NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Lørdag 22. mai 2010, 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: Mandag 14. 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 (20%)

a) Hva er hensikten med innkapsling?

Innkapsling skal sikre et objekt starter og forblir i en *gyldig tilstand* og at innholdet kan leses og endres uten å avdekke den *interne representasjonen og implementasjonen*.

b) Lag en **`Date`**-klasse med metoder for å lese og endre informasjon om *dag*, *måned* og *år*, slik at denne informasjonen blir ordentlig *innkapslet* og *validert*. Velg selv konstruktør(er) og metoder som inngår i innkapslingen, egnet intern representasjon og evt. interne hjelpemetoder. Du kan anta det finnes en hjelpemetode som sier om et år er et skuddår. Gjør kort rede for dine valg og husk at det finnes flere fullgode løsninger.

Det finnes flere fullgode løsninger. Den enkleste (naïve) løsningen er ett int-felt (privat) for hhv. dag, måned og år og get/set-metoder (public) for hver av disse. I hver set-metode bør en validere om endringen vil gi en ugyldig dato, *før* selve endringen skjer. Denne valideringen er lik for alle set-metodene og bør legges i en egen valideringsmetode (public). Ved feil, bør en bruke en IllegalArgumentException eller RuntimeException, eller en egendefinert subklasse av en av disse. Det er ryddig å ha en egen metode som returnerer antall dager i en måned, gitt året, som kan brukes i valideringsmetoden og i c).

Det kan argumenteres for at det er uheldig å teste for gyldighet av hele tilstanden, når tilstanden kan endres bitvis. Da bør det være en begrunnelse, og valideringsmetoden må være public, slik at ekstern kode kan validere selv. Problemstillingen kan unngås ved å velge én set-metode for all informasjonen, slik at hele tilstanden settes (og valideres) på en gang.

Vi velger her det første alternativet.

```
public class Date {

	private int day, month, year;

	public Date(int day, int month, int year) {
		check(day, month, year);
		this.day = day;
		this.month = month;
		this.year = year;
	}

	private boolean isLeapYear(int year) {
		return (year % 4 == 0 && (year % 400 == 0 || year % 100 != 0));
	}

	private int numberOfDays(int month, int year) {
		switch (month) {
		case 1: case 3: case 5: case 7: case 8: case 10: case 12: return 31;
		case 4: case 6: case 9: case 11: return 30;
		case 2: return (isLeapYear(year) ? 29 : 28);
		}
		return -1;
	}
```

```
	private void check(int day, int month, int year) {
		if (day < 1 || day > numberOfDays(month, year)) {
			throw new IllegalArgumentException("day is illegal: " + day);
		}
		if (month < 1 || month > 12) {
			throw new IllegalArgumentException("month is illegal: " + day);
		}
	}

	public int getDay() {
		return day;
	}

	public void setDay(int day) {
		check(day, month, year);
		this.day = day;
	}

	public int getMonth() {
		return month;
	}

	public void setMonth(int month) {
		check(day, month, year);
		this.month = month;
	}

	public int getYear() {
		return year;
	}

	public void setYear(int year) {
		check(day, month, year);
		this.year = year;
	}
}
```

c) Implementer metoder for å endre datoen til neste og forrige dag. Merk at disse metodene skal endre **`Date`**-objektet selv, ikke lage nye **`Date`**-objekter.

Det er mest snakk om å holde tunga rett i munnen.

```
	public void nextDay() {
		day++;
		if (day > numberOfDays(month, year)) {
			day = 1;
			month++;
			if (month > 12) {
				month = 1;
				year++;
			}
		}
	}

	public void previousDay() {
		day--;
		if (day < 1) {
```

```
                month--;
                if (month < 1) {
                        month = 12;
                        year--;
                }
                day = numberOfDays(month, year);
        }
    }
```

Del 2 – Typer (10%)

Anta følgende klasser, grensesnitt og metoder:
- Klasse **A** deklarerer metoden **`A methodA()`**.
- Grensesnittet **G** deklarerer metoden **`G methodG(A)`** .
- Klasse **B** arver fra **A**, implementerer **G** og deklarerer metoden **`B methodB(A)`**.
- Klasse **C** implementerer **G** og deklarerer metoden C **`methodC(G)`**.

a) Hvilke av følgende deklarasjoner/initialiseringer vil gi feil i editoren/ved kompilering:

```
A a = new B();
B b = new A();
G g1 = new A(), g2 = new B(), g3 = new C();
```

