NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Fredag 20. mai 2011, kl. 09:00 - 13:00

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

Språkform: Bokmål
Tillatte hjelpemidler: C
Én valgfri lærebok i Java er tillatt.
Bestemt, enkel kalkulator tillatt.

Sensurfrist: Fredag 10. 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.

Del 1 – Innkapsling (25%)

a) En kan kategorisere innkapslingsmetoder som enten lese- eller endringsmetoder. Hva er den viktigste oppgaven til endringsmetodene, bortsett fra å utføre selve endringen?

Den viktigste oppgaven er å sjekke (validere) om den nye verdien (eller nye verdiene) er lovlige/gyldige, før de evt. endres, f.eks. at et navn kun inneholder bokstaver og mellomrom.

b) Gitt en **Date**-klasse med metoder **getDay()** for å lese datoens dag (1-31), **getMonth()** for måned (1-12) og **getYear()** for år (0-99). Skriv kode for en **Person**-klasse, med felt og innkapslingsmetoder for fødselsdato og kjønn. Kjønn skal kun kunne settes ved oppretting av **Person**-objektet, mens det for fødselsdatoen ikke er noen slik begrensning.

Her er poenget å velge hvilke konstruktører og metoder som naturlig hører med til innkapslingen, inkludert navn og typer. Data som kun skal kunne settes ved oppretting krever en konstruktør med tilsvarende parameter. Andre data trenger get/set-metoder.

```
public static String MALE_GENDER = "male", FEMALE_GENDER = "female";

private String gender;
private Date dateOfBirth;

public Person(String gender) {
	if (gender != MALE_GENDER && gender != FEMALE_GENDER) {
		throw new IllegalArgumentException(gender + " is not a legal gender");
	}
	this.gender = gender;
}

public String getGender() {
	return gender;
}

public Date getDateOfBirth() {
	return dateOfBirth;
}

public void setDateOfBirth(Date dateOfBirth) {
	this.dateOfBirth = dateOfBirth;
}
```

c) Skriv kode for å lagre og innkapsle personnummer i **Person**-objekter. Du kan anta at fødselsdato og kjønn allerede er satt. Et personnummer består grovt sett av fødselsdatoen, et (vilkårlig) løpenummer og to kontrollsifre. Kontrollsifrene gjør det enklere å sjekke om et personnummer er ekte. Mer spesifikt er reglene for personnummer som følger:

- Et personnummer består av 11 siffer, med følgende struktur: **D**1**D**2**M**1**M**2**Y**1**Y**2**N**1**N**2**N**3**K**1**K**2 (fargen illustrerer siffergruppene).
- De seks første sifrene, **D**1**D**2**M**1**M**2**Y**1**Y**2, tilsvarer fødselsdatoens dag (1-31), måned (1-12) og år (0-99).
- De tre neste sifrene, **N**1**N**2**N**3, kan antas å være vilkårlige, men **N**3 må være *partall* for kvinner og *oddetall* for menn.

- De to siste sifrene, **K**1**K**2, er kontrollsifre, som hver for seg beregnes ut fra de foregående sifrene. Formellen for dem begge er **11 – (VS % 11)**[1], hvor **VS** (veid sum) for **K**1 er **D**1*__F__1 + **D**2*__F__2 + … + **N**2*__F__8 + **N**3*__F__9 og **VS** for **K**2 er **D**1*__G__1 + **D**2*__G__2 + … + **N**3*__G__9 + **K**1*__G__10. **F**'ene og **G**'ene er oppgitt i tabellen under. Dersom formelen gir et ett-sifret resultat for både **K**1 *og* **K**2 så er personnummeret gyldig, mens gir formelen et to-sifret resultat for **K**1 og/eller **K**2, så er personnummeret ugyldig.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
F	3	7	6	1	8	9	4	5	2	
G	5	4	3	2	7	6	5	4	3	2

Introduser og bruk gjerne hjelpemetoder for å gjøre koden ryddigere.

