

Eksamensoppgave i**TDT4100 – Objektorientert programmering****Tirsdag 2. juni 2009, kl. 09:00 - 13:00**

*Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikrer Trond Aalberg.
Kontaktperson under eksamen er Hallvard Trætteberg (mobil 918 97263)*

Språkform: Bokmål

Tillatte hjelpemidler: C

Én valgfri lærebok Java er tillatt. Trykt kompendium er ikke tillatt.

Sensurfrist: Tirsdag 23. juni.

Les oppgaveteksten nøye. Finn ut hva det spørres etter i hver oppgave.

Dersom du mener at opplysninger mangler i en oppgaveformulering gjør kort rede for de antagelser og forutsetninger som du finner det nødvendig å gjøre.

Vær oppmerksom på fordelingen av prosentpoeng mellom oppgavene, så du disponerer tiden riktig.

1. Småplukk og testing (15%)

a) Skriv en metode `removeVowels`, som tar inn et `String`-parameter og returnerer en `String`, hvor alle *vokalene* fra parameteret er fjernet. F.eks. skal `removeVowels("Java")` returnere "Jv". Dersom parameteret ikke inneholder noen vokaler så skal parameteret returneres og ikke en ny/annen `String`.

b) Beskriv den *generelle* testeteknikken som JUnit-testing (og JExercise) baserer seg på. Skriv en testmetode som tester om `removeVowels` tilfredsstillter kravene som er beskrevet over. Testmetoden trenger ikke være skrevet iht. JUnit-rammeverket sine regler og konvensjoner.

c) Skriv en metode `findSequence`, som tar inn en `char`-tabell med 'x' er og 'o' er og en posisjon (`int`). Anta at tegnet på den angitte posisjonen er en 'x'. Da skal metoden returnere antall 'o' er som kommer etter, før en ny 'x'. Dersom det ikke er noen 'x' før enden på tabellen, skal -1 returneres.

Dersom tegnet på den angitte posisjonen er en 'o', så er logikken den samme, men med rollene til 'x' og 'o' byttet om. Gitt at charArray er {'x', 'x', 'o', 'o', 'x', 'o'}, så skal
findSequence(charArray, 0) returnere 0,
findSequence(charArray, 1) returnere 2,
findSequence(charArray, 2) returnere 0,
findSequence(charArray, 3) returnere 1,
findSequence(charArray, 4) returnere -1 og
findSequence(charArray, 5) returnere -1.

d) Gitt følgende Iterator-implementasjon og testmetode:

```
public class Illegal123Iterator implements Iterator<Integer> {
    private int next = 0;
    public boolean hasNext() {
        if (next >= 3) {
            return false;
        }
        next++;
        return true;
    }
    public Integer next() {
        return next;
    }
}
```

```
public void testIllegal123Iterator() {
    Iterator<Integer> iterator = new Illegal123Iterator();
    assertTrue(iterator.hasNext());
    assertEquals(1, iterator.next().intValue());
    assertTrue(iterator.hasNext());
    assertEquals(2, iterator.next().intValue());
    assertTrue(iterator.hasNext());
    assertEquals(3, iterator.next().intValue());
    assertFalse(iterator.hasNext());
}
```

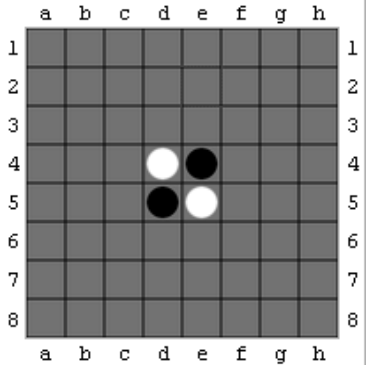
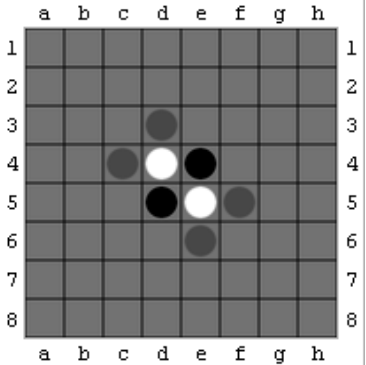
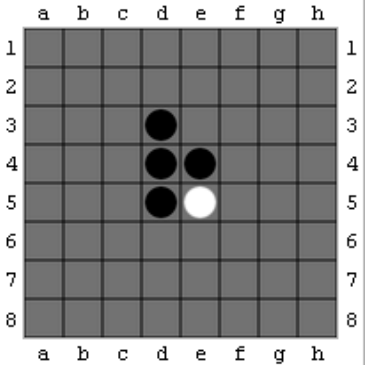
assertTrue-metoden sjekker om argumentet er true, mens assertEquals-metoden sjekker om de to argumentene er like.