Her ble det (dessverre) ikke spurt om noen forklaring, så det er nok å angi hvilke som er feil.
`B b = new A()` er ulovlig, siden en A ikke er en B
`G g1 = new A()`er ulovlig, siden en A ikke implementerer G

b) Anta i det følgende at variablene **a**, **b** og **c** er deklarert til å være av klassene **A**, **B** og **C**. Hvilke av følgende uttrykk vil gi feil i editoren/ved kompilering og hvorfor:

```
a.methodA() == c.methodC(a)
b.methodA().toString()
b.methodG(new B())
((G) a).methodG(a)
((B) a).methodG(b)
((C) a).methodG(a)
```

`c.methodC(a)` er ulovlig, siden en A ikke er en C
`((C) a).methodG(a)` er ulovlig, siden en A ikke også kan være en C

Del 3 – Klasser (35%)

For å kunne stave ord over en dårlig kommunikasjonslinje er det vanlig å bruke et ord-basert alfabet, hvor første bokstav i ordet tilsvarer bokstaven som sies. Nato har standardisert dette i sitt fonetiske alfabet, som begynner med ”alfa”, ”bravo”, ”charlie”, ”delta”, ”echo”, ”foxtrot”. Dersom ordet ”java” staves med dette alfabetet blir det ”juliet alfa viktor alfa”.

Du skal implementere en klasse **`RadioAlphabet`** for å konvertere bokstaver og ord vha. denne teknikken. Du velger selv hvordan alfabetet representeres, f.eks. kan en **`String`**, **`String`**-tabell, **`List`** eller **`Map`** brukes for å representere alfabetet. Merk at det kan være lurt å lese gjennom hele oppgaven og ta en titt på oppgave 5, før du velger mellom en av disse og evt. andre alternativer.

a) Først skal du implementere én konstruktør og to metoder, som følger:

- Konstruktøren skal ta inn ordene i alfabetet som en String og en String som inneholder skilletegnet. Merk at ordene trenger ikke være i alfabetisk rekkefølge, fordi første bokstav i ordet avgjør hvilken bokstav det tilsvarer. F.eks. skal **`new RadioAlphabet(”alfa-bravo-delta-charlie”, ”-”)`** gi et objekt som kan oversette bokstavene ’a’-’d’ med Nato-alfabetet. Alfabetet skal kun kunne inneholde ord for bokstaver (altså ikke tall og skilletegn).

- Metoden **`String convert(char)`** skal returnere ordet som tilsvarer bokstaven som gis som parameter, eller **`null`** dersom bokstaven ikke kan oversettes. F.eks. skal **`convert(’a’)`** returnere **`”alfa”`** og **`convert(’!’)`** skal returnere **`null`**.

- Metoden **`String convert(String)`** skal returnere sekvensen av ord (med mellomrom mellom) i en **`String`** som tilsvarer ordet som gis som parameter. Bokstaver som ikke kan oversettes skal ignoreres. F.eks. skal **`convert(”*abba*”)`** returnere **`”alfa bravo bravo alfa”`**.

Det enkleste er å bruke List/ArrayList eller en Map/HashMap, siden disse har dynamisk størrelse. En Map gir kanskje aller enklest kode (litt mer kompleks konstruktør), men her vises en løsning som bruker List. Lista inneholder kun alfabet-ordene og koden utnytter at første bokstav i ordet er nettopp bokstaven som ordet er oversettelsen av.

Split bør være kjent og gir en enklere konstruktør. Merk hvordan pos-metoden, som finner hvor ordet for en bokstav er, brukes av flere av de andre metodene. I convert-metoden kan bruke String +=, StringBuilder.append eller StringBuffer.append. Merk at det kreves egen logikk for å unngå unødvendige blanke tegn, foran inni eller i slutten av String’en. Siden en bokstav ikke nødvendigvis kan oversettes, er det ikke nok å sjekke indeksen for å avgjøre om det skal legges inn mellomrom. Her løses det delvis med en if, delvis med String.trim.

```
public class RadioAlphabet {

	private List<String> alphabet;

	public RadioAlphabet(String alphabet, String separator) {
		this.alphabet = new
ArrayList<String>(Arrays.asList(alphabet.split(separator)));
	}
```

```
	private int pos(char c) {
		for (int i = 0; i < alphabet.size(); i++) {
			if (alphabet.get(i).charAt(0) == c) {
				return i;
			}
		}
		return -1;
	}

	public String convert(char c) {
		int pos = pos(c);
		return pos >= 0 ? alphabet.get(pos) : null;
	}

	public String convert(String word) {
		StringBuffer /* or StringBuilder */ buffer = new StringBuffer();
		for (int i = 0; i < word.length(); i++) {
			String converted = convert(word.charAt(i));
			if (converted != null) {
				buffer.append(converted);
				buffer.append(' ');
			}
		}
		return buffer.toString().trim();
	}
```

b) Hvorfor kan ikke disse metodene være deklarert med **`static`**-modifikatoren?

