

Course 02199 Introductory Programming  
IMM, Technical University of Denmark  
Mandatory Assignment 2

Class: andre2199  
Joakim Nygård (s032833),  
Jacob Oettinger (s031829)

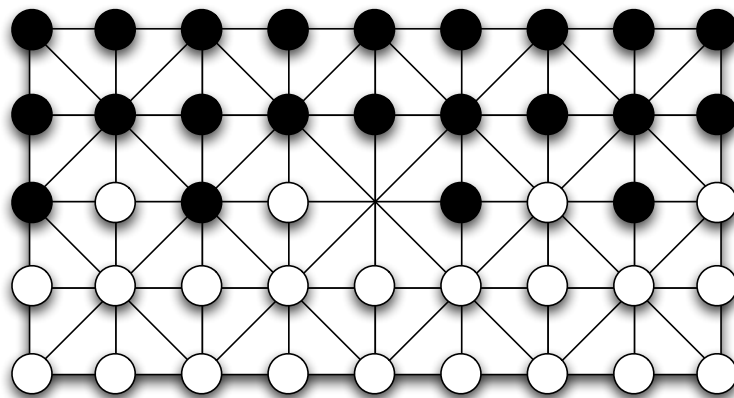
22. januar 2004

# Indhold

<b>1</b>	<b>Introduktion</b>	<b>2</b>
<b>2</b>	<b>Problemanalyse</b>	<b>2</b>
<b>3</b>	<b>Design og implementering</b>	<b>3</b>
3.1	Model-aspekter . . . . .	4
3.2	Visnings-aspekter . . . . .	5
3.3	Styrings-aspekter . . . . .	5
<b>4</b>	<b>Afprøvning</b>	<b>6</b>
<b>5</b>	<b>Konklusion</b>	<b>7</b>
<b>A</b>	<b>Brugervejledning</b>	<b>8</b>
<b>B</b>	<b>Demonstrationsplan</b>	<b>9</b>
<b>C</b>	<b>Tests</b>	<b>10</b>
C.1	Resultater . . . . .	10
<b>D</b>	<b>Programkode</b>	<b>11</b>
D.1	FanoronaModel . . . . .	11
D.2	FanoronaFrame . . . . .	11
D.3	FanoronaPanel . . . . .	11
D.4	FanoronaBoard . . . . .	11
D.5	Const . . . . .	11
D.6	Fanorona . . . . .	11
D.7	FanoronaTest . . . . .	11

# 1 Introduktion

Vi har valgt at udvikle en Javabaseret computerversion af brætspillet Fanorona med grafisk brugerflade. Programmet skal gøre det let for brugeren at foretage træk, sørge for at spillets regler overholdes og angive, når en af spillerne har vundet. Det skal være muligt at gemme og hente spil, og spille videre på disse.



Figur 1: Fanoronas startopstilling med toogtyve sten i hver farve.

## 2 Problemanalyse

Fanorona er for to spillere, der skiftes til at flytte en sten i et forsøg på at eliminere alle modstanderens sten eller bringe denne i en situation, så vedkommende ikke kan flytte sidste sten uden at blive slået i næste tur. Spillet foregår på en plade med 5x9 felter, indbyrdes forbundet af linier langs hvilke træk er mulige. Vi besluttede hurtigt, at et 2-dimensionelt array var den mest overskuelige måde at repræsentere brættets mulige positioner på. En alternativ løsning kunne være et tal på 45 cifre, hvor hvert ciffer svarer til den position, der fremkommer, når man tæller fra række til række startende i øverste venstre hjørne af spillepladen. Sidstnævnte giver mulighed for en række matematiske operationer, når stenene skal flyttes<sup>1</sup>, men den koordinaterede notation, det dobbeltdimensionerede array giver, er lettere at overskue rent udviklingsmæssigt og fordi brættet her er naturligt opdelt i rækker, kan træk beregnes enkelt.

<sup>1</sup>De omkringliggende felter kan, kort fortalt, findes ved at se på de omkringliggende cifre (vandret), cifre med en position, der ligger 9 henholdsvis før og efter (lodret) samt cifre 8 og 10 decimalpositioner før og efter (skråt). Rækkeskift findes hurtigt ved modulo med 9.

Et træk vil fjerne en ubrudt række af modstanderens sten, der ligger på samme linie, som trækket foregår langs, såfremt man rykker frem til en sten af modsat farve eller bort fra en, man stod ved. Sker begge dele, præsenteres spilleren for et valg mellem disse to typer eliminationstræk<sup>2</sup>. Programmet skal altså afgøre, om en sten overhovedet kan vælges og om den kan flyttes til det af spilleren angivne felt. Det er tilfældet, hvis der i umiddelbar nærhed og langs en af brættets linier - evt. hvor spilleren markerede - findes et tomt felt. Der er maksimalt otte omkringliggende felter at betragte og disse er hardcodet som otte if-sætninger<sup>3</sup>.

Afslutningen af spillet kræver specielt mange overvejelser, fordi det her er nødvendigt at overveje, hvorvidt hvert muligt træk lader sig gøre uden at udsætte sin sten for fare, og her er det nødvendigt at overveje ikke bare tilstødende tomme felter (en mulig position efter trækket), men også tilstødende til disse (på nær den placering, vi kom fra) med en radius på to positioner, fordi mulige farer består i både de af modstanderens sten, der kan rykke væk, samt de, der kan rykke nærmere. Det vil altså sige, at vi maksimalt skal tjekke  $8 \cdot 7 \cdot 2 = 112$  positioner. Heldigvis er det kun nødvendigt at køre denne del af programmet, hvis en af spillerne kun har en enkelt sten tilbage<sup>4</sup>.

### 3 Design og implementering

Vi har arbejdet ud fra MVC-ideen og forsøgt at adskille de interne datastrukturer fra den grafiske representation samt ladet en stor del af styringen ligge i en helt tredje klasse, som skitseret på figur 2. Vi har altså tre overordnede klasser, der håndterer hver deres del af spillet. FanoronaModel sørger for at reglerne bliver overholdt og spillets øvrige forløb. FanoronaBoard tager sig af den grafiske del af spillet og tager imod input fra musen (ved BoardListen), der herefter bliver sendt videre til modellen. FanoronaFrame starter selve spillet op, initialiserer de førnævnte modeller og håndterer input fra

---

<sup>2</sup>I noterne til spillet på <http://www.imm.dtu.dk/courses/02100/Project/fanorona.html> beskrives de to muligheder som henholdsvis indfangning ved pres og indfangning ved sug. I denne rapport så vel som i koden vil vi benytte den mere mundrette formulering at trække og skubbe (på engelsk pull and push).

<sup>3</sup>De 8 felter kan findes med udgangspunkt i den analyserede stens koordinat ved henholdsvis at trække 1 fra eller lægge til rækken (vandret og første indgang i det 2-dimensionale array), kolonnen (lodret og anden indgang) eller dem begge (skrå retning). Disse trin kunne implementeres som to løkker. Vi skal selvfølgelig sørge for ikke at komme ud over kanten på brættet, det ville give en exception af type `arrayOutOfBoundsException`.

<sup>4</sup>Vi havde allerede implementeret den oprindelige formulering af afslutningsbetingelserne, da disse blev ændret. De nye regler betyder, at kun den aktive spiller skal have en enkelt sten tilbage, altså en halvering af beregningerne (fra maksimalt 224 til 112). Det turde være klart, at remi aldrig vil opstå, fordi mindst en af spillerne altid vil kunne flytte væk fra truslen. Vi besluttede at beholde algoritmen, da den ikke påvirker spillets udfald og ret beset var en del af opgaven, da denne blev fremlagt. Det er en smal sag at modificere den i henhold til den nye formulering.

menubjælken (v.h.a. FListen). En sidste klasse, FanoronaPanel, er ansvarlig for at tegne statusbaren nederst i vinduet og kaldes af FanoronaBoard, når brættet skal tegnes.

