NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Lørdag 19. 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
Kun Java Pocket Guide, utgitt av O'Reilly forlag, er tillatt.

Sensurfrist: Mandag 11. 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 nødvendig.

Del 1 – Innkapsling (20%)

Gitt følgende klasse, som representerer et *tidspunkt* på dagen:

```
public class DayTime {

	public final int hours, minutes;

	public DayTime(int hours, int minutes) {
		this.hours = hours;
		this.minutes = minutes;
	}

	public String toString() {
		return hours + ":" + minutes;
	}
}
```

a) Hva betyr **final**-nøkkelordet slik det er brukt her? På hva slags måte(r) ivaretar denne klassen formålet med innkapsling og på hva slags måte(r) ikke?

final betyr her at feltet ikke kan endres etter at det er satt i konstruktøren. Selv om feltene er **public** så sikres innkapsling ved at felt-verdiene forblir korrekte, siden kode utenfor klassen ikke kan sette feltene til ugyldige verdier. Imidlertid er det ikke i tråd med innkapsling at kode gjøres avhengig av at data er lagret i spesifikke felt. Ved bruk av get-metoder får implementasjonsklassen større frihet til å endre interne detaljer, uten at annen kode blir påvirket.

b) Beskriv den generelle teknikken og navnekonvensjonen(e) for å representere og kapsle inn en enkel verdi, f.eks. tall eller objektreferanse, som skal kunne endres etter at objektet er opprettet.

Gitt verdi med (logisk) navn ”value” og type X, så vil en ha felt, get- og set-metoder som følger:
private X value;
public X getValue() { return value;}
public void setValue(X value) { this.value = value;}
Dersom X er boolean/Boolean, så brukes gjerne ”is” som prefiks istedenfor ”get”.

c) I hva slags metode(r) brukes *unntak* ifm. innkapsling, og hvordan? Vis gjerne med et eksempel!

I metoder som endrer (en verdi i) et objekt, så bør argumenter valideres. Dette må skje før selve endringen og i tilfelle ugyldig(e) verdier så kastes et unntak av typen IllegalArgumentException(…). Eksempel:

```
public void setValue(int value) {
  if (value < 0) {
    throw new IllegalArgumentException(”Value shouldn’t be negative, but was ” + value);
  }
  this.value = value;
}
```

d) Mange klasser inneholder et List<X>-felt og definerer addX, removeX og andre metoder som tilsvarer og bruker metodene i List-grensesnittet. En alternativ løsning kunne vært å arve fra en List<X>-implementasjon, f.eks. ArrayList<X>. Hvorfor brukes aldri denne løsningen i praksis?

Det vi altså spør om her er hvorfor vi f.eks. gjør slik

```
public class Person {
  List<Person> children = … // contains list of children
  public void addChild(Person p) { … }
  public void removeChild(Person p) { … }
}
```

og ikke slik

```
public class Person extends ArrayList<Person> {
  // trenger ikke add- og remove-metoder, fordi vi arver dem fra ArrayList
}
```

- Når en arver så kan en ikke kun arve de metodene en ønsker, men får alle med på kjøpet. Da vil en være nødt til å redefinere alle en ikke trenger/ønsker at skal være tilgjengelig.
- Det vil ikke alltid være logisk riktig at klassen skal være instanceof List<X>.
- Teknikken kan bare brukes for én slik liste, siden en bare kan arve fra én implementasjonsklasse.

Del 2 – Klasser (50%)

I denne oppgaven skal du implementere klasser og metoder for å håndtere en dagsplan med tidsrom, f.eks. avtaler. Klassen oppgitt i oppgave 1 a) kan brukes som en del av implementasjonen din. Husk at alle metoder kan brukes i implementasjonen av andre.

a) Du skal implementere en klasse **TimeSlot** for å representere et *tidsrom* innenfor en dag, f.eks. en avtale. Et **TimeSlot**-objekt har en beskrivelse (tekst), et start- og sluttidspunkt (time og minutt) og en varighet (minutter). Ingen av disse dataene skal kunne endres etter at **TimeSlot**-objektet er opprettet. Du velger selv hvordan disse dataene skal representeres og hvilke hjelpemetoder som evt. trengs. Merk at varighet kan beregnes fra starttidspunkt og sluttidspunkt, evt. at sluttidspunkt kan beregnes fra starttidspunkt og varighet.