Med static-modifikatoren vil metodene ikke kunne referere til et bestemt RadioAlphabet-objekt. En kan for så vidt gjøre alfabetet static også, men da vil en bare kunne ha ett globalt alfabet.

c) Hva kalles det når metoder i samme klasse (som **`convert`**-metodene) har samme navn? Hva er regelen for å skille dem fra hverandre/velge mellom dem, når de brukes/kalles?

At metoder i en klasse kan ha samme navn kalles ”overloading”. For å avgjøre hvilken som skal kalles, brukes de deklarerte typene til argumentene (ikke returverdien). Merk at dette er noe annet enn polymorfi, som handler om at subklasser kan ha ulike implementasjoner av metoder definert i en felles superklasse.

d) Utvid klassen med metodene **`setWord(String)`** som endrer ordet som brukes for en bestemt bokstav (første bokstav i ordet), og **`removeWord(char)`** som fjerner ordet for en bestemt bokstav fra alfabetet. Merk at **`setWord`** skal kun kunne endre ordet for en bokstav som allerede finnes i alfabetet.

Her brukes pos-metoden som også ble brukt av convert(char). Merk at setWord ikke skal legge inn et nytt ord, kun erstatte et eksisterende.

```
	public void setWord(String word) {
		int pos = pos(word.charAt(0));
		if (pos >= 0) {
			alphabet.set(pos, word);
		}
	}
```

```
	public void removeWord(char c) {
		int pos = pos(c);
		if (pos >= 0) {
			alphabet.remove(pos);
		}
	}
```

Del 4 – Testing (10%)

a) Skriv en JUnit-testklasse som tester begge **`convert`**-metodene i **`RadioAlphabet`**-klassen. Du kan anta at konstruktøren virker. Vi er ikke så nøye på detaljene, bare den generelle testeteknikken er riktig.

Det sentrale her er å rigge opp et test-objekt og å bruke relevante testdata, inkludert bokstaver som ikke har noen oversettelse og ikke-bokstaver. Som nevnt er vi ikke opptatt at JUnit-spesifikke detaljer, men dersom en ikke bruker JUnit-metoder, så bør en selv definere egne assert-aktige hjelpemetoder.

```
	private RadioAlphabet radioAlphabet;

	@Override
	protected void setUp() throws Exception {
		// TODO Auto-generated method stub
		super.setUp();
		radioAlphabet = new RadioAlphabet(alphabet, "-");
	}

	public void testConvertChar() {
		assertEquals("alfa", radioAlphabet.convert('a'));
		assertEquals("hotel", radioAlphabet.convert('h'));
		assertEquals("zulu", radioAlphabet.convert('z'));
		assertNull(radioAlphabet.convert('!'));
	}

	public void testConvertString() {
		assertEquals("hotel alfa lima lima victor alfa romeo delta",
radioAlphabet.convert("hallvard"));
		assertEquals("hotel alfa lima alfa lima",
radioAlphabet.convert("!h!a!l!a!l?"));
		radioAlphabet.setWord("åring");
		assertEquals("hotel victor alfa romeo delta",
radioAlphabet.convert("håvard"));
	}
```

b) Forklar hvordan du vil teste **`setWord`**- og **`removeWord`**-metodene i **`RadioAlphabet`**-klassen vha. JUnit-teknikken. Hva er en vesentlig (kompliserende) forskjell mellom å teste disse metodene og å teste **`convert`**-metodene?

Det vesentlige her er at en må teste oppførselen både før og etter bruken av endringsmetoden, i praksis at convert fungerer ulikt før og etter bruk av setWord og removeWord. Det kan synes som om det kun er nødvendig å teste oppførselen etterpå, fordi en tross alt har andre tester for oppførsel også. Men det er riktig å test både før og etter, for at testen skal dekke tilfellet alene.

```
	public void testSetWord() {
		assertEquals("hotel alfa lima lima victor alfa romeo delta",
radioAlphabet.convert("hallvard"));
		radioAlphabet.setWord("lala");
		assertEquals("hotel alfa lala lala victor alfa romeo delta",
radioAlphabet.convert("hallvard"));
	}

	public void testRemoveWord() {
		assertEquals("hotel alfa lima lima victor alfa romeo delta",
radioAlphabet.convert("hallvard"));
		radioAlphabet.removeWord('l');
		assertEquals("hotel alfa victor alfa romeo delta",
radioAlphabet.convert("hallvard"));
	}
```