### 3.1 Model-aspekter

Brættet repræsenteres internt af et 2-dimensionelt array af heltal<sup>5</sup>. Det giver os mulighed for på koordinatform at angive en placering på brættet. Hver indgang i arrayet kan have 1 af 3 mulige værdier, der angiver henholdsvis en hvid sten, en sort eller et tomt felt. Vi har to variable til angivelse af koordinatet for en evt. valgt sten. Tilsvarende har vi heltalsvariable til at holde styr på, hvis tur det er, hvorvidt spillet er slut samt antallet af hver spillers sten. Sidstnævnte benytter vi til løbende status nederst i spillets vindue, så spillerne kan se, hvem der fører. I dette felt vil også fremgå evt. beskeder relateret til spillets gang.

Der er to primære metoder, der afgør, hvorvidt et træk er gyldigt og hvorvidt en sten kan vælges inden trækket foretages.

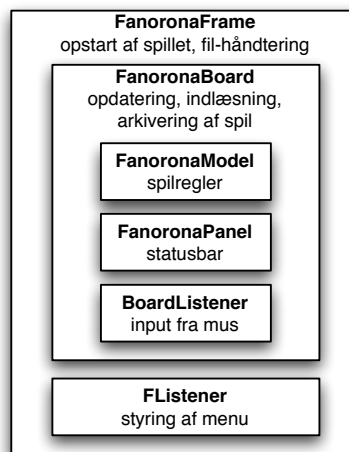
Metoden canSelect() afgør, hvorvidt en sten kan flyttes til en tom position i umiddelbar nærhed og således kan vælges. Der er maksimalt otte mulige retninger at flytte en valgt sten i. Algoritmen består her i en række if-sætninger, der samlet set returnerer sandt, hvis stenen kan vælges.

isLegalMove() afgør om et ønsket træk kan foretages langs en af de foruddefinerede linier på spillebrættet v.h.a. en matematisk repræsentation af liniernes placering i forhold til den valgte sten. Der findes skrå linier på brættet i de positioner, hvor begge koordinater er enten lige eller ulige. Samtidig er det krævet, at stenen kun flyttes en position ad gangen.

checkGameOver() kører hver gang der skiftes aktiv spiller. Funktionen skal afgøre, hvorvidt en af spillerne har tabt. Den komplicerede del eksekveres, hvis en eller begge spillere kun har en enkelt sten tilbage. I denne situation kan stenene være placeret, så den aktive spiller ikke kan flytte uden at gøre det muligt for modstanderen at eliminere<sup>6</sup> stenen ved næste træk. I en sådan situation har vedkommende tabt ved en form for skakmat.

<sup>5</sup>Integers. Selvom denne variabeltypes kapacitet ikke er optimalt udnyttet i vores anvendelse, er det i praksis ubetydeligt med vore dages computere. I andre sammenhænge havde hukommelsesoptimering måske kunnet betale sig.

<sup>6</sup>Indfange, capture.



Figur 2: En enkel hierarkisk fremstilling af de vigtigste klasser.

Algoritmen skal her bedømme faren ved enhvert muligt træk ved at se, om modstanderen ved hver af disse nye positioner kan fange stenen ved enten at skubbe eller trække den bort.

Filformatet til gemte spil afspejler vores valg af variabeltyper til repræsentation af spillets tilstand. Et gemt spil består således af 46 tegn i ascii-format, 45 til hver af spillebrættets positioner samt et enkelt til angivelse af den aktive spiller. For at minimere forbehandling af data ved indlæsning, gemmes hvert tegn med den pågældende heltalsværdi som ascii-kode og den enkelte fil er således ikke umiddelbart læselig. Alternativet, at gemme som ascii-repræsentationen af heltalsværdien ville kræve, at vi inden opsætning af spillet konverterede<sup>7</sup> tilbage til heltal. Der foretages ikke nogen validering af den indlæste data og det er derfor principielt muligt at redigere et gemt spil med ubrugelig opstilling som resultat.

### 3.2 Visnings-aspekter

Brættet tegnes, når dette anses nødvendigt af Javas swing-klasse (f.eks. ved resize af vinduet) og når spilleren foretager et træk. `repaint()`-metoden får adgang til brættets tilstand ved at spørge modellen om farven<sup>8</sup> på en evt. sten på hver af de 45 mulige positioner. En valgt sten farves grå i modsætning til spillernes øvrige sten, der er henholdsvis sorte og hvide. Samtidig opdateres statusfeltet i bunden af vinduet med `point` (antal resterende sten), hvis tur det er og evt. meddelelser fra modellen (ugyldige træk f.eks.). Når spillet er i gang, foretages træk ved at klikke på den sten, man ønsker at flytte og herefter på den tomme plads, man ønsker at flytte til. Hvis et træk giver mulighed for at fjerne modstanderens sten ved såvel at skubbe som at trække, vises en dialog med mulighed for at vælge den ønskede type træk.

### 3.3 Styrings-aspekter

Der er to klasser til styring af spillets gang, `FListen` og `BoardListen`<sup>9</sup>, der tager sig af henholdsvis input fra menuen og fra musen (se figur 3). Mere processororienteret kan man sige, at `FListen` tager sig af at starte nye spil, gemme igangværende og åbne dem igen. `BoardListen` ser efter klik på brættet og er således ansvarlig for at træk kan foretages.

Modellen for spillet indeholder til ethvert tidspunkt et sæt variable, der entydigt beskriver spillets tilstand. Denne tilstand ændres kun ved et kald fra `BoardListen`, når musen benyttes til at flytte sten, eller `FListen`, hvis et nyt eller gemt spil indlæses.

---

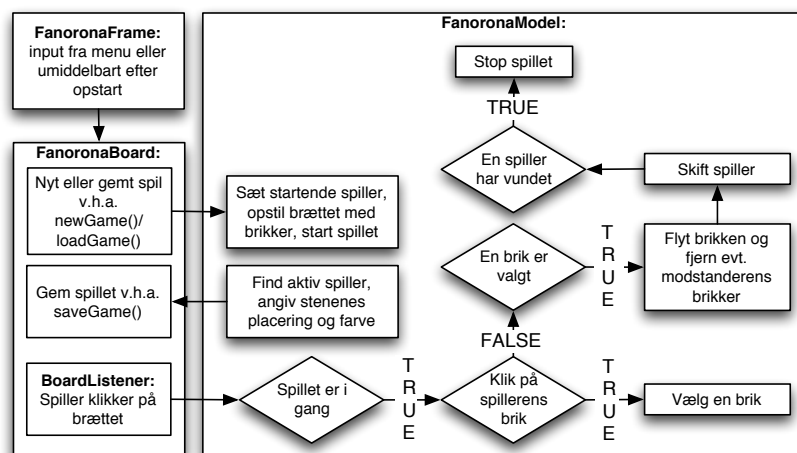
<sup>7</sup>F.eks. ved subtraktion af 48 fra asciiværdien, idet 0 er repræsenteret i ascii som 48.

<sup>8</sup>Strengt taget er der ikke tale om farver men talværdier på hver position i arrayet, der repræsenterer brættet. To af værdierne vises for brugeren som henholdsvis sort og hvid, den sidste som en ledig plads.

<sup>9</sup>Findes i henholdsvis `FanoronaFrame` og `FanoronaBoard`-klasserne, idet deres funktion anses at være nærmest knyttet disse to.