Implementer følgende konstruktører og metoder:
- **TimeSlot(String description, int hours, int minutes, int duration)**: initialiserer med oppgitt beskrivelse og starttidspunkt (timer og minutter) og varighet (minutter). F.eks. vil **new TimeSlot(”TDT4100-forelesning”, 10, 15, 105)** representere en TDT4100-forelesning i tidsrommet 10:15-12:00.
- **String toString()**: returnerer en **String** på formen **beskrivelse@start-slutt** hvor **start** og **slutt** er på formen **tt:mm** (altså to siffer pr. tall). Dersom **toString()**-metoden kalles på **TimeSlot**-objektet fra forrige punkt skal det gi **”TDT4100-forelesning@10:15-12:00”**.
- **DayTime getStartTime()**: returnerer starttidspunktet som et **DayTime**-objekt (se oppgave 1).
- **DayTime getEndTime()**: returnerer sluttidspunktet som et **DayTime**-objekt (se oppgave 1).
- **int getDuration()**: returnerer varighet i minutter

Vi velger å representere start- og sluttidspunktene som antall minutter siden midnatt. Dette gjør mange av beregningene siden lettere. Et alternativ er å ha to DayTime-felt i stedet.

```
private static int asHours  (int minutes) { return minutes / 60;}
private static int asMinutes(int minutes) { return minutes % 60;}

private String description;
```

```
	private int startTime, endTime;

	public TimeSlot(String description, int hours, int minutes, int duration) {
		this.description = description;
		this.startTime = hours * 60 + minutes;
		this.endTime = this.startTime + duration;
	}

	private String twoDigits(int i) {
		return (i < 10 ? "0" : "") + i;
	}
	public String toString() {
		return description + "@" +
		twoDigits(asHours(startTime)) + ":" + twoDigits(asMinutes(startTime)) + "-" +
		twoDigits(asHours(endTime)) + ":" + twoDigits(asMinutes(endTime));
	}

	public DayTime getStartTime() {
		return new DayTime(asHours(startTime), asMinutes(startTime));
	}
	public DayTime getEndTime() {
		return new DayTime(asHours(endTime), asMinutes(endTime));
	}
	public int getDuration() {
		return endTime - startTime;
	}
	public String getDescription() {
		return description;
	}
```

b) Implementer følgende metoder:
- **boolean contains(int hours, int minutes)**: returnerer om dette **TimeSlot**-objektet inneholder tidspunktet angitt med **hours** og **minutes**. Merk at sluttidspunktet regnes ikke som å være inneholdt i tidsrommet. Dette betyr at **new TimeSlot("…", 8, 0, 30).contains(8, 0)** skal gi **true**, mens **new TimeSlot("…", 8, 0, 30).contains(8, 30)** skal gi **false**.
- **boolean overlaps(TimeSlot timeSlot)**: returnerer om dette **TimeSlot**-objektet overlapper med det angitte **TimeSlot**-objektet, dvs. om det finnes et tidspunkt som begge inneholder.

Her er litt av poenget å ha riktig type sammenligning i hver ende av intervallet.

```
	public boolean contains(int hours, int minutes) {
		minutes = hours * 60 + minutes;
		return startTime <= minutes && endTime > minutes;
	}

	public boolean overlaps(TimeSlot other) {
		return startTime < other.endTime && endTime > other.startTime;
	}
```

c) **TimeSlot**-klassen skal støtte sortering. **TimeSlot**-objektet med tidligst starttidspunkt sorteres først, og dersom starttidspunktene er like, så skal det med tidligst sluttidspunkt sorteres først. Forklar og implementer nødvendig kode.

Sortering krever at klassen implementerer Comparable spesialisert til samme klasse.

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

	...

	public int compareTo(TimeSlot other) {
		int diff = startTime - other.startTime;
		if (diff == 0) {
			diff = endTime - other.endTime;
		}
		return diff;
	}
}
```

d) Du skal implementere en klasse **DayPlan**, for å holde oversikt over alle avtalene (altså **TimeSlot**-objekter) for en dag, bl.a. gi muligheten til å legge til og fjerne **TimeSlot**-objekter. Velg selv hvilke felt og evt. hjelpemetoder som trengs.

Implementer følgende metoder:
- **void addTimeSlot(TimeSlot timeSlot)**: legger det angitte tidsrommet til denne dagsplanen
- **void removeTimeSlot(TimeSlot timeSlot)**: fjerner det angitte tidsrommet fra denne dagsplanen
- **TimeSlot getTimeSlotAt(int hours, int minutes)**: returnerer det *tidligste* tidsrommet som inneholder tidspunktet angitt med **hours** og **minutes**, ellers null.