Her må en velge egnet datatype for personummeret. En kan forsåvidt lage en egen klasse, men det er greit å bruke talltabell, tegntabell eller String. En må også lage riktig valideringskode, som bør inneholde sjekk for lengde og at hvert element er et siffer, at fødselsdatoen stemmer, at kjønn stemmer med **D**3 og at kontrollsifrene er riktige. Det er naturlig å lage en hjelpemetode for formelen for kontrollsifrene evt. veid sum, siden den brukes to ganger, og ha én tabell for hvert sett med faktorer. Det kan også være nyttig med en hjelpemetode for å sjekke at to sifre stemmer med et heltall. Hjelpemetodene bør deklareres som **static**.

```
	private String pid;

	private static int[] factors1 = {3, 7, 6, 1, 8, 9, 4, 5, 2}, factors2 =     {5,
4, 3, 2, 7, 6, 5, 4, 3, 2};

	private static int computeControlDigit(String digits, int[] factors) {
		int sum = 0;
		for (int i = 0; i < factors.length; i++) {
			sum += (digits.charAt(i) - '0') * factors[i];
		}
		return 11 - (sum % 11);
	}

	private static boolean checkDigits(String digits, int pos, int num) {
		return (num / 10 == digits.charAt(pos) - '0' && num % 10 ==
digits.charAt(pos + 1) - '0');
	}

	private boolean validatePID(String pid) {
		if (pid.length() != 11) {
			return false;
		}
		for (int i = 0; i < pid.length(); i++) {
			if (! Character.isDigit(pid.charAt(i))) {
				return false;
			}
		}
		int day = dateOfBirth.getDay(), month = dateOfBirth.getMonth(), year =
dateOfBirth.getYear();
		if (! (checkDigits(pid, 0, day) && checkDigits(pid, 2, month)) &&
checkDigits(pid, 4, year)) {
```

[1] % er modulo-operatoren

```
                return false;
            }
            boolean isOdd = ((pid.charAt(8) - '0') % 2) == 1;
            if ((gender == MALE_GENDER) != isOdd) {
                return false;
            }
            int k1 = computeControlDigit(pid, factors1), k2 = computeControlDigit(pid,
factors2);
            if (k1 != pid.charAt(9) - '0' || k2 != pid.charAt(10) - '0') {
                return false;
            }
            return true;
      }

      public String getPID() {
            return pid;
      }

      public void setPID(String pid) {
            if (! validatePID(pid)) {
                throw new IllegalArgumentException(pid + " is not a valid PID for " +
gender + " and " + dateOfBirth);
            }
            this.pid = pid;
      }
```

d) Beskriv hvordan du kan bruke en såkalt *checked exception* for å avverge at fødselsdatoen endres etter at personnummeret er satt! Hva slags konsekvenser vil dette ha for kode som kaller endringsmetoden for fødselsdatoen?

En checked exception er en subklasse av Exception som ikke samtidig er en subklasse av RuntimeException. En slik Exception må deklareres vha. **throws** og den kallende metoden må enten håndtere unntaket med **try**/**catch** eller deklarere det med **throw**s.

```
      public void setDateOfBirth(Date dateOfBirth) throws Exception {
            if (pid != null) {
                throw new Exception("Cannot change date of birth after PID has been
set");
            }
            this.dateOfBirth = dateOfBirth;
      }
```

Del 2 – Klasser (40%)

Du skal implementere klasser for å representere informasjon ifm. gjennomføring av seriespill i sporter som fotball, håndball, volleyball osv. Klassen **MatchResult** skal representere informasjon om en kamp og dens resultatet, mens klassen **LeagueTable** skal representere et sett kamper og beregne serietabellen basert på kampresultatene. I del 2 skal vi begrense oss til tabeller for *ferdigspilte fotballkamper*. Et eksempel på to kampresultater og en tabell er vist nedenfor:

```
Rosenborg - Lillestrøm: 4 - 4
Lillestrøm - Rosenborg: 1 - 5
Rosenborg   4
```

```
Lillestrøm  1
```

Som vi ser har Rosenborg fått totalt 4 poeng, 3 poeng for seier og 1 poeng for uavgjort, mens Lillestrøm har fått 1 poeng for den ene uavgjort-kampen. Tabellen utelater informasjon om mål for og imot og hjemme- og bortekamper.

Tenk på helheten i løsningen før du går i gang med hver del. Kanskje kan det være lurt å begynne med c) før du skrive koden for a) og b).