Hvis spilleren forsøger at foretage et ugyldigt træk, ændres spillets tilstand ikke, istedet sættes en variabel med forklarende tekst<sup>10</sup>. Denne tekst hentes ud af modellen, når brættet ønsker at gentegne sig selv. Ved ethvert gyldigt træk nulstilles denne tekst, så fejlmeddelelsen bliver fjernet.

Ønsker spillerne at gemme deres spil, er dette muligt fra menuen. En i forhold til styresystemet standard gem-dialog giver brugeren mulighed for at gemme spillet som en .fan fil v.h.a. JFileChooser-klassen. FListen kalder brættets saveGame(), der henter spillets tilstand fra modellen, og gemmer det med en FileWriter-instans. Det tilsvarende sker, når man åbner et gemt spil, idet loadGame() her kaldes med en array af chars fra FileReaderen.



Figur 3: Forsimplet skitse af programmets håndtering af brugerinput.

## 4 Afprøvning

Vi har under hele udviklingsfasen testet koden ved funktionelle test. Dels har vi afprøvet spillet fra start til slut for at undersøge stabilitet, dels har vi håndskrevet en række gemte spil med specielle situationer for at teste f.eks. metoden til afgørelse af spillets afslutning. Endvidere har vi outputtet mellemresultater fra en lang række metoder til terminalen under kørsel og således foretaget en form for strukturel test, idet vi for hvert metodekald har været i stand til at forudsige det forventede output. Vi har foretaget en komplet strukturtest af metoden isLegalMove() fra klassen FanoronaModel uden at finde fejl.

<sup>10</sup>I stil med "Du kan ikke flytte hertil". I en mere udviklet udgave af spillet vil det være fornuftigt at placere disse meddelelser i en tekstfil istedet for, som her, at hardcode det ind i programmet. Det letter oversættelser til andre sprog og fejlrettelser.

Så vidt vi har kunnet afgøre, fungerer vores spil uden fejl. Programmet fanger ugyldige træk og afslutter spillet korrekt, når dette er forventet.

## 5 Konklusion

Vi mener, det er lykkedes os at udvikle en velfungerende udgave af Fanorona med en overskuelig brugerflade. Vi har med succes kørt spillet på Windows, Linux og Mac OS X, og netop platformsuafhængigheden er en styrke ved Javabaseret udvikling<sup>11</sup>. Samtidig har vi med Javas Swing-API hurtigt fået en brugbar, interaktiv brugerflade op at køre.

Vi har, som nævnt ovenfor, med succes foretaget en struktureltest af den metode i modellen, der afgør, om et træk er lovligt. I fejlfindingsprocessen har vi outputtet mellemresultater fra flere forskellige metoder til terminalen og har således kunnet følge programmets flow under kørsel. Det kan betragtes som en uformel strukturel test af de pågældende funktioner. Vi har under udviklingen spillet talrige testspil og således udført funktionelle tests af programmet. De sidste kørsler har ikke givet anledning til rettelser.

Der er rig mulighed for forbedring af spillets udformning. Tilføjelse af en valgfri, computerstyret spiller ville give lejlighed til at spille alene mod computeren og inklusion af mulighed for at fortsætte sit træk<sup>12</sup> ville gøre spillet mere komplet i forhold til det oprindelige. Det skal bemærkes, at mens spillets menuer og fejlmeddelelser er på engelsk, har vi skrevet brugervejledningen på dansk, da den skulle inkluderes i nærværende rapport. Det ville være oplagt i en udvidet version, at placere al tekst i separate filer, så disse let kunne oversættes. Endelig ville netværksunderstøttelse med mulighed for at spille online mod andre øge underholdningsværdien.

---

<sup>11</sup>Omind det med omtanke og en passende mængde erfaring også lader sig gøre i andre sprog som f.eks. C++ (Se <http://www.gdconf.com/archives/2000/malenfa.doc> for en diskussion af dette).

<sup>12</sup>Se evt. <http://www.imm.dtu.dk/courses/02100/Project/fanorona.html>

## A Brugervejledning

Spillet åbnes ved enten at køre følgende kommando fra terminalen: `java -jar s031829.jar`. På visse platforme kan man dobbeltklikke på `.jar`-filen<sup>13</sup>.

Når spillet åbnes, startes automatisk et nyt spil og hvid spiller kan straks begynde med første træk.

Den sten, der ønskes flyttet, markeres med musen. Herefter klikkes på en tilstødende, tom position og den valgte sten flyttes hertil. Evt. fangede sten fjernes automatisk, med mindre det både er muligt at skubbe og trække en række sten. I så fald bedes spilleren om at vælge det ønskede.

Nederst i spillets vindue kan antallet af hver spillers sten løbende følges ligesom evt. meddelelser om spillets status vil optræde her.

At starte et nyt spil:

For at starte et nyt spil skal man klikke på "new game" i "game" menuen

At gemme et spil:

Under et spil er det på et hvert tidspunkt muligt at gemme spillet. Dette gøres ved at trykke på "save" i "game" menuen. Herefter vil brugeren blive stillet over for en dialog hvor der skal vælges navn til det gemte spil. Hvis spillet har været gemt før vil det gamle navn allerede være valgt. Det er dog muligt at vælge et nyt navn til spillet, så det gamle ikke overskrives.

At hente et spil:

Det er muligt at hente et tidligere gemt spil, ved at klikke på "load" i "game" menuen. Herefter vil brugeren blive stillet over for en dialog, hvor der kan vælges tidligere gemte spil. Hvis et spil hentes vil et eventuelt igangværende spil blive stoppet.

At stoppe spillet:

For at stoppe spillet skal der klikkes på "quit" i "game" menuen.

---

<sup>13</sup>Testet på Windows XP og Mac OS X.

## B Demonstrationsplan

Ved demonstrationen af spillet har vi planlagt nedenstående træk til illustration af spillets funktionalitet. Trækkene er ikke en udtømmende funktionel test, men skulle give et indtryk af, at spillets regler overholdes.

Demonstrationen forberedes med kommandoen: `jar -xvf s031829.jar *.fan`

Formål	Handling	Resultat
Lovligt træk	Flyt fra (4,5) til (3,5)	2 sten forsvinder
Ulovligt træk	Flyt fra (1,4) til (4,5)	Status: "Can't move here"
Forberedelse	Flyt (1,4) til (1,5)	
	Flyt (3,5) til (3,2)	
	Flyt (2,6) til (3,5)	
Træk/skub	Flyt (2,5) til (2,6), vælg pull	4 sten forsvinder
Gem	Gem som "demo"	Titel ændres
	Nyt spil	Titel ændres
Åbn	Åbn "demo"	Sten placeres som før
Afslutning	Åbn "whitelooser.fan"	
	Flyt sort sten ned	Status: "White lost"
Afslutning	Åbn "whitelooser.fan"	
	Flyt sort sten til højre	
	Flyt hvid sten ned	Status: "White player wins"

## C Tests

Der er gennemført en strukturel test af metoden `isLegalMove()` fra klassen `FanoronaModel`. Testen bliver gennemført af det lille program `FanoronaTest`, til hvilket kildekoden findes som bilag. Allerede i et så relativt simpelt program som dette bliver det tydeligt, hvor svært det kan være at eliminere fejl. Strukturelle tests hjælper til dette og skulle programmet sendes på gaden, ville det være fornuftigt at foretage sådanne tests af flere andre metoder, her iblandt `checkGameOver()`. Kombineret med betatest foretaget af egentlige spillere ville det minimere risikoen for et fejlfyldt slutprodukt.