Del 5 – Arv og observerbarhet (25%)

a) Morse-alfabetet er et alfabet hvor bokstaver (og andre tegn) representeres med sekvenser av prikker og streker. F.eks. skrives det internasjonale nødsignalet ”SOS” som ”... --- ...” fordi ’s’ skrives som ”...” og ’o’ som ”---”. Du skal lage en subklasse av **`RadioAlphabet`**-klassen kalt **`MorseAlphabet`**, som støtter dette alfabetet. F.eks. skal **`new MorseAlphabet()`** gi et ferdig-initialisert objekt som kan konvertere **`”sos”`** til **`”... --- ...”`** og **`”test”`** til **`”- . ... -"`**. **`convert`**-metodene og **`removeWord`** skal virke som spesifisert i del 3. **`setWord`**-metoden må nødvendigvis endres slik at den tar inn tegnet som første argument og ordet som andre.

Det er et mål å utnytte arv slik at det blir mest mulig gjenbruk av kode, f.eks. ved at en eller flere av de arvede metodene ikke trenger å redefineres. Forklar med tekst og kode hvordan du vil gjøre dette. Forklar om nødvendig hvordan du må endre på **`RadioAlphabet`**-klassen for å få det til, men merk at den må fortsatt virke slik det er beskrevet i del 3.

Besvarelser vurderes ut fra korrekthet og grad av gjenbruk. Noen løsninger på del 3, vil egne seg bedre til gjenbruk enn andre, så vi må delvis vurdere ut fra det vi oppfatter som potensialet for gjenbruk. Det teller positivt at en reflekterer over om en kunne økt graden av gjenbruk, dersom en hadde skrevet om RadioAlphabet slik og slik, selv om en ikke har hatt tid til å gjøre det.

Generelt så er det best om en kan gjenbruke logikk som allerede er public. protected-metoder foretrekkes fremfor protected-felt.

Med vår løsning blir det lite ekstra kode, siden den eksisterende representasjonen med forbokstav og ord i ett kan også brukes for morsealfabetet. Hvert element i lista vil være tegnet foran prikkene og strekene, f.eks. ”s...” og ”o---”. Trikset er å utnytte superklassens metoder, ved å justere på argument/returverdi. convert(char) kaller super-metoden og fjerner første bokstav. Den nye setWord-metoden kaller superklassens setWord-metode med tegnet lagt foran ordet. Convert(String) og remove-metoden arves i sin helhet.

```
public class MorseAlphabet extends RadioAlphabet {

	private final static String morseAlphabet = "s... o--- e. t-";

	public MorseAlphabet() {
		super(morseAlphabet, " ");
```

```
	}

	public String convert(char c) {
		String word = super.convert(c);
		return (word != null ? word.substring(1) : null);
	}

	public void setWord(char c, String word) {
		super.setWord(c + word);
	}
}
```

b) Du skal gjøre **`RadioAlphabet`**-klassen (og dermed også subklassen) såkalt ”observerbar”, vha. følgende lyttergrensesnitt:

```
public interface RadioAlphabetListener {
   public void radioAlphabetChanged(RadioAlphabet, char);
}
```

Dersom en **`RadioAlphabet`**-instans endres, skal altså metoden **`radioAlphabetChanged`** kalles på alle lytterne, med **`RadioAlphabet`**-instansen som første argument og tegnet viss’ ord ble endret/fjernet som det andre. Beskriv hvordan **`RadioAlphabet`** og evt. **`MorseAlphabet`** må endres for å implementere denne oppførselen.

Observerbarhet krever
1) at en holder styr på lytterne og
2) at alle endringsmetoder kaller lytternes lyttermetode.

Med vår løsning trenger en faktisk ikke å endre MorseAlphabet, siden metodene for 1) arves og all endring 2) skjer gjennom arvede metoder.

1) Holder styr på lytterne

```
	private List<RadioAlphabetListener> listeners = new
ArrayList<RadioAlphabetListener>();

	public void addRadioAlphabetListener(RadioAlphabetListener listener) {
		listeners.add(listener);
	}

	public void removeRadioAlphabetListener(RadioAlphabetListener listener) {
		listeners.remove(listener);
	}
```

2) Alle endringsmetoder kaller lytternes lyttermetode:

```
	protected void fireRadioAlphabet(char c) {
		for (RadioAlphabetListener listener : listeners) {
			listener.radioAlphabetChanged(this, c);
		}
	}

	public void setWord(String word) {
		int pos = pos(word.charAt(0));
```

```
            if (pos >= 0) {
                  alphabet.set(pos, word);
                  // added
                  fireRadioAlphabet(word.charAt(0));
            }
      }

      public void removeWord(char c) {
            int pos = pos(c);
            if (pos >= 0) {
                  alphabet.remove(pos);
                  // added
                  fireRadioAlphabet(c);
            }
      }
```