Det kan være lurt å ha i bakhodet at i del 3 skal koden generaliseres til å håndtere sporter med andre regler for å beregne tabeller og i del 4 skal koden håndtere at kamper kan legges inn i tabellen før de er ferdigspilt. Men la ikke dette gjøre koden i del 2 unødvendig komplisert.

a) Implementer **MatchResult**:

- Representasjon og innkapsling av informasjon om navnene på hjemme- og bortelagene. Navnene skal ikke kunne endres etter at et **MatchResult**-objekt er laget.
- Representasjon og innkapsling av informasjon om kampresultatet, dvs. antall hjemme- og bortemål.
- Metodene **isParticipant(String participant)** for å spørre om (laget) **participant** spiller i denne kampen, **isDraw()** for å spørre om resultatet ble uavgjort og **isWinner(String participant)** for å spørre om (laget) **participant** vant denne kampen.

Her må en velge datatype for lagnavnene og målene, hhv. String og ints. Det er ikke viktig hva en kaller feltene/metodenavnene, men pga. generaliseringen i oppgave 3 så har vi valgt sportnøytrale navn. En bør ha én eller flere metoder for å sette og/eller legge til mål. Her har vi valgt å ha én metode, hvor laget som får mål angis med lagnavnet. En kan også ha en metode for hvert lag.

```
public class MatchResult {

	private final String participant1, participant2;
	private int points1 = 0, points2 = 0;

	public MatchResult(String participant1, String participant2) {
		this.participant1 = participant1;
		this.participant2 = participant2;
	}

	public String toString() {
		return participant1 + " - " + participant2 + ": " + points1 + " - " +
points2;
	}

	public String getParticipant1() {
		return participant1;
	}

	public String getParticipant2() {
		return participant2;
	}

	public boolean isParticipant(String participant) {
		return participant.equals(participant1) || participant.equals(participant2);
	}
```

```
	public int getPoints(String participant) {
		if (participant.equals(participant1)) {
			return points1;
		} else if (participant.equals(participant2)) {
			return points2;
		} else {
			return -1;
		}
	}

	public boolean isWinner(String participant) {
		return participant.equals(participant1) && points1 > points2 ||
participant.equals(participant2) && points2 > points1;
	}

	public boolean isDraw() {
		return points1 == points2;
	}

	public void addPoints(String participant, int points) {
		if (participant.equals(participant1)) {
			points1 += points;
		} else if (participant.equals(participant2)) {
			points2 += points;
		}
	}
}
```

b) Implementer **LeagueTable**:

- Representasjon av (navnene til) alle lagene. Navnene skal ikke kunne endres etter at **LeagueTable**-objektet er laget.
- Representasjon av informasjon om kampresultater, dvs. **MatchResult**-objekter. Det skal kun være lov å legge inn nye kamper, ikke fjerne dem. Det skal ikke være lov å legge inn kamper for lag som ikke er med i tabellen.
- Metoden **getParticipantPoints(MatchResult matchResult, String participant)** som returnerer antall poeng som (laget) **participant** fikk for resultatet **matchResult**. Ta høyde for at **participant** faktisk ikke er med i **matchResult**.

Tabellpoeng summeres basert på kampresultatene, og tabellen sorteres etter antall poeng, med laget med flest poeng øverst. Når et kampresultat legges inn, så skal tabellen oppdateres.

Merk at du antageligvis vil trenge en klasse for å representere hver rad i tabellen, f.eks. kalt **LeagueTableRow**, som lagrer data om et lag og summen av poengene laget har fått for seier og uavgjort i sine kamper. Denne klassen kan implementere relevante grensesnitt ifm. sortering.

Nedenfor er en rett frem implementasjon av tabellraden. Denne implementerer Comparable-grensesnittet for å støtte sortering med Java sin sort-metode og har derfor metoden compareTo. Merk at logikken er slik at objektet som skal sorteres som først i en liste logisk sett må være minst og derfor returnere < 0 i sammenligningen.

```
public class LeagueTableRow implements Comparable<LeagueTableRow> {

	private String participant;
```

```
	public LeagueTableRow(String participant) {
		this.participant = participant;
	}

	public String getParticipant() {
		return participant;
	}

	private int points = 0;

	public int getPoints() {
		return points;
	}

	public void addPoints(int points) {
		this.points += points;
	}

	// Comparable<LeagueTableRow>

	public int compareTo(LeagueTableRow other) {
		return other.points - points;
	}
}
```