### C.1 Resultater

Output fra terminalen:

```
Starting structural test of method isLegalMove()...
```

---

```
Move from (0,0) to (-1,0) as expected: false
Move from (0,0) to (0,-1) as expected: false
Move from (0,0) to (0,0) as expected: false
Move from (4,8) to (4,9) as expected: false
Move from (4,8) to (5,9) as expected: false
Move from (0,0) to (1,1) as expected: true
Move from (0,0) to (0,1) as expected: true
Move from (0,0) to (1,0) as expected: true
move from (0,1) to (1,2) as expected: false
move from (0,0) to (2,0) as expected: false
Move from (0,0) to (0,2) as expected: false
Move from (0,0) to (2,2) as expected: false
```

---

The test succeeded! The method `isLegalMove()` is working.

## D Programkode

### D.1 FanoronaModel

```
1 import javax.swing.JLabel;
2 import javax.swing.JOptionPane;
3
4
5 /**
6  * FanoronaModel 1.24, 2004/01/19
7  *
8  * This class is responsible for the gameplay and rules.
9  *
10 * ©Joakim Nyg&aring;rd and Jacob Oettinger
11 */
12 class FanoronaModel
13 {
14     private int [][] board;
15     private int currentPlayer,noOfRows,noOfCols;
16     private int activeRow,activeCol;        // selection
17     private Const defines;                 // memorable defines
18     private int blackStones, whiteStones;  // stones left
19     private int death;                     // the player that lost
20     private String statusMsg;
21
22     /**
23      * Constructor method.
24      *
25      */
26     public FanoronaModel(int ro,int co)
27     {
28         noOfRows=ro;
29         noOfCols=co;
30         board = new int[ro][co];
31     }
32
33     /**
34      * Begins a game.
35      *
36      */
37     public void startGame()
38     {
39         death = 0;
40         activeRow=-1;    // no selected piece
41         activeCol=-1;
42
43         setStatus("");
44     }
45
46     /**
47      * Ends the current game. Called by checkGameOver();
48      *
49      */
```

```

50 private void stopGame()
51 {
52     currentPlayer = defines.EMPTY;    // neither black or white
        player, so the game ends.
53 }
54
55 /**
56  * Clear the board.
57  *
58  */
59 public void resetStones()
60 {
61     blackStones = whiteStones = 0;
62 }
63
64 /**
65  * Used to load a game, loads one stone at given coordinate
        with given color.
66  *
67  */
68 public void setStone(int ro,int co, int color)
69 {
70     board[ro][co] = color;
71     if(color == defines.BLACK)
72         blackStones++;
73     else if(color == defines.WHITE)
74         whiteStones++;
75 }
76
77
78 /**
79  * Used to get game state when saving. Also for drawing the
        board.
80  * Returns color of stone at ro,co
81  */
82 public int getStone(int ro,int co)
83 {
84     return board[ro][co];
85 }
86
87 /**
88  * Called when user clicks at ro,co. Decides what action to
        take
89  *
90  */
91 public void manipStone(int ro, int co)
92 {
93     if(currentPlayer != defines.EMPTY) // not game over
94     {
95         if(board[ro][co] == currentPlayer || board[ro][co] ==
            defines.SELECTED) // click on players piece
96         {
97             if( canSelect(ro,co)) // can move
98                 selectStone(ro,co);

```

```

99         else
100             setStatus("Can't select this");
101     }
102     else if(board[ro][co] == defines.EMPTY) // click empty
103         square
104     {
105         if(activeRow!=-1 && activeCol!=-1)
106         {
107             if(isLegalMove(activeRow,activeCol,ro,co)) // can
108                 move selection here
109                 moveStone(ro,co);
110             else
111                 setStatus("Can't move here");
112         }
113     }
114     else
115     {
116         setStatus("You need to select a stone");
117     }
118 }
119 }
120
121
122 /**
123  * Selects the stone clicked by user.
124  *
125  */
126 private void selectStone(int ro,int co)
127 {
128     setStatus("");
129
130     if(activeRow!=-1 && activeCol!=-1) // got a previous
131         selection
132         board[activeRow][activeCol]=currentPlayer; // deselect
133         board[ro][co]= defines.SELECTED; // select new
134         activeRow=ro;
135         activeCol=co;
136 }
137
138 /**
139  * Moves a previously selected stone to ro,co. Coordinate
140  * should be validated by isLegalMove() first.
141  * Removes any stones captured by the move.
142  */
143 private void moveStone(int ro,int co)
144 {
145     setStatus("");
146
147     boolean push=false;
148     boolean pull=false;
149     int rowDir = ro-activeRow; // direction of move

```

```

149     int colDir = co-activeCol;
150     int pushRow = ro+rowDir;    // pushed stone
151     int pushCol = co+colDir;
152     int pullRow = activeRow-rowDir; // pulled stone
153     int pullCol = activeCol-colDir;
154     int removedStones = 0;
155
156     board[activeRow][activeCol] = defines.EMPTY; // update
        board to reflect move
157     board[ro][co] = currentPlayer;
158     activeCol = activeRow = -1;    // no selection anymore,
        the stone has moved.
159
160     if(isOnBoard(pushRow,pushCol))    // inside board
161     {
162         if(board[pushRow][pushCol] == getOpponent()) // other
            players stone
163             push=true;                // pushed
164     }
165     if(isOnBoard(pullRow,pullCol))    // inside board
166     {
167         if(board[pullRow][pullCol] == getOpponent()) // other
            players stone
168             pull=true;                // pulled
169     }
170     if(push && pull) // if both, ask.
171         push=askPush();
172
173     if(push)
174     {
175         // keep pushing stones while inside board, not current
            player's stone and not empty
176         while(isOnBoard(pushRow,pushCol) && board[pushRow][
            pushCol] == getOpponent())
177         {
178             board[pushRow][pushCol]=defines.EMPTY;
179             pushRow+=rowDir;;
180             pushCol+=colDir;
181             removedStones++;
182         }
183     }
184     else if(pull)
185     {
186         // keep pulling stones while inside board, not current
            player's stone and not empty
187         while(isOnBoard(pullRow,pullCol) && board[pullRow][
            pullCol] == getOpponent())
188         {
189             board[pullRow][pullCol]=defines.EMPTY;
190             pullRow-=rowDir;;
191             pullCol-=colDir;
192             removedStones++;
193         }
194     }

```

```

195     removeCapturedStones(removedStones);
196     changePlayer();    // next player
197 }
198
199
200 /**
201  * Checks wether user can move a stone from fromRo,fromCo to
202   toRo,toCo.
203  */
204 private boolean isLegalMove(int fromRo,int fromCo,int toRo,
205   int toCo)
206 {
207     int rowDist = Math.abs(fromRo-toRo);
208     int colDist = Math.abs(fromCo-toCo);
209
210     if(!isOnBoard(toRo,toCo)) // outside board
211         return false;
212     else if(rowDist == 1 && colDist == 1) // diagonal move
213         return((fromRo%2==0 && fromCo%2==0) || (fromRo%2!=0 &&
214             fromCo%2!=0));
215     else // straight line, return true if distance is 1
216         return((rowDist == 1 && colDist == 0) || (colDist
217             == 1 && rowDist == 0));
218 }
219
220 /**
221  * Returns true if the given coordinates are on the board
222  */
223 private boolean isOnBoard(int row, int col)
224 {
225     return((row>=0 && row<noOfRows) && (col>=0 && col<noOfCols)
226         );
227 }
228
229 /**
230  * Is user allowed to select the stone? True if there is at
231   least one empty
232   * position next to the stone and that stone can move in
233   that direction.
234  */
235 private boolean canSelect(int ro,int co)
236 {
237     boolean can=false;
238     if(ro+1<noOfRows)
239         can=(can || (board[ro+1][co]==0));
240     if(ro-1>=0)
241         can=(can || (board[ro-1][co]==0));
242     if(co+1<noOfCols)
243         can=(can || (board[ro][co+1]==0));
244     if(co-1>=0)
245         can=(can || (board[ro][co-1]==0));

