# Semesteroppgave: testing og dokumentering

DATS1600 oppgavesett 4.

I dette oppgavesettet skal dere implementere spillereglene for GoL. Implementering av disse spillereglene kan klassifiseres som middels vanskelighetsgrad. Implementasjonen burde her nøye testes før dere går til neste oppgavesett, siden det er lett å produsere bugs i koden. Før dere starter med dette oppgavesettet, påse at dere forstår spillereglene til GoL.

## Enkel testing

Testing presenteres i forelesning (uke 6). Hvis du føler deg ukomfortabel med programmering, burde du gå til tutorialtimene fra uke 10 til og med 13. Implementasjon av logikk kan fort føre til kode av lav kvalitet og lesbarhet. Gå til tutorialtimene i uke 5, 6, og 7 for presentasjoner relatert til dette emnet.

I denne oppgaven antar vi at spillebrettet er minst 3x3 stort. For å implementere nextGeneration() metoden korrekt, må to oppgaver løses for hver celle:

1. Tell antall levende naboceller.
2. Bestem om cellen skal være død eller levende.

Lag en privat metode i klassen til nextGeneration som teller antall levende naboceller til en gitt celle. Først, tenk på hvordan dere koder antall levende naboceller til hver celle. Ut ifra dette, bestem signaturen til metoden. Eksempel på signatur:

```
private int countNeighbours(int x, int y) { … }
```

I første omgang, kan dere anta at cellen ikke ligger på kanten av spillebrettet. Det vil si at cellen alltid vil ha nøyaktig åtte naboceller.

Etter at dere har implementert metoden, test den med cellen (1,1). Skriv ut antallet til konsollen og kontroller at antallet er korrekt. Gjør endringer på spillebrettet (manuelt i koden) og test forskjellige konfigurasjoner til nabocellene. Vær 100 % sikre på at implementasjonen er korrekt før dere går til neste steg. Tips: det er åtte forskjellige måter å telle én levende celle. Test alle disse åtte konfigurasjonene (dvs. test om alle cellene er med i tellingen). Gjør liknende systematisk testing for å være sikre på at implementasjonen er korrekt.

Diskuter hvordan dere skal håndtere kanten til spillebrettet. Gjør endringer i countNeighbours metoden hvis nødvendig og test celler som er relatert til kanter og hjørner i brettet.

## Unit testing (enhetstesting) og JUnit

Metoden nextGeneration() er en offentlig metode relatert til komplisert logikk og burde testes via enhetstesting (JUnit i Java). Først, les om hvordan JUnit er integrert i IDEen dere bruker. Eventuelt prøv JUnit i et separat Java testprosjekt (som ikke er relatert til GoL).

Opprett følgende metoder, som kan brukes i testingen:

- public String toString() – (overkjørelse av toString metoden i Object). Returner tilstanden for hver celle, der brettet er formatert som en 1D tabell (row major). Hvis cellen er i live, skal ‘1’

skrives og hvis cellen er død, skal '0' skrives. Returnert String fra eksempelet i foregående oppgavesett: 1001011001101001

- public void setBoard(<type_til_brett> gameBoard) – sett 2D brettet direkte. **Merk**: denne metoden er nødvendig for å kunne teste flere konfigurasjoner av brettet. Metoden kan fjernes etter at innlasting fra fil er implementert (sett 5). Hvis dere gjør dette, må dere endre på enhetstestene dere lager her (noe som er OK).

Opprett en JUnit test klasse for å teste klassen relatert til GoL, helst i en egen pakke (package) for testing. Denne testklassen skal tilby følgende metode: @Test public void testNextGeneration() { … }

Bruk metodene setBoard(…) og toString() for å teste nextGeneration() metoden. Det vil si, opprett en rekke brett og manuelt kalkuler hva neste generasjon skal være for hvert brett. Metoden toString kan brukes til å sammenlikne de to brettene. Eksempel på en enkel test:

```
@Test
public void testNextGeneration() {
	byte[][] board = {
			{ 0, 0, 0, 0 },
			{ 0, 1, 1, 0 },
			{ 0, 1, 1, 0 },
			{ 0, 0, 0, 0 }
	};
	GameOfLife gol = new GameOfLife();
	gol.setBoard(board);
	gol.nextGeneration();
	org.junit.Assert.assertEquals(gol.toString(),"0000011001100000");
}
```

Denne metoden kan utvides for å teste en rekke konfigurasjoner av spillebrettet. Oppsettet av testNextGeneration burde være slik at når ingen tester i metoden feiler, så er nextGeneration metoden korrekt implementert. Husk å både teste positive og negative resultater!

## Dokumentering av kode via Javadoc

Alle klasser og interfacer som skrives for semesteroppgaven skal dokumenteres ved hjelp av Javadoc. Sett dere inn i Javadoc. En bra ressurs: http://www.oracle.com/technetwork/articles/java/index-137868.html.

Dokumenter de klassene dere har laget hittil. Bruk IDEen deres til å produsere de korresponderende HTML dokumentene. Det er vanlig å skrive Javadoc på engelsk.

I tillegg til bruk av Javadoc kommentarer, skal dere også bruke generelle Java kommentarer for å beskrive koden deres, spesielt der programlogikken ikke er innlysende. Dette er spesielt viktig når dere arbeider i gruppe, der alle gruppemedlemmene skal kunne forstå (og bygge på) nåværende programkode. Kommenter alle metoder som inneholder flere enn fem instruksjoner. Strukturer disse metodene slik at koden er av høy lesbarhet.