```
	private List<TimeSlot> timeSlots = new ArrayList<TimeSlot>();

	public void addTimeSlot(TimeSlot timeSlot) {
		timeSlots.add(timeSlot);
		Collections.sort(timeSlots);
	}

	public void removeTimeSlot(TimeSlot timeSlot) {
		timeSlots.remove(timeSlot);
	}

	public TimeSlot getTimeSlotAt(int hours, int minutes) {
		for (TimeSlot timeSlot : timeSlots) {
			if (timeSlot.contains(hours, minutes)) {
				return timeSlot;
			}
		}
		return null;
	}
```

e) Implementer følgende to metoder for tidsplanlegging:
- **boolean containsOverlapping()**: returnerer om det finnes overlappende tidsrom i denne dagsplanen.
- **Collection<TimeSlot> getFreeTime()**: returnerer en samling **TimeSlot**-objekter som representerer *fritiden* en har i løpet av en dag, dvs. tidsrommene som denne dagsplanen *ikke* dekker.

For begge disse metodene kan det være lurt å definere hjelpemetoder for å gjøre løsningen ryddigere.

```
	public boolean containsOverlapping() {
		for (int i = 0; i < timeSlots.size(); i++) {
			TimeSlot timeSlot = timeSlots.get(i);
			for (int j = i + 1; j < timeSlots.size(); j++) {
				TimeSlot otherTimeSlot = timeSlots.get(j);
```

```
                        if (timeSlot != otherTimeSlot &&
timeSlot.overlaps(otherTimeSlot)) {
                              return true;
                        }
                  }
            }
            return false;
      }

      private static void addTimeSlotIfNonEmpty(Collection<TimeSlot> timeSlots, DayTime
startTime, DayTime endTime) {
            int duration = (endTime.hours * 60 + endTime.minutes) - (startTime.hours * 60
+ startTime.minutes);
            if (duration > 0) {
                  timeSlots.add(new TimeSlot(null, startTime.hours, startTime.minutes,
duration));
            }
      }

      public Collection<TimeSlot> getFreeTime() {
            Collection<TimeSlot> freeTime = new ArrayList<TimeSlot>();
            TimeSlot previous = new TimeSlot(null, 0, 0, 0);
            for (TimeSlot timeSlot : getAllTimeSlots()) {
                  DayTime startTime = timeSlot.getStartTime();
                  addTimeSlotIfNonEmpty(freeTime, previous.getEndTime(), startTime);
                  DayTime endTime = timeSlot.getEndTime();
                  if (! previous.contains(endTime.hours, endTime.minutes)) {
                        previous = timeSlot;
                  }
            }
            addTimeSlotIfNonEmpty(freeTime, previous.getEndTime(), new DayTime(24, 0));
            return freeTime;
      }
```

Del 3 – Arv og delegering (20%)

a) Du skal implementere støtte for TDT4100-forelesninger, som et spesielt tidsrom som alltid er fra **10:15** til **12:00**. Vis hvordan en klasse **TDT4100Lecture** kan implementere dette vha. arv fra **TimeSlot**.

Velger å bruke konstruktøren for å sette riktige verdier. Alternativt kan get-metoden redefineres til å returnere riktige (konstant)verdier.

```
public class TDT4100Lecture extends TimeSlot {

      public TDT4100Lecture() {
            super("TDT4100 lecture", 10, 15, 105);
      }
}
```

b) Du skal implementere støtte for en dagsplan som alltid inneholder et **TDT4100Lecture**-objekt, dvs. en TDT4100-forelesning fra 10:15-12:00. Det skal ikke være mulig å fjerne TDT4100-forelesningen eller legge inn andre tidsrom som overlapper med den. Følgende kode illustrerer hvordan det skal virke:

```
DayPlan tuesday = new TDT4100DayPlan();
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00"
tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
// throws appropriate exception, since it overlaps with the TDT4100 lecture
```

Vis hvordan dette kan implementeres i en **TDT4100DayPlan**-klasse vha. arv fra **DayPlan**.