```

```

242     if(ro+1<noOfRows && co+1<noOfCols)
243         can=(can || (board[ro+1][co+1]==0 && isLegalMove(ro,co,ro
                +1,co+1)));
244     if(ro+1<noOfRows && co-1>=0)
245         can=(can || (board[ro+1][co-1]==0 && isLegalMove(ro,co,ro
                +1,co-1)));
246     if(ro-1>=0 && co+1<noOfCols)
247         can=(can || (board[ro-1][co+1]==0 && isLegalMove(ro,co,ro
                -1,co+1)));
248     if(ro-1>=0 && co-1>=0)
249         can=(can || (board[ro-1][co-1]==0 && isLegalMove(ro,co,ro
                -1,co-1)));
250     return can;
251 }
252
253
254 private void removeCapturedStones(int amount)
255 {
256     if(currentPlayer == defines.BLACK)
257         whiteStones -= amount;
258     else if(currentPlayer == defines.WHITE)
259         blackStones -= amount;
260 }
261
262
263 /**
264  * The only place in the model that has GUI. We need user
        feedback to
265  * decide wether we should pull or push the stones captured.
266  * Called by moveStone().
267  */
268 private boolean askPush()
269 {
270     Object[] options = { "Push", "Pull" };
271     JOptionPane askPushPull = new JOptionPane();
272     int ans=askPushPull.showOptionDialog(null, "Want to push or
        pull?", "Push or Pull",JOptionPane.DEFAULT_OPTION,
        JOptionPane.QUESTION_MESSAGE,null, options, options[0])
        ;
273     if(ans==0)
274         return true;
275     else
276         return false;
277 }
278
279
280 /**
281  * End turn and go to next player.
282  *
283  */
284 public void changePlayer()
285 {
286     currentPlayer = getOpponent();
287

```

```

288     setStatus("");
289
290     checkGameOver();
291 }
292
293
294 /**
295  * Separate method to allow for future modifications to
296  *   statustext.
297  * Sets the internal statusMsg. Retrieved by board via
298  *   getStatus().
299  */
300 private void setStatus(String errorMsg)
301 {
302     statusMsg = errorMsg;
303 }
304
305 /**
306  * Returns the status of the game for display by board.
307  */
308 public String getStatus()
309 {
310     return statusMsg;
311 }
312
313
314 /**
315  * Returns an array with number of remaining stones for
316  *   display.
317  */
318 public int[] getScore()
319 {
320     int[] score = new int[2];
321
322     score[0] = whiteStones;
323     score[1] = blackStones;
324     return score;
325 }
326
327
328 /**
329  * Used when loading and starting games to set the correct
330  *   player.
331  */
332 public void setPlayer(int player)
333 {
334     currentPlayer = player;
335 }
336
337

```

```

338  /**
339  * Used when saving a game to get the currenctly active
      player.
340  *
341  */
342  public int getPlayer()
343  {
344      return (currentPlayer);
345  }
346
347  /**
348  * Returns the player that is not currently playing.
349  */
350  private int getOpponent()
351  {
352      if(currentPlayer == defines.BLACK)
353          return defines.WHITE;
354      else
355          return defines.BLACK;
356  }
357
358
359  /**
360  * The horror-method. Checks if we have a winner.
361  *
362  * There are three ways of ending the game:
363  * 1. One of the players has zero stones left. Other player
      wins.
364  * 2. One player has only one stone left and can't move it
      without exposing
365  *     it to capture. Other player can move.
366  * 3. Both players has only one stone left and neither can
      move without
367  *     exposing it to capture by other player. Will never
      happen, but early
368  *     project description required it and we left it as it
      was. It will not
369  *     effect gameplay.
370  *
371  */
372  private void checkGameOver()
373  {
374      boolean die = true;
375
376      // one of the players has zero stones left
377      if(blackStones == 0)
378      {
379          setStatus("White player wins!");
380          stopGame();
381      }
382      else if(whiteStones == 0)
383      {
384          setStatus("Black player wins!");
385          stopGame();

```

```

386     }
387     // special gameover if one of the players has only 1 stone
        left
388     else if(( blackStones == 1) || ( whiteStones == 1))
389     {
390         // find the last stone by looking at the board.
391         for(int i=0;i<noOfRows;i++)
392         {
393             for(int j=0;j<noOfCols;j++)
394             {
395                 // make sure it's currentPlayer's stone.
396                 if(board[i][j] == currentPlayer)
397                 {
398                     // consider every possible move to [i-r,j-c] from [
                        i,j].
399                     for(int r=-1;r<2;r++)
400                     {
401                         for(int c=-1;c<2;c++)
402                         {
403                             // considered move is good
404                             if(isLegalMove(i,j,i-r,j-c) && board[i-r][j-c
                                ] == defines.EMPTY)
405                             {
406                                 // we're not dead yet.
407                                 boolean danger = false;
408                                 // look in all directions from proposed move.
409                                 for(int nr=-1;nr<2;nr++)
410                                 {
411                                     for(int nc=-1;nc<2;nc++)
412                                     {
413                                         // don't consider the position we're at
                                            and make sure we're on the board.
414                                         if(((nr!=0 && nc==0) || (nr==0 && nc!=0)
                                                || (nr!=0 && nc!=0)) && isOnBoard(i-
                                                r-nr,j-c-nc))
415                                         {
416                                             if(isLegalMove(i-r,j-c,i-r-nr,j-c-nc))
                                                // are there any lines this way?
417                                             {
418                                                 // we're next to an empty position,
                                                    possible push situation
419                                                 if(board[i-r-nr][j-c-nc] == defines.
                                                    EMPTY)
420                                                 {
421                                                     // are we still on the board if
                                                        looking one position behind the
                                                        empty position?
422                                                     if(isOnBoard(i-r-2*nr,j-c-2*nc))
423                                                     {
424                                                         // check if opponent has a stone
                                                            further away in same
                                                            direction, if he has, we can'
                                                            t move here

```

```

425         if(board[i-r-2*nr][j-c-2*nc] ==
426             getOpponent())
427         {
428             danger = true;
429         }
430     }
431     // we're next to opponent's stone,
432     // possible pull situation
433     else if(board[i-r-nr][j-c-nc] ==
434         getOpponent())
435     {
436         if(isOnBoard(i-r-2*nr,j-c-2*nc))
437             // on the board
438         {
439             // if there is empty space behind
440             // the opponent's stone, we can
441             // 't move here
442             if(board[i-r-2*nr][j-c-2*nc] ==
443                 defines.EMPTY)
444                 danger = true;
445         }
446     }
447     }
448     }
449     }
450     }
451     }
452     }
453     }
454     // has one of the players already died during previous
455     // checkGameOver()?
456     if(death==getOpponent())
457     {
458         // if this player dies too, it must be a draw
459         if(die)
460         {
461             setStatus("It's a draw");
462             stopGame();
463         }
464         // otherwise, the first player just lost.
465         else
466         {

```

```

466         String playerName = (currentPlayer==defines.WHITE)? "
           Black" : "White";
467         setStatus(playerName+" lost.");
468         stopGame();
469     }
470 }
471 // if we're the first to die, check if other player dies
           too.
472 else if(die)
473 {
474     death = currentPlayer;
475     changePlayer();
476 }
477 }
478 }
479 }

