NTNU
Det skapende universitet

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100
Objektorientert programmering med Java

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97263

Eksamensdato: 2013, fredag 16. august

Eksamenstid (fra-til): kl. 09:00-13:00

Hjelpemiddelkode/Tillatte hjelpemidler: C – Kalkulator og en spesifisert bok:

Kun «Big Java» av Cay S. Horstmann, utgitt av Wiley, er tillatt.

Blanke lapper for å finne fram til riktig side, gul/rosa markering av ord, og kommentarer av typen «NB», «OBS», «Les Dette» etc. er ok, men andre skrevne notater er ikke tillatt, og må fjernes fra boken før eksamen starter.

Annen informasjon:

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.

Målform/språk: Bokmål

Antall sider: 7

Antall sider vedlegg: 0

Kontrollert av:

20/8 2013 _____

Dato *Hallvard Trætteberg.*

Introduksjon for hele oppgavesettet, basert på Wikipedia

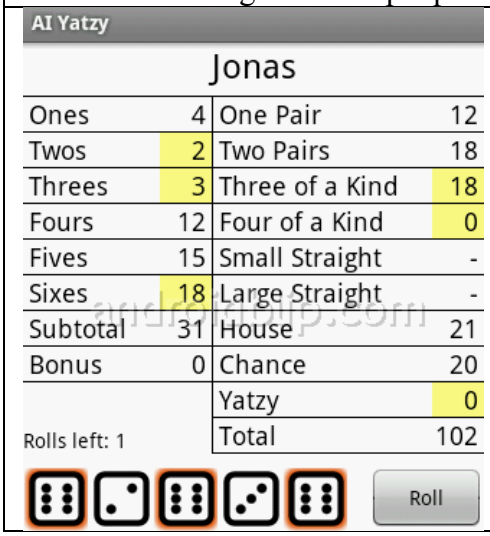
Årets oppgavesett tar utgangspunkt i terningspillet Yatzy. Meningen med spillet er å oppnå en høyest mulig poengsum. I standardversjonen av Yatzy tildeles poeng for å oppnå 15 forskjellige typer terningkombinasjoner ved å kaste 5 terninger (med 6 sider). Målet er å klare alle typene i løpet av ett spill med flest mulig poeng i hver kombinasjon. I hver runde får hver spiller inntil tre kast for å oppnå én av disse typene. Mellom hvert av de tre kastene kan spilleren velge å legge til side (spare) et antall av terningene og kaste resten (eller ingen) om igjen. Dersom en f.eks. får 1-1-2-6-6 i første kast, kan en spare sekserne og kaste de tre andre, for å prøve å få så mange seksere som mulig. Etter kastene velger en hvilken type kombinasjon det skal registreres poeng på. Har en fått tre seksere kan en f.eks. registrere 18 poeng på "seksere" eller "tre like" dersom disse er "ledige". En kan nemlig bare registrere poeng på en kombinasjonstype én gang, og kan derfor ikke forbedre poengene for en type senere i spillet. Dersom en etter tre kast ikke klarer å oppnå en gjenværende kombinasjonstype, må en likevel velge en type som da får verdien 0. Derfor er det om å gjøre å satse på (gjenværende) kombinasjonstyper som utnytter terning(kast)ene godt.

Standardversjonen av Yatzy har følgende kombinasjonstyper:

- Så mange som mulig av en bestemt verdi (enere, toere, … , seksere), altså en kombinasjonstype pr. mulig verdi
- Ett par – (minst) to like terninger
- To par – (minst) to like terninger og to like terninger *av en annen* verdi
- Tre og fire like – henholdsvis (minst) tre og fire like terninger
- Liten og stor straight – Henholdsvis kombinasjonen 1-2-3-4-5 og kombinasjonen 2-3-4-5-6
- Hus – tre like terninger og to like terninger *av en annen* verdi
- Sjanse – vilkårlig kombinasjon
- Yatzy – fem like terninger, gir alltid 50 poeng

For alle kombinasjoner utenom Yatzy er poengsummen like summen av øynene på terningene som inngår i kombinasjonen. Dersom en kombinasjonstype forekommer flere ganger i en kombinasjon, er det alltid den mest verdifulle som telles. For eksempel vil kombinasjonen 5-5-4-4-2 gi 10 poeng som "ett par", siden et femmer-par er mer verdt enn et firer-par, eller 18 poeng som "to par".

