NTNU
Institutt for datateknikk
og informasjonsvitenskap

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`.

Her var det meningen å vise enkel bruk av iterasjon og String, evt. StringBuilder. En kan også bruke replace med regexp, og det er vi nødt til å tillate, selv om det er en snarvei. Det trekkes ikke mye om en ikke får returnert parameteret, dersom det ikke inneholder noen vokaler. Her er noen løsningsvarianter:

```
public static String removeVowels1(String s) {
      String result = null;
      for (int i = 0; i < s.length(); i++) {
```

```
            char c = s.charAt(i);
            if ("AEIOUYÆØÅaeiouyæøå".indexOf(c) >= 0) {
                  if (result == null) {
                        result = s.substring(0, i);
                  }
            } else if (result != null) {
                  result += c;
            }
      }
      return (result != null ? result : s);
}

public static String removeVowels2(String s) {
      StringBuilder builder = new StringBuilder();
      for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if ("AEIOUYÆØÅaeiouyæøå".indexOf(c) < 0) {
                  builder.append(c);
            }
      }
      return (builder.length() == s.length() ? s : builder.toString());
}

public static String removeVowels3(String s) {
      String vowels = " AEIOUYÆØÅaeiouyæøå";
      for (int i = 0; i < vowels.length(); i++) {
            String c = String.valueOf(vowels.charAt(i));
            s = s.replace(c, "");
      }
      return s;
}

public static String removeVowels4(String s) {
      return s.replaceAll("[AEIOUYÆØÅaeiouyæøå]", "");
}

public static String removeVowels5(String s) {
      for (int i = 0; i < s.length();) {
            char c = s.charAt(i);
            if ("AEIOUYÆØÅaeiouyæøå ".indexOf(c) >= 0) {
                  s = s.substring(0, i) + s.substring(i + 1);
            } else {
                  i++;
            }
      }
      return s;
}

public static String removeVowels6(String s) {
      for (int i = s.length() - 1; i >= 0; i--) {
            char c = s.charAt(i);
            if ("AEIOUYÆØÅaeiouyæøå".indexOf(c) >= 0) {
```

```
                        s = s.substring(0, i) + s.substring(i + 1);
                    }
                }
                return s;
        }
}
```

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

Den generelle teknikken er å rigge opp objekter med en før-tilstand, endre tilstanden og sammenligne med forventet etter-tilstand. Noen ganger er det ingen før- og etter-tilstand, da en kun sjekker et selvstendig metodekall, dvs. sammenligner returverdi med fasiten.

Her brukes teknikken for å sjekke removeVowels:

```
        public void testRemoveVowels1() {
                assertEquals("Jv", removeVowels("Java"));
                assertEquals("bjktrntrt prgrmmrng", removeVowels("Objektorientert
programmering"));
                String angstskrik = "ngstskrk";
                assertTrue(angstskrik == removeVowels(angstskrik));
                String emptyString = "";
                assertTrue(emptyString == removeVowels(emptyString));
        }
```

Merk bruken av assertTrue og == og ikke assertEquals for å sjekke for identisk likhet. Det er ikke påkrevd å bruke disse to metodene, men effekten må være den samme.

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 sammen, 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`.

Det avgjørende er å ikke starte på pos, men pos + 1, og returnere -1 dersom en kommer til enden før tegnet på pos er funnet på nytt. Merk at dersom en antar at tabellen er gyldig, trenger en ikke nevn verken 'x' eller 'o' i koden.