```

## D.2 FanoronaFrame

```

1 import javax.swing.JFileChooser.*;
2 import javax.swing.*;
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.net.*;
7
8 public class FanoronaFrame extends JFrame
9 {
10     // An instance of the class FanoronaBoard is declared
11     private FanoronaBoard board;
12
13
14     public FanoronaFrame()
15     {
16         // Menubar with the needed menus and buttons
17         JMenuBar menuBar=new JMenuBar();
18         JMenu gameMenu=new JMenu("Game");
19         JMenuItem newItem=new JMenuItem("New game");
20         JMenuItem openItem=new JMenuItem("Open");
21         JMenuItem saveItem=new JMenuItem("Save");
22         JMenuItem quitItem = new JMenuItem("Quit");
23         menuBar.add(gameMenu);
24         gameMenu.add(newItem);
25         gameMenu.add(openItem);
26         gameMenu.add(saveItem);
27         gameMenu.addSeparator();
28         gameMenu.add(quitItem);
29
30         JMenu helpMenu=new JMenu("Help");
31         JMenuItem rulesItem=new JMenuItem("Rules");
32         JMenuItem guideItem=new JMenuItem("Guide");
33         JMenuItem aboutItem=new JMenuItem("About");
34         menuBar.add(helpMenu);
35         helpMenu.add(guideItem);

```

```

36     helpMenu.add(rulesItem);
37     helpMenu.addSeparator();
38     helpMenu.add(aboutItem);
39
40     // An internal listener for the "buttons" in the menus
41     FListener Fli= new FListener();
42     newItem.addActionListener(Fli);
43     openItem.addActionListener(Fli);
44     saveItem.addActionListener(Fli);
45     quitItem.addActionListener(Fli);
46     guideItem.addActionListener(Fli);
47     rulesItem.addActionListener(Fli);
48     aboutItem.addActionListener(Fli);
49
50
51     // The menubar is added to the frame
52     setJMenuBar(menuBar);
53
54     // The statuspanel is created as a FanoronaPanel
55     FanoronaPanel statusPanel = new FanoronaPanel();
56     getContentPane().add(statusPanel,"South");
57
58     // The board is initialised and the statuspanel is sent
59     // making it possible for the board to access its
60     // methods
61     board = new FanoronaBoard(statusPanel);
62     getContentPane().add(board,"Center");
63
64     // A new game is started automatically started when the
65     // program is started
66     board.newGame();
67
68 }
69
70 // Internal listener for the menu
71 class FListener implements ActionListener
72 {
73     File selectedFile; // A File for holding the currently
74     // selected file
75
76     public void actionPerformed(ActionEvent evt)
77     {
78         if (evt.getSource() instanceof JMenuItem)
79         {
80             String command = evt.getActionCommand();
81             // Take action according to the pressed button
82             if(command.equals("New game"))
83             {
84                 // Start a new game
85                 board.newGame();
86                 // Reset the title. No game loaded/saved

```

```

86         setTitle("Fanorona");
87         // Reset the selectedFile. No game loaded/
           saved
88         selectedFile = new File("");
89
90
91     }
92     if(command.equals("Open"))
93     {
94         // 45 positions on the board + current
           player
95         char[] saveGame = new char[46];
96
97         // A new filechooser for selectingthe file
           to load
98         JFileChooser FC = new JFileChooser();
99         // Add our FanFilter that filters all but .
           fan files
100        FC.addChoosableFileFilter(new FanFilter());
101        int returnVal = FC.showOpenDialog(null);
102        if(returnVal == JFileChooser.APPROVE_OPTION
           )
103        {
104            selectedFile = FC.getSelectedFile();
105            try{
106                // Read the selected file
107                FileReader fr = new FileReader(
                    selectedFile);
108                // Put the filecontents in an array
                    of chars
109                fr.read(saveGame);
110                // update the board
111                board.loadGame(saveGame);
112                // Show loaded gamename in title
113                setTitle("Fanorona - "+selectedFile
                    .getName());
114                fr.close();
115            }
116            catch(IOException e)
117            {
118                System.out.println("Problems
                    opening or reading "+
                    selectedFile.getName());
119            }
120        }
121    }
122    if(command.equals("Save"))
123    {
124        // A new filechooser for selectingthe file
           to load
125        JFileChooser FC = new JFileChooser();
126        // Add our FanFilter that filters all but .
           fan files
127        FC.addChoosableFileFilter(new FanFilter());

```

```

128         // If the current game is loaded from a
           file or allready as been saved, set the
           selected file
129     FC.setSelectedFile(selectedFile);
130     try
131     {
132         int returnVal = FC.showSaveDialog(null)
           ;
133         if(returnVal == JFileChooser.
           APPROVE_OPTION)
134         {
135             selectedFile = FC.getSelectedFile()
           ;
136             // Get the absolutepath for the
           selected file
137             String name = selectedFile.
           getAbsolutePath();
138             // If the file does not have the
           right extension add it.
139             if(!name.endsWith(".fan"))
140             {
141                 selectedFile = new File(name+".
           fan");
142             }
143
144             FileWriter fw = new FileWriter(
           selectedFile);
145             // Write the file
146             fw.write(board.saveGame());
147             // Show saved gamename in title
148             setTitle("Fanorona - "+selectedFile
           .getName());
149             fw.close();
150         }
151     }
152     catch(IOException e)
153     {
154         System.out.println("Problems writing "+
           selectedFile.getName());
155     }
156 }
157 if(command.equals("Quit"))
158 {
159     // EXIT
160     System.exit(0);
161 }
162 if(command.equals("Rules"))
163 {
164     // Show the rules in a new frame
165     JFrame RulesFrame = new JFrame();
166     RulesFrame.setTitle("Fanorona - Rules");
167     RulesFrame.setSize(400,400);
168     RulesFrame.setLocation(230,230);
169     // Read the rules from the rules.txt file.

```

```

170         // Little work-a-round needed to get file
171         from .jar
172         URL rulesFile = getClass().getResource("
173         rules.txt");
174
175         // Create not-editable JScrollPane for the
176         rules
177         JTextPane RulesPane = new JTextPane();
178         RulesPane.setEditable(false);
179         JScrollPane RulesSP = new JScrollPane(
180         RulesPane);
181
182         try{
183             // Read file with right encoding(ø&#a
184             InputStreamReader ir = new
185             InputStreamReader(rulesFile.
186             openStream(),"ISO-8859-1");
187             RulesPane.read(ir,null);
188         }
189         catch(IOException e){
190             System.out.println("The Rules.txt file
191             is not present");
192         }
193         // Add text to the frame
194         RulesFrame.getContentPane().add(RulesSP,"
195         Center");
196         RulesFrame.setVisible(true);
197     }
198     if(command.equals("Guide"))
199     {
200         JFrame GuideFrame = new JFrame();
201         GuideFrame.setTitle("Fanorona - Guide");
202         GuideFrame.setSize(400,400);
203         GuideFrame.setLocation(230,230);
204         // Read the guide from the guide.txt file.
205         // Little work-a-round needed to get file
206         from .jar
207         URL guideFile = getClass().getResource("
208         guide.txt");
209         // Create not-editable JScrollPane for the
210         rules
211         JTextPane GuidePane = new JTextPane();
212         JScrollPane GuideSP = new JScrollPane(
213         GuidePane);
214         GuidePane.setEditable(false);
215         try{
216             // Read file with right encoding(ø&#a
217             InputStreamReader ir = new
218             InputStreamReader(guideFile.
219             openStream(),"ISO-8859-1");
220             GuidePane.read(ir,null);
221         }