Figur: Eksempel på Yatzy-app, med terninger, kombinasjoner og poeng



I denne Yatzy-app'en er de ulike kombinasjonene og poengene vist i to kolonner, med kombinasjonene for hver av de ulike verdiene, summen av dem og bonusen vist til venstre, og de andre og totalsummen vist til høyre. Jonas har allerede fått poeng for kombinasjonene enere, firere, femmere, ett par, to par, hus og sjanse. Han har også vært nødt til å nulle ut liten og stor "straight". Delsummen på 31 for enere, firere og femmere er ennå for liten til å få bonus, siden det krever 63. I inneværende runde har han kastet 6-2-6-3-6 og tabellen indikerer med gult at dette gir 2 poeng dersom det settes på toere, 3 poeng på treere, 18 på seksere og tre like og 0 på fire like og Yatzy. Jonas ser ut til å ha bestemt seg for å beholde de tre sekserne og kaste terningene som viser to og tre.

Totalsummen er summen av alle poengene for de ulike kombinasjonene pluss en bonus du kan oppnå dersom du får minst 63 poeng på kombinasjonene for hver av de seks verdiene. Dette tilsvarer at du får minst tre av den bestemte verdien i gjennomsnitt, altså 3 enere som settes på kombinasjonen "enere", 3 toere som settes på kombinasjonen "toere" osv.

For alle oppgavene gjelder det at du kan (og bør) gjenbruke andre metoder vi har deklarert i samme eller tidligere deloppgaver, selv om du ikke har implementert dem (riktig). Prøv også å unngå duplisering av kode og definer gjerne hjelpemetoder for å gjøre koden ryddigere.

Del 1 – Dice-klassen (35 %)

Klassen **Dice** representerer (alle) terningene som kastes i et og samme spill og har metoder for rulle alle eller bestemte terninger og for å telle/summere for ulike typer kombinasjoner.

a) Selv om standardterningene har seks sider, ønsker vi å unngå å "hardkode" dette overalt i programmet. Hvordan er det vanlig å kode (med) slike "konstanter", så en kun trenger å endre verdien ett sted?

```
public static final int MAX_DIE_VALUE = 6;
```

static- og final-modifikatorene brukes for å deklarere konstanter. Disse kan så refereres til med klassenavn.feltnavn, i dette tilfellet **Dice.MAX_DIE_VALUE**

b) Implementer felt og konstruktør(er) for representere og initialisere terningverdiene. Du må ta høyde for at antall terninger kan varieres fra spill til spill.

```
private final int[] dice;

public Dice(int diceCount) {
	dice = new int[diceCount];
}
```

c) Implementer **String toString()**-metoden, som returnerer en String med verdien på alle terningene, med en bindestrek (-) *mellom* hver verdi. Rekkefølgen har ingen betydning.

```
public String toString() {
	String result = "";
	for (int value : dice) {
		if (result.length() > 0) {
			result += "-";
		}
		result += value;
	}
	return result;
}
```

d) Implementer metoden **int getValueCount(int value)**, som returnerer antall terninger som viser den angitte verdien. Dersom terningene f.eks. viser 6-4-1-4-2 og metoden kalles med 4 som argument, så skal metoden returnere 2.

```
public int getValueCount(int value) {
	int count = 0;
	for (int v : dice) {
		if (v == value) {
```

```
                count++;
            }
        }
        return count;
}
```

e) Implementer metoden **int getHighestValueOfSame(int count, int butNot)**. Denne returnerer den høyeste verdien som finnes på minst **count** terninger, og som ikke er **butNot**, eller 0 hvis det ikke finnes minst **count** antall av noen verdi. Anta at terningene viser 1-3-1-6-3. Dersom metoden kalles med **count** lik 2 og **butNot** lik 0 som argumenter, så skal den returnere 3 fordi 3 er den høyeste verdien det er (minst) to av. Dersom den kalles med 2 og 3 som argumenter, så skal den returnere 1 fordi 1 er den høyeste (og eneste) verdien det er to av *og som ikke er 3*. Og dersom den kalles med 3 og 0 som argumenter, så skal 0 returneres, fordi det ikke er 3 terninger med samme verdi.

```
public int getHighestValueOfSame(int count, int butNot) {
        for (int value = ValuesDice.MAX_DIE_VALUE; value >= 1; value--) {
            if (value != butNot && getValueCount(value) >= count) {
                return value;
            }
        }
        return 0;
}
```

f) Implementer metoden **int getStraightSum(int startValue, int endValue)**. Denne returnerer summen av verdiene fra og med **startValue** og til og med **endValue**, men bare dersom det er minst én terning med hver verdi. Anta at terningene viser 2-1-3-5-3. Dersom metoden kalles med startValue lik 1 og endValue lik 3, så skal den returnere 6 fordi det er summen av 1, 2 og 3 og alle disse finnes blant terningene. Dersom metoden kalles med 1 og 5, så skal den returnere 0, fordi ingen terning viser 4.

```
public int getStraightSum(int startValue, int endValue) {
        int sum = 0;
        for (int value = startValue; value <= endValue; value++) {
            if (getValueCount(value) < 1) {
                return 0;
            }
            sum += value;
        }
        return sum;
}
```