Her er tre varianter:

```
        public static int findSequence1(char[] charArray, int pos) {
                char c = charArray[pos];
                pos++;
```

```
            for (int count = 0; pos < charArray.length; pos++, count++) {
                  if (charArray[pos] == c) {
                        return count;
                  }
            }
            return -1;
      }

      public static int findSequence2(char[] charArray, int pos) {
            char c = charArray[pos];
            pos++;
            int count = 0;
            while (pos < charArray.length) {
                  if (charArray[pos] == c) {
                        return count;
                  }
                  pos++;
                  count++;
            }
            return -1;
      }

      public static int findSequence3(char[] charArray, int pos) {
            for (int count = 0; pos + count + 1 < charArray.length; count++) {
                  if (charArray[pos + count + 1] == charArray[pos]) {
                        return count;
                  }
            }
            return -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.

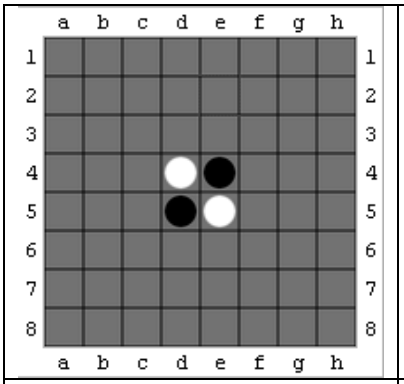
Poenget her er at hasNext()-metoden ikke skal endre på iterator-tilstanden, siden den skal kunne kalles flere ganger etter hverandre. Det er kun next()-metoden som skal ta et steg fremover. Imidlertid så testes ikke dette poenget. Løsninger altså 1) å legge to (eller flere) kall til hasNext() mellom hvert kall til next(), og 2) å flytte bruken av next++ fra hasNext() til next():

```
	public void testCorrect123Iterator() {
		Iterator<Integer> iterator = new Correct123Iterator();
		assertTrue(iterator.hasNext());
		assertTrue(iterator.hasNext());
		assertEquals(1, iterator.next().intValue());
		assertTrue(iterator.hasNext());
		assertTrue(iterator.hasNext());
		assertEquals(2, iterator.next().intValue());
		assertTrue(iterator.hasNext());
		assertTrue(iterator.hasNext());
		assertEquals(3, iterator.next().intValue());
		assertFalse(iterator.hasNext());
		assertFalse(iterator.hasNext());
	}


public class Correct123Iterator implements Iterator<Integer> {

	private int next = 1;