Klassen Illegal123Iterator er ment å generere tallene 1, 2, 3 og testmetoden testIllegal123Iterator er ment å teste dette. Ved første øyekast virker begge riktige, siden testmetoden tester at sekvensen som returneres er 1, 2, 3 og ikke gir noen feil ved kjøring. Imidlertid inneholder både klassen og testmetoden logiske feil.

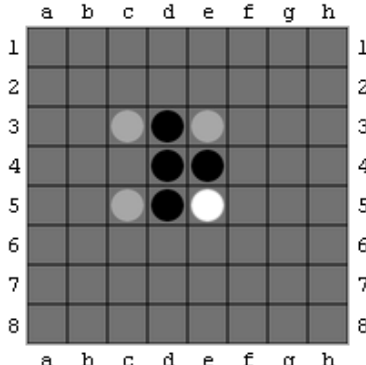
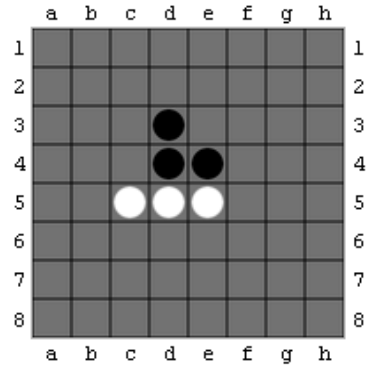
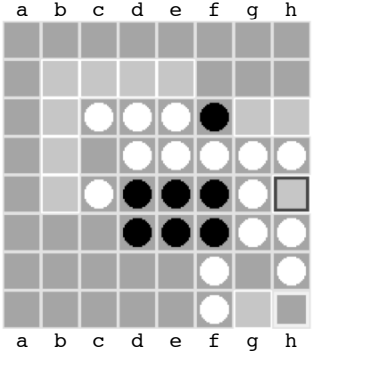
Vis først hvordan testmetoden burde vært skrevet for å finne feilen i Illegal123Iterator-klassen. Vis deretter hvordan Illegal123Iterator-klassen burde vært skrevet for å tilfredsstille kravene til en Iterator-implementasjon.

2. Othello/Reversi (40%)

Othello-spillet, også kalt Reversi, spilles av to spillere på et spillebrett med 8x8 ruter. Hver spiller har hvert sitt sett med brikker, hhv. sorte og hvite. Spillet starter med 4 brikker lagt på brettet slik figuren under til venstre viser (bilder tatt fra <http://en.wikipedia.org/wiki/Reversi>):

		
Startposisjon, sort sin tur	4 mulige plasseringer	Sort brikke er plassert på d3

Spillerne veksler på å plassere én brikke om gangen på brettet, og sort starter. En brikke må plasseres slik at minst én sekvens av motstanderbrikke fanges mellom egne brikker. Sekvenser kan være vertikale, horisontale eller diagonale. I figuren over i midten er fire mulige plasseringer av en sort brikke angitt. I alle tilfellene vil én sekvens av én hvit brikke bli fanget. Når en brikke er plassert, erstattes motstanderbrikkene som er fanget, med brikker av egen farge. Dette er vist i figuren til høyre, hvor en sort brikke er plassert på d3 og den hvite brikken på d4 er fanget og erstattet med en sort en. Det er nå hvit sin tur, og det er tre mulige plasseringer, som vist i figuren under til venstre. Dersom hvit velger alternativet c5, fanges den sorte brikken på d5 og blir erstattet med en hvit, slik figuren under i midten viser.

		
Hvits tur, 3 muligheter	Hvit brikke er plassert på c5	Brikker kan fanges i mer enn én retning.

Merk at brikker kan fanges i mer enn én retning. Dette er illustrert i figuren over til høyre, hvor en sort brikke plassert på h5 vil fange de hvite brikkene på g5 og g4. De andre mulighetene er her angitt med lysere farge. Det er også mulig å fange mer enn én brikke i hver retning (dette er ikke vist her).