Tabell-klassen må ha en konstruktør som tar inn navnene til lagene. Det er naturlig å opprette én tabellrad for hvert lag og en trenger ikke å huske navnene, siden de ligger i radene. En må ha en metode for å legge til et resultat som oppdaterer antall poeng for hvert lag og sorterer i etterkant. Innkapsling av radene, dvs. at en har metoder for å lese radene, er ikke så sentralt. Det kan gjøres på ulike måter og her gjør vi det enkelt med en iterator()-metode. (Siden vi har iterator()-metoden så har vi markert at klassen implementerer Iterable, men det er ikke viktig.)

```
public abstract class LeagueTable implements MatchListener, Iterable<LeagueTableRow>{

	private List<MatchResult> matchResults;
	private List<LeagueTableRow> tableRows;

	protected LeagueTable(List<String> participantNames) {
		this.matchResults = new ArrayList<MatchResult>();
		this.tableRows = new ArrayList<LeagueTableRow>();
		for (String participant : participantNames) {
			tableRows.add(new LeagueTableRow(participant));
		}
	}

	private LeagueTableRow findEntry(String participant) {
		for (LeagueTableRow entry : tableRows) {
			if (entry.getParticipant().equals(participant)) {
				return entry;
			}
		}
		return null;
	}

	public Iterator<LeagueTableRow> iterator() {
		return tableRows.iterator();
```

```
	}

	public int getParticipantPoints(MatchResult match, String participant) {
		if (match.isWinner(participant)) {
			return 3;
		} else if (match.isParticipant(participant) && match.isDraw()) {
			return 1;
		}
		return 0;
	}

	private void addPoints(MatchResult matchResult, String partipant) {
		findEntry(partipant).addPoints(getParticipantPoints(matchResult, partipant));
	}

	public void addMatchResult(MatchResult matchResult) {
		if (findEntry(matchResult.getParticipant1()) == null ||
findEntry(matchResult.getParticipant2()) == null) {
			throw new IllegalArgumentException("Both teams must be part of the
league");
		}
		matchResults.add(matchResult);
		addPoints(matchResult, matchResult.getParticipant1());
		addPoints(matchResult, matchResult.getParticipant2());
		Collections.sort(tableRows);
	}
}
```

c) Tegn et objekt/instansdiagram (figur som viser instansene i en objektstruktur og deres innhold) som tilsvarer tabellen over, for strukturen av **MatchResult**-, **LeagueTable**- og evt. **LeagueTableRow**-objekter.

Denne deloppgaven er ment å hjelpe en å tenke over (logikken til) datastrukturen. Instansdiagram viser hvilke objekter som eksisterer, verdiene til feltene og koblinger dem imellom. En kan velge å tegne koblinger som piler med navn på eller med eksplisitte tabeller med piler. Vi er uansett ikke nøye på notasjonen. Det skulle for øvrig stått at instansene skulle tilsvare tabelleksemplet over, men det ble ikke med.

d) I en ordentlig fotballtabell så spiller også totalt antall mål for og imot og resultat i innbyrdes kamper inn på tabellen. Beskriv kort med tekst og kode hvordan du vil støtte dette.

Klassen for en tabellrad må utvides til å holde mer informasjon. Koden for compareTo må først sjekke tallene som har størst prioritet ift. sammenligningen, f.eks. beregne differansen og returnere denne dersom den er ulik 0. Så fortsetter en på samme måte med de andre tallene for sammenligning. Håndtering av innbyrdes oppgjør er vanskelig og krever at en slår opp i tabellens liste over kamper. Det kan gjøres på to måter, enten ved å inkludere en referanse til tabellen i tabellraden eller ved å la tabellklassen implementere Comparator, fordi en da kan slå opp innbyrdes oppgjør i tabellen.

Del 3 – Generalisering og arv (20%)

Ulike sporter har ulike regler for poenggiving i både kamper og serietabeller og sortering av tabellene. I tennis er kampen slutt når én deltaker har vunnet 2 eller 3 sett for hhv. kvinner og menn, så sluttresultatet 1-0 finnes ikke i tennis. Fotball gir 3 tabellpoeng for seier, mens håndball gir 2. I noen sporter teller innbyrdes oppgjør før målforskjell ved sortering, mens det for andre er omvendt.

a) Forklar med tekst og kode hvordan du kan lage sport-spesifikke resultat-klasser, f.eks. **TennisResult**, som arver fra **MatchResult** og implementerer logikk for å håndheve sportens regler for poenggiving, bl.a. for å unngå registrering av ugyldige resultater. Beskriv også nødvendige endringer av **MatchResult**.