	public boolean hasNext() {
		return (next <= 3);
	}
	public Integer next() {
		int result = next;
		next++;
		return result;
		// same effect as return next++;
	}
	public void remove() {
```

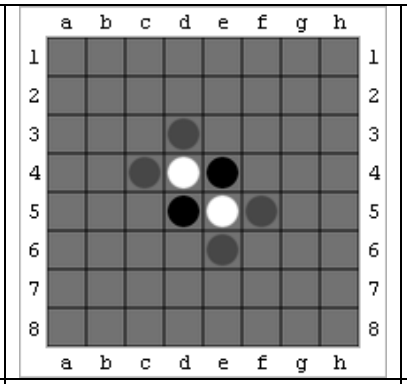
```
        }
}
```

2. Othello/Reversi (40%)

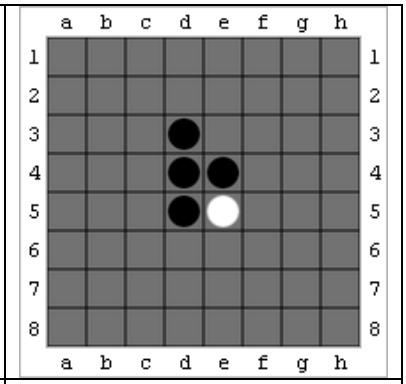
Othello-spillet, også kalt Reversi, spilles av to spillere på et spillebrett med 8x8 ruter. Hver spiller har hvert sitt sett med brikker, hhv. sorte og hvite. Spillet starter med 4 brikker lagt på brettet slik figuren under til venstre viser (bilder tatt fra http://en.wikipedia.org/wiki/Reversi):



Startposisjon, sort sin tur

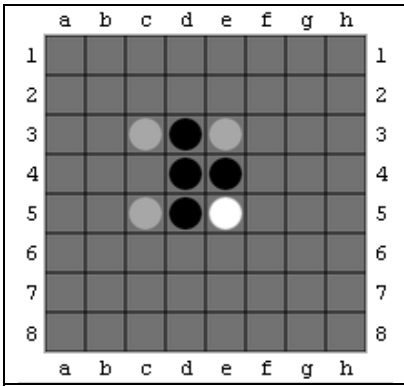


4 mulige plasseringer

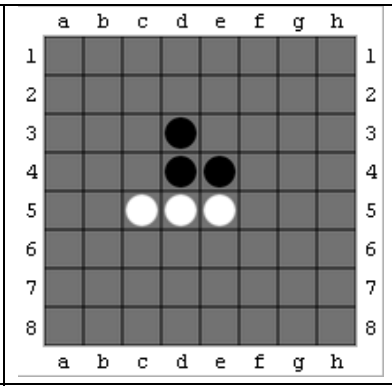


Sort brikke er plassert på d3

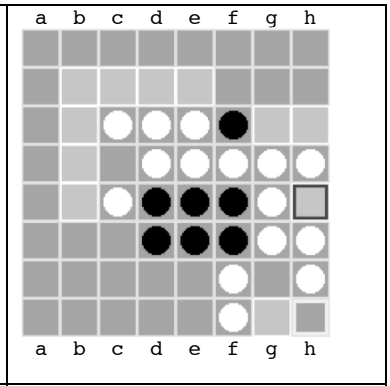
Spillerne veksler på å plassere én brikke om gangen på brettet, og sort starter. En brikke må plasseres slik at minst én sekvens av motstanderbrikke fanges mellom egne brikker. Sekvenser kan være vertikale, horisontale eller diagonale. I figuren over i midten er fire mulige plasseringer av en sort brikke 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.

Det naturlige er å bruke en to-dimensjonal tabell til å representere brettet og en enum til å representere brikkene (null er tom). En kan evt. bruke en annen type verdi for brikken, så lenge en kan representere tre ulike tilstander tom, sort og hvit, så int, char og Boolean duger (men ikke boolean). Brikken selv trenger ingen innkapsling, men brettet gjør, så det er naturlig å lage en Brett-klasse med et privat brett-felt av typen Piece[][]. Piece er her brikke-typen. En trenger innkapslingsmetoder for å lese og sette en rute, i tillegg til korrekt initialisering til en tabell med dimensjonene 8x8 i konstruktøren:

En annen løsning er definere en egen brikketype, som har en char, enum eller int som felt. Dette støtter endring av fargen, tilsvarende å snu brikken, istedenfor å erstatte den. I praksis går det for det samme.

To løsningsvarianter som ikke er gode, men som heller ikke gir trekk (dersom de fungerer):
- Mange definerte en brikkeklasse som inneholdt posisjonen til brikken, gjerne samtidig som at brettet var definert som char[][]. Posisjonen i brikken ble i praksis ikke brukt.
- Mange brukte String-baserte posisjoner i innkapslingen, istedenfor x,y, antageligvis pga. måten dette ble brukt på i sjakkoppgaven året før. Dette er noe som bør håndteres ifm. brukerinput, ikke i innkapslingen til brettet.
Selv om man ikke får trekk for slikt, så sluker det tid på eksamen, og gjør koden unødvendig komplisert.

```
public class Board {

	public static final int BOARD_WIDTH = 8;
	public static final int BOARD_HEIGHT = 8;

	private Piece[][] board;

	public Board() {
		board = new Piece[BOARD_HEIGHT][BOARD_WIDTH];
		board[3][3] = Piece.WHITE;
		board[3][4] = Piece.BLACK;
		board[4][4] = Piece.WHITE;
		board[4][3] = Piece.BLACK;
	}

	public boolean isLegalPosition(int x, int y) {
		return x >= 0 && x < BOARD_WIDTH && y >= 0 && y < BOARD_HEIGHT;
	}

	public Piece getPiece(int x, int y) {
		return isLegalPosition(x, y) ? board[y][x] : null;
	}

	public void setPiece(int x, int y, Piece piece) {
		if (isLegalPosition(x, y)) {
			board[y][x] = piece;
		}
```