Når dere er ferdige med hver oppgave i arbeidet med semesteroppgaven, er det viktig at dere dokumenterer de klassene og metodene dere har implementert. Dette blir spesielt viktig for klasser dere implementerer individuelt. Kort sagt, burde medlemmer i gruppen ikke lese Java kode for å sette seg inn i klasser/metoder andre har skrevet. Javadoc dokumentasjonen og/eller kommentarer burde være nok til å få en generell oversikt over det som har blitt implementert hittil i prosjektet.

## Møte med kontaktperson

Etter at dere har utført oppgavene over, skal dere møte kontaktpersonen deres for å diskutere deres progresjon og utføring så langt i prosjektet. Det er deres ansvar å kontakte deres kontaktperson for å avtale møte og møte er obligatorisk (en del av obligatorisk oppgave 2).

Absolutt siste frist for møte: uke 14.

Dere skal diskutere følgende på møtet:

- Samarbeidet i gruppen og bidraget for hvert medlem. Er det nødvendig å gjøre endringer i samarbeidet? I ekstreme tilfeller, der for eksempel et medlem ikke har bidratt, så kan det være aktuelt å splitte opp gruppen.
- Deres nåværende kodestruktur. Hvilke type objekter bruker dere og hva er sammenhengen mellom objektene? Er det aktuelt å endre på kodestrukturen?
- Deres kode. Er koden godt dokumentert og er den lesbar?

Hvis du ønsker å "advare" kontaktperson før møtet om noe du føler ikke går som det skal, så send en epost til kontaktpersonen. Hvis problemet er kritisk, så vil fagansvarlig kontaktes av kontaktperson.

## Utvidelsesoppgaver

NB: det anbefales å implementere kjernen til programmet (sett 5) før dere starter med utvidelsesoppgaver.

### Ytelsesforbedringer

Etter hvert vil vi se på større spillebrett med flere tusen celler. Nåværende implementasjon kan være upassende for slike brett, siden hver celle sjekkes for levende naboer.

En mulig optimalisering er å bruke følgende faktum: hvis en celle ikke endres, og ingen av de åtte nabocellene endres, kan ikke cellen endres ved neste generasjon.

I tillegg til spillebrettet, som håndterer hvilke celler som er levende eller døde, opprett en liknende 2D tabell med samme størrelse som spillebrettet. Denne tabellen skal håndtere hvilke celler som er inaktive, der man ikke kaller countNeighbours metoden hvis en celle er inaktiv. Inkorporer oppdatering av denne tabellen i nextGeneration metoden.

Videre optimaliseringer kan gjøres. En typisk plaskehals er overføringer av data fra hovedminne til grafikkminne (GPU) for tegning av grafikk. Dette må gjøres mellom hvert bilde hvis dataene til brettet ligger på hovedminne og er spesielt kostbart fordi overføringen foregår på den relativt trege PCI express bus'en. En bedre løsning er å bevare representasjonen av brettet på minne til GPU brikken. Logikken for oppdatering av brettet må derfor utføres på GPU brikken, noe som innebærer at du må programmere på GPUen.

Implementasjon av spillereglene på GPUen er komplisert fordi du trenger å hente informasjon om andre datapunkter fra hvert enkelt datapunkt (du må telle levende naboer, noe som krever at nåværende celle aksesserer naboceller). Et valg er å bruke GPU 'shadere', som er små programmer som kan kjøres på ulike typer dataer ettersom de transformeres fra celler på et spillebrett til pixler. Dette er imidlertid et lite attraktivt valg fordi du ikke enkelt kan hente informasjon fra andre celler enn din nåværende celle (du kan trikse med teksturer, men dette blir fort avansert). Et bedre valg er å utføre oppgaven med NVIDIA sitt C-liknende bibliotek CUDA (de bruker et eget-definert språk som likner C). Du kan bruke wrapper biblioteket jcuda for integrasjon med Java og du trenger en NVIDIA GPU som er kompatibel med CUDA på din datamaskin.

Informasjon: http://jcuda.org/ (se eksempler som utfører operasjoner på 2D matriser).

For å visualisere spillebrettet er anbefalt fremgangsmåte å opprette et bilde av brettet på GPUen (en tekstur) og tegne dette bilde med OpenGL (via JWJGL); se informasjon relatert til JWJGL i utvidelsesoppgaven i oppgavesett 2.

Merk: hvis du frykter programmeringsspråket C, så er dette ikke en oppgave for deg (GPU programmering kan ikke gjøres direkte i Java).

## Manipulering av spilleregler

Conway sine regler er ikke nødvendigvis 'satt i sten'. En myriade av alternative regler eksisterer:

https://en.wikipedia.org/wiki/Life-like_cellular_automaton

Gjør endringer i kodestrukturen slik at generelle regler kan representeres. En endring burde være dynamisk: man skal kunne endre på reglene til spillet uten å endre eller fjerne det nåværende spillebrettet. Regler skal kunne manipuleres direkte av brukeren fra GUI og spillet skal oppdateres deretter (igjen, uten å slette nåværende spillebrett). I tillegg til den kryptiske RLE notasjonen (fra Wiki artikkelen), skal en mer 'vennlig' presentasjon av ulike regler presenteres. Du må selv vurdere hvordan forskjellige regler best presenteres til brukeren.