```

```

210         catch(IOException e){
211             System.out.println("The guide.txt file
                is not present");
212         }
213         // Add text to the frame
214         GuideFrame.getContentPane().add(GuideSP,"
                Center");
215         GuideFrame.setVisible(true);
216     }
217     if(command.equals("About"))
218     {
219         // Create an about-popup using an
                information MessageDialog
220         JOptionPane about = new JOptionPane();
221         about.showMessageDialog(null, "Fanorona -
                The national game of Madagascar\nJoakim
                Nygård & Jacob Oettinger\n2004","About
                ", JOptionPane.INFORMATION_MESSAGE);
222     }
223 }
224 }
225
226 }
227
228
229
230 }
231 /**
232  * A FileFilter for the open and save dialogs
233  */
234 class FanFilter extends javax.swing.filechooser.FileFilter {
235     public boolean accept(File file)
236     {
237         String filename = file.getName();
238         return filename.endsWith(".fan"); // Returns true if
                the questioned fil has the right extension
239     }
240     public String getDescription()
241     {
242         return "Fanorona (*.fan)"; // The description of our
                filetype
243     }
244 }

```

### D.3 FanoronaPanel

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 /**
5  * FanoronaPanel 1.1, 2004/01/15
6  *
7  * Create and display status messages in the statusbar.
8  *

```

```

9  *   ©Joakim Nygård and Jacob Oettinger
10 */
11 public class FanoronaPanel extends JPanel
12 {
13     JLabel player,status,score;    // three types of feedback
14
15     public FanoronaPanel()
16     {
17         // Set the panels layout to gridlayout
18         GridLayout layout = new GridLayout(1,3);
19         this.setLayout(layout);
20
21         // Add labels for information to the panel
22         player = new JLabel("",SwingConstants.CENTER);
23         this.add(player);
24         status = new JLabel("",SwingConstants.CENTER);
25         this.add(status);
26         score = new JLabel("",SwingConstants.CENTER);
27         this.add(score);
28     }
29     // Set the textlabel
30     public void setStatus(String text)
31     {
32         status.setText(text);
33     }
34     // Set the playerlabel
35     public void setPlayer(String text)
36     {
37         player.setText(text);
38     }
39     // Set the score according to two integers
40     public void setScore(int white, int black)
41     {
42         score.setText("W:"+white+" B:"+black);
43     }
44 }

```

#### D.4 FanoronaBoard

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5
6  /**
7   * FanoronaBoard 1.1, 2004/01/19
8   *
9   * This class is responsible for handling mouse actions and sets
10  * up
11  * the board when loading or starting a new game. Primary
12  * function is
13  * to draw the board and the stones on it.
14  *
15  * ©Joakim Nygård and Jacob Oettinger

```

```

14 */
15 class FanoronaBoard extends JPanel
16 {
17     private FanoronaModel BoardM; // this is the model, handles
        all gameplay
18     private FanoronaPanel panel; // statusbar at the bottom of
        the window
19     private Const defines; // shortcuts
20     private int gameRows = 5; // size of the board
21     private int gameCols = 9;
22
23     /**
24      * Constructor method.
25      * Create a new game with correct dimension and set up
        listener
26     */
27     public FanoronaBoard(FanoronaPanel parentStatus)
28     {
29         BoardM = new FanoronaModel(gameRows,gameCols);
30         panel = parentStatus;
31         BoardListener blis = new BoardListener();
32         addMouseListener(blis);
33         setBackground(Color.orange);
34     }
35
36     /**
37      * Load a saved game.
38      * Set up the board with stones as indicated by the array of
        chars
39     */
40     public void loadGame(char [] gameData)
41     {
42         int pos = 1;
43
44         BoardM.setPlayer(gameData[0]); // who starts this
            savegame
45         BoardM.resetStones(); // clear the counter of each
            player's stones
46
47         for(int r=0;r<gameRows;r++)
48         {
49             for(int c=0;c<gameCols;c++)
50             {
51                 BoardM.setStone(r,c,gameData[pos]); // for each char in
                    the file, place the stones
52                 pos++;
53             }
54         }
55         BoardM.startGame(); // start the game :)
56         repaint();
57     }
58
59
60     /**

```

```

61     * Create a new game.
62     *
63     */
64     public void newGame()
65     {
66         BoardM.setPlayer(defines.WHITE); // white begins
67         BoardM.resetStones();           // clear the counter of each
            player's stones
68
69         for(int r=0;r<gameRows;r++)     // initialize board
70         {
71             for(int c=0;c<gameCols;c++)
72             {
73                 if(r<(gameRows/2) || (r==(gameRows/2) && c%2==0 && c!=(
                    gameCols/2)))
74                     BoardM.setStone(r,c,defines.BLACK);
75                 else if(r>(gameRows/2) || (r==(gameRows/2) && c%2!=0))
76                     BoardM.setStone(r,c,defines.WHITE);
77                 else
78                     BoardM.setStone(r,c,defines.EMPTY);
79             }
80         }
81
82         BoardM.startGame(); // begin...
83         repaint();
84     }
85
86     /**
87     * Returns an array of chars with info on active player and
            position of pieces
88     *
89     */
90     public char[] saveGame()
91     {
92         char[] state = new char[46]; // 45 positions on the board
            + current player
93         int counter = 1;
94
95         state[0] = (char)BoardM.getPlayer();
96         for(int r=0;r<gameRows;r++)
97         {
98             for(int c=0;c<gameCols;c++)
99             {
100                 int corrected = BoardM.getStone(r,c);
101                 if(corrected == defines.SELECTED) // don't save
                    selections
102                     corrected = BoardM.getPlayer();
103                 state[counter] = (char)corrected;
104                 counter++;
105             }
106         }
107         return state;
108     }
109

```

```

110
111  /*
112   * Draws the boardlines, stones and any selection. Also
113   * updates the statuspanel
114   */
115 public void paintComponent(Graphics g)
116 {
117     super.paintComponent(g);
118
119     int width  = this.getWidth();
120     int height = this.getHeight();
121     int colWidth  = width/(gameCols+1);
122     int rowHeight = height/(gameRows+1);
123
124     int bigCols=(gameCols-1)/2;
125     int bigRows=(gameRows-1)/2;
126
127     // draws the boardlines
128     for(int bR=0; bR<bigRows;bR++)
129     {
130         for(int bC=0; bC<bigCols;bC++)
131         {
132             g.drawLine( bC*colWidth*2+colWidth, bR*rowHeight*2+
133                 rowHeight, (bC+1)*colWidth*2+colWidth, bR*rowHeight
134                 *2+rowHeight); // Top linie
135             g.drawLine( bC*colWidth*2+colWidth, bR*rowHeight*2+2*
136                 rowHeight, (bC+1)*colWidth*2+colWidth, bR*rowHeight
137                 *2+2*rowHeight); // midt linie
138             g.drawLine( bC*colWidth*2+colWidth, bR*rowHeight*2+
139                 rowHeight, bC*colWidth*2+colWidth, (bR+1)*rowHeight
140                 *2+rowHeight); //Venstre kant
141             g.drawLine( bC*colWidth*2+2*colWidth, bR*rowHeight*2+
142                 rowHeight, bC*colWidth*2+2*colWidth, (bR+1)*
143                 rowHeight*2+rowHeight); // midt kant
144             g.drawLine( bC*colWidth*2+colWidth, bR*rowHeight*2+
145                 rowHeight, (bC+1)*colWidth*2+colWidth, (bR+1)*
146                 rowHeight*2+rowHeight); // kryds
147             g.drawLine( (bC+1)*colWidth*2+colWidth, bR*rowHeight*2+
148                 rowHeight, bC*colWidth*2+colWidth, (bR+1)*rowHeight
149                 *2+rowHeight); //kryds
150         }
151     }
152     g.drawLine( colWidth, bigRows*rowHeight*2+rowHeight,
153         bigCols*colWidth*2+colWidth, bigRows*rowHeight*2+
154         rowHeight); // bund
155     g.drawLine( bigCols*colWidth*2+colWidth, rowHeight, bigCols
156         *colWidth*2+colWidth, bigRows*rowHeight*2+rowHeight);
157         // h-jre
158
159     // puts the stones in place with correct color
160     for(int r=0;r<gameRows;r++)
161     {
162         for(int c=0;c<gameCols;c++)
163         {