```
	}
}

public enum Piece {
	BLACK, WHITE;
	// hjelpemetode som returnerer den andre brikkefargen (ift. this)
	public Piece other() {
		switch (this) {
		case BLACK: return WHITE;
		case WHITE: return BLACK;
		}
		return null;
	}
}
```

Det spørres kun etter en forklaring, så det er et spørsmål hvor mye kode som behøves. Det skal en del til for at en tekstlig forklaring viser at en behersker alle detaljene, så noe kode er nok nødvendig.

Det er greit om en bruker unntak i get/setPiece, når en har ulovlige koordinater. En bør da bruke IllegalArgumentException, i hvert fall en RuntimeException, altså unchecked exception.

Den kan også være naturlig å legge til noen hjelpemetoder for metoden i punkt b), selv om det ikke er rene innkapslingsmetoder.

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.

Her er det vesentlig å definere naturlige parameterlister, f.eks. utnytte Direction-enum'en og brikketypen og kalle eksisterende metoder, der det er nyttig. Følgende hjelpemetode i Board-klassen, for å telle og bytte når det er lov, er nyttig:

```
public int count(Piece piece, int x, int y, Direction direction, boolean
```

```
replace) {
	int count = 0;
	x += direction.dx;
	y += direction.dy;
	while (getPiece(x, y) == piece.other()) {
		count++;
		x += direction.dx;
		y += direction.dy;
	}
	if (getPiece(x, y) != piece) {
		count = 0;
	}
	if (count > 0 && replace) {
		x -= direction.dx;
		y -= direction.dy;
		while (getPiece(x, y) == piece.other()) {
			setPiece(x, y, piece);
			x -= direction.dx;
			y -= direction.dy;
		}
	}
	return count;
}
```

Her kan en vise at en kan bruke Direction-klassen for å lete i en bestemt retning. Dersom metoden ligger utenfor Board-klasse, så må en bruke innkapslingsmetodene. Her har vi gjort det selv om metoden ligger i Board-klassen, såkalt *intern* innkapsling. Noen har klart å bruke findSequence fra oppgave 1 og det er greit.

```
public int countTraps(Piece piece, int x, int y, Direction direction) {
	return countAndReplace(piece, x, y, direction, false);
}

public boolean canTrap(Piece piece, int x, int y) {
	int count = 0;
	for (Direction direction: Direction.values()) {
		count += countAndReplace(piece, x, y, direction, false);
	}
	return count > 0;
}

public void place(Piece piece, int x, int y) {
	setPiece(x, y, piece);
	for (Direction direction: Direction.values()) {
		countAndReplace(piece, x, y, direction, true);
	}
}
```

De tre metodene over er de som etterspørres i oppgaven, og må ha analoge parameter- og returtyper. canTrap summerer antallet som fanges og returnerer true om antallet er > 0. place()-metoden plasserer en brikke på en bestemt posisjon og bytter ut alle som blir fanget. I begge disse får en vist at en kan iterere over alle verdiene i en enum med values()-metoden.

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 tilsvarer `System.out` og nederste `System.in`:

`` 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 ```	``` 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 ```	``` 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 ```
`` 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.

Her er litt av poenget å dele opp problemet i passende metoder og evt. klasser. Det er ikke nødvendig å ha en egen klasse for selve spillet, men det gir ekstra poeng. Denne vil i så fall holde rede på brettet og hvem sin tur det er. Uansett bør en ha metoder for å skrive ut tilstanden, dvs. brettet og hvem sin tur det er, finne ut om spilleren må passe, dvs. ingen posisjoner er lovlige for en gitt spiller, og for å utføre en plassering. I main-metoden brukes så disse metodene.

main-metoden bør gå i løkke og lese input fra brukeren, oversette dette til en posisjon på brettet og utføre plasseringen. Det bør sjekkes for lovlighet, om en må passe og om spillet er ferdig. Spillerne må veksle mellom å ha turen. Det er ikke nødvendig å kopiere input- og output-formatet vist i oppgaven, men brettet og hvem som har turen bør skrives. (I etterkant angrer jeg på at jeg tok med +'ene i utskriften, siden det lurte mange til å ha dem med i brettrepresentasjonen.)

Det er mange detaljer som kan bli feil i en besvarelse, og vi ser mest på struktur og logikk og mindre på kodingsteknikk. Fullstendig utskrift og gode feilmeldinger er ikke viktig. Vi foretrekker kode, men gode forklaringer kan gi nesten full pott.

Her er en mulig Reversi-klasse:

```
private Board board;

private Piece turn;

public Reversi() {
      board = new Board();
      turn = Piece.BLACK;
```

```
}

private void printState(boolean coordinates) {
	if (coordinates) {
		System.out.println(" abcdefgh");
	}
	for (int y = 0; y < Board.BOARD_HEIGHT; y++) {
		if (coordinates) {
			System.out.print(y + 1);
		}
		for (int x = 0; x < Board.BOARD_WIDTH; x++) {
			Piece piece = board.getPiece(x, y);
			System.out.print(piece == null ? (board.canTrap(turn, x, y) ?
'+' : '.') : (piece == Piece.BLACK ? 'x' : (piece == Piece.WHITE ? 'o' : '?')));
		}
		if (coordinates) {
			System.out.print(y + 1);
		}
		System.out.println();
	}
	if (coordinates) {
		System.out.println(" abcdefgh");
	}
	System.out.println("It is " + turn + "'s turn");
}

private void pass() {
	turn = turn.other();
}

private void place(int x, int y) {
	board.place(turn, x, y);
	turn = turn.other();
}

public static void main(String[] args) {
	Reversi reversi = new Reversi();
	Scanner scanner = new Scanner(System.in);
	reversi.printState(true);
	while (scanner.hasNextLine()) {
		String line = scanner.nextLine();
		if (line == null || line.length() == 0) {
			break;
		}
		if (line.length() > 1) {
			char first = line.charAt(0), last = line.charAt(line.length() -
1);
			int x = first - 'a', y = last - '1';
			if (! reversi.board.isLegalPosition(x, y)) {
				System.out.println(first + "," + last + " is not a valid
board location");
			} else if (reversi.board.getPiece(x, y) != null) {
```

```
                        System.out.println("There is already a piece at " + x +
"," + y);
                  } else {
                        if (reversi.board.canTrap(reversi.turn, x, y)) {
                              reversi.place(x, y);
                        } else {
                              System.out.println(x + "," + y + " is not a valid
place to put a " + reversi.turn + " piece");
                        }
                  }
            }
            reversi.printState(true);
            if (! reversi.board.canTrap(reversi.turn)) {
                  System.out.println(reversi.turn + "must pass");
                  reversi.pass();
                  if (! reversi.board.canTrap(reversi.turn)) {
                        System.out.println(reversi.turn + "must also pass");
                        break;
                  }
            }
      }
      int whiteCount = reversi.board.count(Piece.WHITE);
      int blackCount = reversi.board.count(Piece.BLACK);
      System.out.println("There are " + whiteCount + " white piece and " +
blackCount + " black pieces left, hence ");
      if (whiteCount > blackCount) {
            System.out.println("white wins!");
      } else if (blackCount > whiteCount) {
            System.out.println("black wins!");
      } else {
            System.out.println("a draw");
      }
}
```

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.

Selv om vi skisserer en løsning, er det mange mulig løsninger, hvor en i ulik grad får vist hva en kan av Java og kodingsteknikk. I skissen har vi lagt opp til bruk av grensesnitt, abstrakt klasse og vanlige klasser, og en besvarelse bør få med seg mest mulig av dette for å få full pott. Det ligger imidlertid i kortene at en god besvarelse kan avvike mye fra den foreslåtte løsningen. På en eksamen er det ikke tid/anledning til fin-design, så ved sensur handler det mer om å tolke koden og se hvor mye mening den gir, heller enn å sammenligne den med vårt løsningsforslag. Nedenfor utdyper vi løsningen vi skisserte i oppgaveteksten.

GameResult-grensesnittet bør inneholde nyttige metoder som er relevante for funksjonen til Highscores-klassen. To oppgaver er vesentlige: sammenligning for sortering og tekst for utskrift. Førstnevnte er fanget opp ved å arve fra Comparable, og sistenevnte ved å definere to String-metoder.