Dersom en spiller ikke har noen lovlige plasseringer, så går turen videre til den andre spilleren. Spillet fortsetter inntil ingen av spillerne har noen lovlige plasseringer, f.eks. fordi brettet er fullt eller alle brikkene har samme farge. Spilleren med flest brikker på brettet vinner.

a) Forklar hvordan du vil representere Brett og (plasserte) brikker med Java-klasser og hvordan du vil innkapsle tilstanden med metoder.

b) Gitt følgende enum-klasse og klassene du har beskrevet:

```
public enum Direction {
    NORTH(0, -1), NORTHEAST(1, -1), EAST(1, 0), SOUTHEAST(1, 1), SOUTH(0, 1),
    SOUTHWEST(-1, 1), WEST(-1, 0), NORTHWEST(-1, -1);
    public final int dx, dy;
    private Direction(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }
}
```

Deklarer og implementer metoder i brettklassen din for å:

- telle hvor mange brikker som evt. fanges i en bestemt retning, dersom en bestemt brikke plasseres på en bestemt rute
- sjekke om en gitt brikke er lovlig å plassere på en bestemt rute
- plassere en gitt brikke på en bestemt rute (gitt at det er lov) og erstatte fangede brikker med brikker med motsatt farge

Definer gjerne ekstra metoder for å gjøre implementasjonen enklere å skrive og forstå. Dersom en metode er vanskelig å skrive, så deklarer den og forklar hva den er ment å gjøre.

c) Beskriv med tekst og kode hvordan du vil implementere et spill som kommuniserer med spillerne vha. `System.in` og `System.out`, og som styrer spillet iht. reglene angitt over. Eksempel på interaksjon med brukeren, hvor øverst rad tilsvare `System.out` og nederste `System.in`:

<pre>abcdefgh 1.....1 2.....2 3...+...3 4..+ox...4 5...xo+..5 6....+...6 7.....7 8.....8 abcdefgh It is BLACK's turn</pre>	<pre>abcdefgh 1.....1 2.....2 3..+x+...3 4...xx...4 5..+xo...5 6.....6 7.....7 8.....8 abcdefgh It is WHITE's turn</pre>	<pre>abcdefgh 1.....1 2.....2 3...x...3 4...xx...4 5..ooo...5 6.+++++.6 7.....7 8.....8 abcdefgh It is BLACK's turn</pre>
d3	c5	

Bruk metodene definert tidligere i oppgaven, også om du ikke har implementert dem.

Besvarelsen blir vurdert etter hvor logisk og strukturert løsningen er, om innkapsling er brukt, hvorvidt Java er godt og riktig brukt og om koden er logisk riktig.

3. Highscore-liste (35%)

En highscore-liste brukes for å rangere ulike spilleres resultater. For eksempel kan en ha en highscore-liste med antall flytt/dytt som en har brukt på et bestemt Sokoban-nivå eller tiden en har brukt på en bestemt Sudoku. Merk at det ikke bør tillates å ha ulike typer spill i samme highscore-liste, hvis det ikke gir mening å sammenligne resultatene.

Du skal implementere en slik liste og vise hvordan den skal brukes for et bestemt spill. Du står fritt til å definere klasser og metoder som du synes er nødvendige. Vi ønsker en generell løsning, slik at mange typer spill kan håndteres. Begrens evt. problemet og løsningen for å gjøre det enklere og mindre tidkrevende, heller enn å lage en halvveis løsning.

Skisse til løsning, som du kan velge å jobbe videre med:

- Vi definerer et grensesnitt (interface) kalt `GameResult` som arver fra `Comparable`.
- Vi definerer en abstrakt klasse kalt `SinglePlayerGameResult`, som implementerer `GameResult` og med felt og metoder for å håndtere navnet på en spiller som har spilt et spill. Tilsvarende klasser kan lages for tur-baserte spill med flere spillere.
- For et bestemt spill, f.eks. et Sokoban-nivå eller Sudoku, lager en så en implementasjon med felt for å lagre informasjon om spillet og spillresultatet. F.eks. vil en `SokobanGameResult`-klasse arve fra `SinglePlayerGameResult` og inneholde informasjon om brettet og antall flytt/dytt som ble brukt. I tillegg vil klassen implementere `Comparable`-logikken og `toString()`-metoden iht. hvordan spillresultatet er representert.
- `Highscores`-klassen inneholder innkapslingsmetoder for legge til og fjerne resultater, og skrive ut en sortert liste med resultater.

Besvarelsen blir vurdert etter hvor logisk og strukturert løsningen er, om innkapsling er brukt, hvorvidt Java er godt og riktig brukt og om koden er logisk riktig.

4. Observatør-observert-teknikken (10%)

- a) Forklar den generelle virkemåten til observatør-observert-teknikken.
- b) Beskriv med tekst og kode hvordan du vil gjøre din highscore-liste-implementasjon over observerbar vha. denne teknikken.