Litt av poenget her er å tenke på hva slags grensesnitt subklassen må ha, og at ikke superklassen gir **public**-tilgang til mer enn nødvendig og **protected**-tilgang til det subklassen trenger. **MatchResult**-klassen bør markeres som **abstract**. Metoden for å legge til mål/poeng i **MatchResult** (**addPoints**) bør gjøres **protected** og så lager en sport-spesifikke metoder for å legge til mål/poeng i subklassen, som kaller metoden i superklassen. Her er et forslag for fotball:

```
public FootballMatch(String homeName, String awayName) {
      super(homeName, awayName);
}

public void addHomeGoal() {
      addPoints(getParticipant1(), 1);
}

public void addAwayGoal() {
      addPoints(getParticipant2(), 1);
}
```

b) Forklar med tekst og kode hvordan du kan lage sport-spesifikke tabell-klasser, f.eks. **FootballLeagueTable** og **TennisTournamentTable**, som arver fra **LeagueTable** og implementerer logikk iht. sportens regler for å beregne tabellpoeng. Beskriv også nødvendige endringer av **LeagueTable**.

LeagueTable-klassen og **getParticipantPoints**-metoden bør markeres som **abstract**. **getParticipantPoints**-metoden implementeres så i de sport-spesifikke subklassene (koden fra del 2 vil være en naturlig del av en fotballtabell-subklasse).

Del 4 – Diverse (15%)

a) Hva slags generell teknikk kan anvendes for å sikre at en serietabell automatisk oppdateres hvis et **MatchResult**-objekt endres etter at det er lagt inn i tabellen? Forklar med tekst og kode hvordan du vil implementere støtte for dette. Fokusér på å forklare hvordan den generelle teknikken anvendes/tilpasses, ikke på å få logikken for poengberegning helt riktig (kan være litt intrikat i dette tilfellet).

Her må en bruke observatør-observert-teknikken. Trikset er å la tabellen lytte til endringer i målene/poengene i et kampresultat. Når endringen skjer så må en 1) fjerne poengene en hadde delt ut basert på resultatet før endringen og 2) legge til poengene basert på resultatet etter endringen og 3) sortere igjen. Dette krever at lyttergrensesnitt og –metoden får nok informasjon til å utlede resultat før endringen, noe som kan være litt fiklete. En grei (men ineffektiv) måte å gjøre det på er å blanke hele tabellen og beregne den på nytt ved å gå gjennom alle kampene.

En bør helst forklare hvordan en generelt implementerer lytting med 1) et grensesnitt, 2) liste av lyttere, 3) add/remove-metoder for lyttere, 4) metode for å kringkaste endringshendelser til lytterne og 5) kall av kringkastingsmetoden når endringen skjer.

b) Forklar hvordan du vil teste at:
- metoden **isWinner(…)** i 2 a) pkt. 3 er riktig implementert
- innbyrdes sammenligning av tabellrader i 2 b) er korrekt
- løsningen i 1 d) er korrekt

Du trenger ikke skrive kode, men forklaringen må være detaljert nok til å illustrere testemetoden.

På de to første er poenget å teste et rikt nok utvalg tilfeller. For **isWinner()** i 2 a) bør en teste tilfeller med seier og tap for hjemme- og bortelag og uavgjortresultat og sjekke at **isWinner()** returnerer riktig verdi for begge lagene. For sammenligning av tabellrader i 2 b) bør en teste tilfeller hvor returverdien blir <, > og = 0 (hvis en har brukt Comparable-grensesnittet) eller tilsvarende. En kan også teste at sammenligning med seg selv gir 0, for en skal jo ikke bare teste naturlig input. For løsningen i 1 d) er poenget korrekt håndtering av unntak. Trikset er å ha et kall til **fail()** i **try**- eller **catch**-delen alt ettersom en forventer et unntak eller ikke. En bør helst bruke **instanceof** for sjekke at unntaket ikke er en subklasse av **RuntimeException**. Eksempel på kode for Exception-testing:

```
public void testSetDateOfBirth() {
	try {
		person.setDateOfBirth(date);
	} catch (Exception e) {
		fail();
	}
	try {
		person.setPID(pid);
	} catch (RuntimeException e) {
	}
	try {
		person.setDateOfBirth(null);
		fail();
	} catch (Exception e) {
		assertFalse(e instanceof RuntimeException);
	}
}
```