g) Implementer de to metodene **void roll()** og **void roll(int[] values)**, som begge tilordner nye, tilfeldige verdier til terningene. Den første gir ny verdi til alle terningene, mens den andre kun gir ny verdi til terningene med verdier angitt av **values**-tabellen. Merk at én verdi tilsvarer kun én terning, selv om det er flere med samme verdi, så for å slå om igjen to firere må **values**-tabellen inneholde to 4ere. Metoden skal utløse et passende unntak, dersom en verdi i **values** ikke har en tilsvarende terning.

```
private Random random = new Random();

private int getRandomDieValue() {
        return random.nextInt(MAX_DIE_VALUE) + 1;
}

private void roll(boolean[] indices) {
```

```
	for (int die = 0; die < dice.length; die++) {
		if (indices == null || indices[die]) {
			dice[die] = getRandomDieValue();
		}
	}
}

public void roll(int[] values) {
	boolean[] indices = new boolean[dice.length];
	for (int value : values) {
		int die = 0;
		while (die < dice.length) {
			if ((! indices[die]) && value == dice[die]) {
				break;
			}
			die++;
		}
		if (die < dice.length) {
			indices[die] = true;
		} else {
			throw new IllegalArgumentException("No die with value " +
value);
		}
	}
	roll(indices);
}

public void roll() {
	roll((boolean[]) null);
}
```

Del 2 – ScoreCard-klassen (30 %)

Klassen **ScoreCard** skal holde orden på alle poengene til én spiller, altså informasjon tilsvarende tabellen vist i figuren i introduksjonen. En **ScoreCard**-instans skal altså kunne *lagre* ett tall pr. terningkombinasjon f.eks. 4 for "enere", 18 for "to par", nulling av "liten straight" og (foreløpig) ingenting/tomt for yatzy. I tillegg skal den kunne beregne og evt. sette poengene en får for terningene for en bestemt kombinasjon.

a) Forklar hvordan du vil lagre poeng-dataene i **ScoreCard**-klassen. Hint: Det kan være nyttig med en **enum**-klasse som representerer hver av de 15 terningskombinasjonene og **ordinal()**-metoden for indeksering i en tabell. Lag en passende *konstruktør*, slik at dataene er riktig initialisert, samt metoden **getScore** for å hente ut (lese) poeng knyttet til en bestemt kombinasjon. Velg selv passende parametre og returtype.

```
private Integer[] scores;

public ScoreCard() {
	this.scores = new Integer[Combinations.values().length];
}

enum Combinations {
		ONES, TWOS, THREES, FOURS, FIVES, SIXES, PAIR, TWO_PAIRS,
THREE_OF_A_KIND, FOUR_OF_A_KIND, SMALL_STRAIGHT, LARGE_STRAIGHT, FULL_HOUSE,
CHANCE, YATZY;
}
```

b) Implementer to metoder, **getScore** og **setScore**, den første for å beregne hvor mange poeng en får for å registrere terningene (**Dice**-instans) på en bestemt terningkombinasjon og den andre for å faktisk registrere terningene på en bestemt kombinasjon. Merk at disse metoden må håndtere tilfellet hvor en får 0 poeng, altså nuller ut kombinasjonen. Velg selv passende parametre og returtype.

```
public int getScore(Dice dice, Combinations combination) {
	switch (combination) {
		case PAIR:		return dice.getHighestValueOfSame(2, 0) * 2;
		case TWO_PAIRS: return getDualCountScore(2, 2);
		case THREE_OF_A_KIND:	return dice.getHighestValueOfSame(3, 0) * 3;
		case FOUR_OF_A_KIND:	return dice.getHighestValueOfSame(4, 0) * 4;
		case SMALL_STRAIGHT: return dice.getStraightSum(1, 5);
		case LARGE_STRAIGHT: return dice.getStraightSum(2, 6);
		case FULL_HOUSE: return getDualCountScore(3, 2);
		case CHANCE: return dice.getSum();
		case YATZY: return dice.getHighestValueOfSame(5, 0) > 0 ? 50 : 0;
		default: {
			int value = combination.ordinal() + 1;
			return dice.getValueCount(value) * value;
		}
	}
}

// helper method, used by getScore for the two pairs and full house cases
private int getDualCountScore(int count1, int count2) {
	int value1 = getHighestValueOfSame(count1, 0);
	int value2 = getHighestValueOfSame(count2, value1);
	return (value1 > 0 && value2 > 0 ? value1 * count1 + value2 * count2 : 0);
}

public void setScore(Dice dice, Combinations combination) {
	int score = getScore(dice, combination);
	scores[combination.ordinal()] = score;
}
```

Del 3 – Observerbarhet (10 %)

a) Hva vil det si at et objekt er ”observerbart”? Hvorfor er dette nyttig?