```
public interface GameResult extends Comparable<GameResult> {
      public String getGameString();
      public String getResultString();
}
```

SinglePlayerGameResult er en abstrakt klasse som er ment å være nyttig som superklasse for resultater for spill med én spiller. Det er derfor naturlig at klassen holder informasjon om spilleren. Konstruktøren er protected, for å markere at det ikke er noe poeng å kalle den fra andre klasser enn

subklasser. Vi har valgt å implementere en forhåpentligvis nyttig compareTo-metode, men merk at denne blir redefinert og ikke brukt av SokobanGameResult-klassen.

```
public abstract class SinglePlayerGameResult implements GameResult {

	private String player;

	protected SinglePlayerGameResult(String player) {
		this.player = player;
	}

	public String getGameString() {
		return "played by " + player;
	}

	// default implementation
	public int compareTo(GameResult o) {
		return getResultString().compareTo(o.getResultString());
	}
}
```

SokobanGameResult-klassen viser at en har skjønt bruken av grensesnitt og abstrakte klasser. En kan godt velge et annet spill(resultat), men klassen må uansett holde all relevant informasjon om spillet som er spilt og hva resultatet ble til bruk ved rangering. I vår implementasjon lagres hhv. en String for Sokoban-nivået og antall flytt og dytt som ble brukt ved løsning. compareTo-metoden fra Comparable og de to GameResult-metodene blir alle implementert.

```
public class SokobanGameResult extends SinglePlayerGameResult {

	private String sokobanLevelString;
	private int moves, pushes;

	public SokobanGameResult(String player, String sokobanLevelString, int
moves, int pushes) {
		super(player);
		this.sokobanLevelString = sokobanLevelString;
		this.moves = moves;
		this.pushes = pushes;
	}

	public String getGameString() {
		return sokobanLevelString + " " + super.getGameString();
	}

	public String getResultString() {
		return moves + " moves/" + pushes + " pushes";
	}

	public int compareTo(GameResult o) {
		if (! (o instanceof SokobanGameResult)) {
			throw new ClassCastException("Cannot compare " + o + " to " +
this);
```

```
            }
            SokobanGameResult other = (SokobanGameResult)o;
            int diff = (moves + pushes) - (other.moves + other.pushes);
            if (diff == 0) {
                  diff = pushes - other.pushes;
            }
            return diff;
      }
}
```

Merk at det ble gitt poeng for løsninger som ble forenklet, f.eks. hadde kun én klasse for ett spesifikt spill.

Highscores-klassen binder det hele sammen. Her viser en hvordan en utnytter GameResult-grensesnittet inkludert compareTo-metoden og hvordan en innkapsler en liste med informasjon.

```
public class Highscores {

      private List<GameResult> gameResults = new ArrayList<GameResult>();

      public void addGameResult(GameResult gameResult) {
            int position = 0;
            while (position < gameResults.size()) {
                  if (gameResult.compareTo(gameResults.get(position)) < 0) {
                        break;
                  }
                  position++;
            }
            gameResults.add(position, gameResult);
      }

      public void removeGameResult(GameResult gameResult) {
            gameResults.remove(gameResult);
      }

