadpools* sætter en øvre grænse for hvor mange tråde serveren må lave, hvorfor det bliver nødvendigt at blokere de enkelte klienter når de er inaktive, for eksempel når de ikke modtager noget input. I mellem tiden vil klienter som modtager beskeder eller skal sende noget anvende en af trådene i *threadpoolen* hvis sådan en er tilgængelig. Hvis det sidste ikke er tilfældet bliver klienten blokeret indtil en tråd bliver ledig, det vil sige, indtil en af de andre klienter bliver inaktive (Internetkilde 11).

I dette projektforløb har jeg som nævnt benyttet mig af *non-blocking* designet, idet det har tilladt en relativ simpel implementering af kravene i kravspecifikationen. Dette skyldes at nødvendigheden for synkronisering imellem trådene ikke har været tilsted da antallet af dem er begrænset samtidig med at designet ikke fordrer dette. Dette har dog tillige medført at ressourceforbruget er steget hvorfor en implementering af et *blocking* server design ville være i stand til at mindske ressourceforbruget men i samme ombæring være relativt mere komplekst idet implementeringen af til eksempel *threadpools* fordrer en øget administrativ indsats.

I tilfælde af en eventuel videreudvikling og op-skallering af serveren kan det tillige diskuteres om hvorvidt den øgede kompleksitet af *blocking­*-serverdesigns opvejer det lavere ressourceforbrug som man normalt anser som værende hovedårsagen til anvendelsen af dette design fremfor *non-blocking*-serverdesigns. Det kan dog diskuteres om hvorvidt *blocking-*serverdesigns stadig besidder denne fordel, grundet udviklingen inden for CPU’er som på grund af deres flere kerner udnytter tråde (one-thread-per-client) relativt godt i forhold til CPU’er med kun én af slagsen (Internetkilde 10).

# Konklusion

Igennem dette projektforløb er der blevet forsøgt designet og implementeret et programmel som skulle gøre to normalbrugere i stand til at kunne spille og afvikle spillet Othello over et netværk. I henhold til denne udlægning af kravet kan det anvises at *Othello-Net* opfylder det. Der skal dog knyttes visse forbehold til denne konklusion. I den uddybende kravspecifikation påpeges det at *Othello-Net* skal være brugervenligt i sådan en grad at to normalbrugere skal være i stand til at afvikle spillet efter at have læst brugervejledningen. Dette vil også være muligt men grundet anmærkningerne som påpeget i blandt andet diskussionen mangler der en brugervenlighed i designet af brugerfladerne. Derudover er der tillige betydelige forbedringsmuligheder i henhold til det overordnede design af såvel *Othello-Net* og dens netværksstruktur. Dertil kan der ligeledes peges på et forbedringspotentiale i henhold til selve implementeringen af *Othello-Net*, hvis klasser mangler abstraktion og dermed muligheden for eventuel genbrug – der er altså mangler når det kommer til de polymorfiske aspekter af programmellet.

Årsagen til de beskrevne mangler grunder i at udviklingen af de indledende prototyper har optaget en for stor del af udviklingsprocessen samlet set, hvorfor hovedparten af udviklingsprocessen har haft fokus på de funktionelle krav og en afledt effekt af dette har været en nedprioritering af de ikke-funktionelle krav.

I henhold til den overordnede problemstilling kan det derfor konkluderes at opfyldelsen er tilfredsstillende men dette til trods, mangler de førnævnte aspekter, som defineret i kravspecifikationen, til en hvis grad at blive opfyldt.