Et observerbart objekt har mekanismer for å si fra til såkalte observatører om alle relevante endringer av tilstanden sin. Dette er nyttig for å holde tilstanden til avhengige objekter konsistent, f.eks. at et brukergrensesnitt viser den faktiske tilstanden til en objekter i en applikasjon.

b) Forklar med tekst og evt. kode hvordan du vil gjøre **ScoreCard** observerbar.

Det defineres et lyttergrensesnitt, f.eks. **ScoreCardListener** med en metode **scoreCardChanged**, som observatørene/lytterne må implementere. **ScoreCard** utvides til å holde en liste av slike, med metoder for å legge til (**addScoreCardListener**) og fjerne (**removeScoreCardListener**) lyttere. Når tilstanden til **ScoreCard**-instansen endres i **setScore**-metoden, går en gjennom **ScoreCardListener**-lista og kaller **scoreCardChanged**-metoden med argumenter som angir hva som ble endret.

Del 4 – Testing (15 %)

a) Med kun **get**- og **roll**-metodene som er spesifisert i **Dice**-klassen i oppgave 1, er det vanskelig å gjennomføre testing av klassen, hvorfor? Hvordan vil du utvide/endre på koden for å gjøre det mulig å teste disse metodene?

Problemet er at det ikke er mulig å sette terningverdiene direkte, siden de bare settes (indirekte) til uforutsigbare verdier i roll-metodene. En kan derfor ikke veksle mellom å sette tilstanden og teste hvilke svar get-metodene gir. En løsning er å lage en egen setDice-metode og en annen er å gjøre det mulig å sette objektet (med set-metode eller i konstruktøren) som genererer "tilfeldige" tall, slik at vi kan gi inn en som genererer forutsigbare verdier.

b) Forklar med tekst og evt. kode hvordan du vil teste **getScore**- og **setScore**-metoden i **ScoreCard** –klassen slik de er spesifisert i oppgave 2 b). Det viktigste er å få frem den generelle teknikken for testing, ikke å bruke et spesifikk rammeverk som JUnit.

getScore-metoden er ikke avhengig av tilstanden til en **ScoreCard**-instans, og kan testes ved å kalle metoden med varierende og dekkende argumenter og sjekke returverdien mot "fasiten". **setScore**-metoden har som formål å endre tilstanden til en **ScoreCard**-instans og kan bare sjekkes "indirekte" ved at en først endrer tilstanden ved å kalle den og så sjekker returverdien til en get-metode som leser ut samme tilstand (i dette tilfellet **getScore**-metoden fra oppgave 2a)).

Del 5 – Arv og UML (10 %)

De ulike terningkombinasjonene kan klassifiseres i grove kategorier. F.eks. kan enere, toere, treere osv. kategoriseres som "så mange som mulig av en bestemt verdi", ett par, tre like, fire like og yatzy kan kategorisers som "(minst) et bestemt antall av en vilkårlig verdi" og liten og stor straight har opplagte likheter.

a) I lys av dette, forklar hvordan arv kan brukes for å strukturere koden for å beregne poeng.

Det defineres et grensesnitt med én metode for å beregne poeng for terninger satt på en gitt kombinasjon, tilsvarende getScore-metoden fra oppgave 2 b). Deretter defineres én klasse pr. kategori som implementerer dette grensesnittet og implementerer getScore-metoden iht. poengreglene for denne kategorien.

b) Utvid kategoriseringen av terningkombinasjonene til å dekke alle typene og tegn et UML-diagram som dokumenterer den tilsvarende kodestrukturen.

Kategorier:
- så mange som mulig av en bestemt verdi – summerer verdien av alle terningene med denne verdien
- (minst) et bestemt antall av en vilkårlig verdi – finner høyeste verdi av dette antallet og summerer
-- yatzy – trenger en egen subklasse, siden regelen med 50 poeng ikke følger den generelle regelen
- (minst) et bestemt antall av én verdi og (minst) et bestemt antall av en annen verdi (to par og hus)
- straight – (minst) en av hvert tall i et intervall
- sjanse –vilkårlige terningverdier