```
public class TDT4100DayPlan extends DayPlan {

	private TDT4100Lecture tdt4100Lecture;

	public TDT4100DayPlan() {
		super();
		tdt4100Lecture = new TDT4100Lecture();
		super.addTimeSlot(tdt4100Lecture);
	}

	@Override
	public void addTimeSlot(TimeSlot timeSlot) {
		if (timeSlot.overlaps(tdt4100Lecture)) {
			throw new IllegalArgumentException("Cannot overlap TDT4100 lecture!");
		}
		super.addTimeSlot(timeSlot);
	}

	@Override
	public void removeTimeSlot(TimeSlot timeSlot) {
		if (timeSlot == tdt4100Lecture) {
			throw new IllegalArgumentException("Cannot remove TDT4100 lecture!");
		}
		super.removeTimeSlot(timeSlot);
	}
}
```

c) Det er ofte nyttig å kunne la én dagsplan bygge på eller inkludere en eller flere andre, f.eks. la dagsplanen for en bestemt tirsdag inkludere tirsdagsplanen som gjelder for hele semesteret (hvor bl.a. TDT4100-forelesningen ligger). Følgende kode illustrerer hvordan det skal virke:

```
DayPlan repeatingTuesday = new TDT4100DayPlan();
DelegatingDayPlan tuesday = new DelegatingDayPlan(repeatingTuesday);
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00" since tuesday logically includes the
TDT4100Lecture in repeatingTuesday

tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
System.out.println(tuesday.containsOverlapping());
// prints "true" since timeSlot in tuesday overlaps with TDT4100Lecture in
repeatingTuesday
```

Skisser med tekst og kode hvordan *delegeringsteknikken* og arv fra **DayPlan** kan brukes for å implementere denne oppførselen, inkludert hvordan du evt. vil modifisere **DayPlan** for å gjøre løsningen ryddigere.

Delegering er en teknikk hvor et objekt, videreformidler kall til en "delegat" når det er behov for delegatens ferdigheter. I dette tilfellet er det viktig at TimeSlot-objektene som ligger i delegaten regnes med i logikken. F.eks. må **getTimeSlotAt(…)**-metoden sjekke egne TimeSlot-objekter og delegatens og returnere det tidligste av de to.

I løsningen innføres en **getAllTimeSlots()**-metode i **DayPlan**, som brukes istedenfor **this.timeSlots** i metodene **containsOverlapping()** og **getFreeTime()** i **DayPlan** som trenger å behandle alle **TimeSlot**-objektene på en gang. Dermed trenger vi bare å redefinere **getAllTimeSlots()**.

```
public class DelegatingDayPlan extends DayPlan {

	private DayPlan delegate;

	public DelegatingDayPlan(DayPlan delegate) {
		super();
		this.delegate = delegate;
	}

	//

	@Override
	public TimeSlot getTimeSlotAt(int hours, int minutes) {
		TimeSlot timeSlot1 = super.getTimeSlotAt(hours, minutes);
		TimeSlot timeSlot2 = delegate.getTimeSlotAt(hours, minutes);
		if (timeSlot1 != null && timeSlot2 != null) {
			return timeSlot1.compareTo(timeSlot2) < 0 ? timeSlot1 : timeSlot2;
		} else if (timeSlot1 != null) {
			return timeSlot1;
		} else {
			return timeSlot2;
		}
	}

	//

	@Override
	protected Collection<TimeSlot> getAllTimeSlots() {
		Collection<TimeSlot> timeSlots = super.getAllTimeSlots();
		if (delegate != null) {
			List<TimeSlot> timeSlotList = new ArrayList<TimeSlot>(timeSlots);
			timeSlotList.addAll(delegate.getAllTimeSlots());
			Collections.sort(timeSlotList);
			timeSlots = timeSlotList;
		}
		return timeSlots;
	}
}
```

Del 4 – Input/output (IO) (10%)

a) Hva er den grunnleggende forskjellen på input/output-klassene **InputStream**/**OutputStream** og deres subklasser ift. **Reader**/**Writer** og deres subklasser?

InputStream-/**OutputStream**-klassene håndterer **byte**-verdier, mens **Reader**-/**Writer**-klassene håndterer char-verdier, som krever koding av tegn til/fra bytes (iht. Unicode-regler).

b) Ifm. input/output brukes en egen type unntak, hvilken? På hva slags måte påvirker det kode som driver med input/output?

Input/output-metoder kaster gjerne **IOException**, som er en såkalt "checked exception". Slike brukes gjerne for feil som er utenfor vår kontroll. Disse krever at kode må fange dem opp med **try/catch** eller deklarere med **throws** at de kastes videre.

c) Hvorfor må vi lukke input- og output-"strømmer" med **close()**-metoden når vi er ferdige med dem? Hvordan sikrer man at det skjer også i tilfelle unntak?

Strømmer bruker gjerne ressurser utenfor Java og **close()**-metoden sikrer at Java samhandler riktig med disse, f.eks. frigjør dem. For å sikre at dette alltid skjer, er det vanlig å ha **close()**-kallet i en **try**/**finally**-blokk.