      public void print(PrintWriter writer) {
            for (GameResult gameResult: gameResults) {
                  writer.printf("%s for %s\n", gameResult.getResultString(),
gameResult.getGameString());
            }
            writer.flush();
      }
}
```

En kan gjerne bruke et SortedSet, slik at en slipper å finne innsettingsposisjonen selv. Merk at dette kan påvirke implementasjonen i neste oppgave. Feltet for å holde spillresultatene bør deklareres som et grensesnitt, f.eks. List, og settes til en instans av en implementasjon, f.eks. ArrayList, og en bør bruke <GameResult> for å spesialisere lista/settet.

Et alternativ er å sortere ifm. utskrift. Dette gjør add/removeGameResult trivielle, men gjør det vanskeligere å implementere observerbarhet med detalji-nformasjon om endringer.

4. Observatør-observert-teknikken (10%)

a) Forklar den generelle virkemåten til observatør-observert-teknikken.

Observatør-observert-teknikken brukes når en eller flere objekter, her kalt observatører, må holdes konsistent med et annet objekt, her kalt observert, og det observerte objektet sin implementasjon ikke skal være for tett knyttet til observatørenes. Teknikken er basert på at det observerte objektet sier fra til en eller flere observatører om at tilstanden er endret, slik at observatørene kan oppdatere sin tilstand ift. deres regler for konsistens.

Rent kodingsteknisk inneholder en løsning følgende:

- Det defineres et lyttergrensesnitt (interface) med metoder tilsvarende de endringene av det observerte objektet som kan forekomme.
- Det observerte objektet administrerer en liste av lyttere.
- Observatøren implementerer lyttergrensesnittet og legges til som lytter på det observerte objektet.
- Ved hver endring som det skal rapporteres om, kalles tilsvarende metode hos alle lytterne.

Vi krever ikke at det redegjøres for dette i punkt a), men i punkt b) må det vises at dette kan gjøres i praksis.

b) Beskriv med tekst og kode hvordan du vil gjøre din highscore-liste-implementasjon over observerbar vha. denne teknikken.

I tilfellet Highscores-klassen bør en identifisere hva slags endringer det er relevant å lytte på. Her er det ikke relevant å lytte til endringer i GameResult-objektene, siden det ikke gir mening å endre på dem etter at de er lagt inn i lista. Derimot bør det sies fra om at nye GameResult-objekter er lagt til i lista og hvor de havnet i en sortert liste. Nedenfor vises elementer fra en mulig implementasjon:

Lyttergrensesnittet:

```
public interface HighscoresListener {
	public void gameResultAdded(Highscores highscores, GameResult gameResult,
int position);
	public void gameResultRemoved(Highscores highscores, GameResult
gameResult, int position);
}
```

I Highscores-klassen:

```
public void addGameResult(GameResult gameResult) {
	...
	gameResults.add(position, gameResult);
	fireGameResultAdded(gameResult, position);
}

public void removeGameResult(GameResult gameResult) {
	int position = gameResults.indexOf(gameResult);
	if (position >= 0) {
		gameResults.remove(position);
```

```
            fireGameResultRemoved(gameResult, position);
      }
}

private List<HighscoresListener> listeners = new
ArrayList<HighscoresListener>();

public void addHighscoresListener(HighscoresListener listener) {
      listeners.add(listener);
}
public void removeHighscoresListener(HighscoresListener listener) {
      listeners.remove(listener);
}

protected void fireGameResultAdded(GameResult gameResult, int position) {
      for (HighscoresListener listener : listeners) {
            listener.gameResultAdded(this, gameResult, position);
      }
}
protected void fireGameResultRemoved(GameResult gameResult, int position) {
      for (HighscoresListener listener : listeners) {
            listener.gameResultRemoved(this, gameResult, position);
      }
}
```

Igjen er mange variasjoner mulige, men koden bør ikke (er ikke så mye grunn til å) avvike så mye fra koden over. Det er greit å begrense detaljer som posisjonen som er endret og dette vil forenkle koden en del. En kan også utelate fire-metoden, bare tilsvarende kode er med i add/removeGameResult. Lytterlista bør deklareres som List og ikke ArrayList.