```

```

147         if(this.BoardM.getStone(r,c)==defines.WHITE)
148             g.setColor(Color.white);
149         else if(BoardM.getStone(r,c)==defines.BLACK)
150             g.setColor(Color.black);
151         else if(BoardM.getStone(r,c)==defines.SELECTED)
152             {
153                 if(BoardM.getPlayer()==defines.WHITE)
154                     g.setColor(Color.lightGray);
155                 else
156                     g.setColor(Color.gray);
157             }
158         if(BoardM.getStone(r,c)!=defines.EMPTY)
159             {
160                 g.fillOval(c*colWidth-(colWidth/4)+colWidth,r*
161                     rowHeight-(rowHeight/4)+rowHeight,colWidth/2,
162                     rowHeight/2);
163                 g.setColor(Color.black);
164             }
165     }
166     // update statuspanels
167     int player = BoardM.getPlayer();
168     if(player != defines.EMPTY)
169     {
170         panel.setPlayer((player == defines.BLACK) ? "Black is up"
171             : "White is up");
172     }
173     else
174         panel.setPlayer("");
175     int [] score = BoardM.getScore();
176     panel.setScore(score[0],score[1]);
177     panel.setStatus(BoardM.getStatus());
178 }
179
180 /**
181  * BoardListener 1.0, 2004/01/14
182  *
183  * This class is responsible for handling mouse input.
184  *
185  * ©Joakim Nyg&aring;rd and Jacob Oettinger
186  */
187 class BoardListener implements MouseListener
188 {
189     public void mouseClicked(MouseEvent me)
190     {
191         int xClick = me.getX();
192         int yClick = me.getY();
193         int width = getWidth();
194         int height = getHeight();
195
196         // calculate the clickable area around each gridpoint on
197         the board

```

```

197     int colWidth    = width/(gameCols+1);
198     int rowHeight   = height/(gameRows+1);
199     int colClick    = (xClick+(colWidth/2))/colWidth-1;
200     int rowClick    =(yClick+(rowHeight/2))/rowHeight-1;
201
202     // if user clicked inside the board, handle it
203     if( colClick >=0 && colClick < gameCols && rowClick
        >=0 && rowClick < gameRows )
204     {
205         BoardM.manipStone(rowClick,colClick);
206         repaint();
207     }
208     //System.out.println("x,y: "+xClick+", "+yClick+";"+
        colClick+", "+rowClick);
209 }
210
211 public void mousePressed(MouseEvent me)
212 {
213     // nothing
214 }
215
216 public void mouseReleased(MouseEvent me)
217 {
218     // nothing
219 }
220 public void mouseExited(MouseEvent me)
221 {
222     // nothing
223 }
224 public void mouseEntered(MouseEvent me)
225 {
226     // nothing
227 }
228 }
229 }

```

## D.5 Const

```

1 public class Const extends Object
2 {
3     // Constants for the differernt states possible on the
        board
4     public static final int EMPTY = 0;
5     public static final int BLACK = 1;
6     public static final int WHITE = 2;
7     public static final int SELECTED = 3;
8 }

```

## D.6 Fanorona

```

1 public class Fanorona
2 {
3     public static void main(String args[])
4     {

```

```

5         // Start the game in a frame with some settings
6         FanoronaFrame frame = new FanoronaFrame();
7         frame.setTitle("Fanorona");
8         frame.setSize(600,422);
9         frame.setDefaultCloseOperation(frame.EXIT_ON_CLOSE);
10        frame.setLocation(200,200);
11        frame.setVisible(true);
12    }
13 }

```

## D.7 FanoronaTest

```

1
2 /**
3  * FanoronaTest 1.0, 2004/01/19
4  *
5  * A class to run a structural test of the method isLegalMove()
6  * from FanoronaModel
7  * in accordance with the notes Systematic Software Test by
8  * Peter Sestoft.
9  *
10 * ©Joakim Nyg&aring;rd and Jacob Oettinger
11 */
12 class FanoronaTest
13 {
14     static int noOfRows = 5;
15     static int noOfCols = 9;
16
17     public static void main(String args[])
18     {
19         boolean didGood = true;
20
21         System.out.println("Starting structural test of method
22         isLegalMove()...\n");
23         System.out.println("-----");
24
25         didGood = didGood && doTest(0,0,-1,0,false); // outside
26         board
27         didGood = didGood && doTest(0,0,0,-1,false); // outside
28         board
29         didGood = didGood && doTest(0,0,0,0,false); // moving
30         nowhere
31         didGood = didGood && doTest(4,8,4,9,false); // outside
32         board
33         didGood = didGood && doTest(4,8,5,9,false); // outside
34         board
35         didGood = didGood && doTest(0,0,1,1,true); // legal
36         diagonal move
37         didGood = didGood && doTest(0,0,0,1,true); // legal move
38         left
39         didGood = didGood && doTest(0,0,1,0,true); // legal move
40         down

```

```

31     didGood = didGood && doTest(0,1,1,2,false);    // illegal
        diagonoal move
32     didGood = didGood && doTest(0,0,2,0,false);    // moving too
        far
33     didGood = didGood && doTest(0,0,0,2,false);    // moving too
        far
34     didGood = didGood && doTest(0,0,2,2,false);    // too far
        diagonal
35
36     System.out.println("-----");
        ");
37
38     if(didGood)
39         System.out.println("The test succeeded! The method
        isLegalMove() is working.");
40     else
41         System.out.println("What? How did you get here?");
42 }
43
44
45 public static boolean doTest(int fromRo,int fromCo,int toRo,
        int toCo,boolean expected)
46 {
47     if(expected == isLegalMove(fromRo,fromCo,toRo,toCo))
48     {
49         System.out.println("Move from ("+fromRo+", "+fromCo+") to
        ("+toRo+", "+toCo+") as expected: "+expected);
50         return true;
51     }
52     System.out.println("Move from ("+fromRo+", "+fromCo+") to ("
        +toRo+", "+toCo+") gave wrong result: "+(!expected));
53     return false;
54 }
55
56
57 /**
58  * Checks wether user can move a stone from fromRo,fromCo to
        toRo,toCo.
59  *
60  */
61 private static boolean isLegalMove(int fromRo,int fromCo,int
        toRo,int toCo)
62 {
63     int rowDist = Math.abs(fromRo-toRo);
64     int colDist = Math.abs(fromCo-toCo);
65
66     if(!isOnBoard(toRo,toCo)) // outside board
67         return false;
68     else if(rowDist == 1 && colDist == 1) // diagonal move
69         return((fromRo%2==0 && fromCo%2==0) || (fromRo%2!=0 &&
        fromCo%2!=0));
70     else // straight line, return true if distance is 1
71         return((rowDist == 1 && colDist == 0) || (colDist
        == 1 && rowDist == 0));

```

```
72 }
73
74
75 /**
76  * Returns true if the given coordinates are on the board
77  */
78 private static boolean isOnBoard(int row, int col)
79 {
80     return((row>=0 && row<noOfRows) && (col>=0 && col<noOfCols)
81           );
82 }
83 }
```