NTNU
Kunnskap for en bedre verden

KANDIDAT
10748

PRØVE
TDT4100 1 Objektorientert programmering

Emnekode	TDT4100
Vurderingsform	Skriftlig eksamen
Starttid	06.06.2018 07:00
Sluttid	06.06.2018 11:00
Sensurfrist	27.06.2018 00:00
PDF opprettet	05.08.2020 11:47

Oppgavetekst

Oppgave	Tittel	Oppgavetype
i	Front page	Dokument
i	Standard Java classes and methods	Dokument
i	Farkle	Dokument

Dice class

Oppgave	Tittel	Oppgavetype
1	Fields and constructors	Programmering
2	Dice as text	Programmering
3	Dice values	Programmering
4	Score	Programmering
5	Iterable	Programmering
6	add and remove	Programmering

Implementation and use of interface

Oppgave	Tittel	Oppgavetype
7	Three or more of a kind	Programmering
8	Total score computation	Programmering
9	Summing with the Stream technique	Programmering

Interface, inheritance and delegation

Oppgave	Tittel	Oppgavetype
10	The Dice interface	Flervalg
11	Inheritance	Programmering
12	Delegation	Programmering

Farkle round

Oppgave	Tittel	Oppgavetype
13	final	Flervalg (flere svar)
14	Code in bank	Fyll inn tekst
15	Exception classes	Flervalg
16	Code in keepAndRoll	Langsvar
17	FarkleRound object diagram	Langsvar

Functional interfaces and testing

Oppgave	Tittel	Oppgavetype
18	Functional interface?	Langsvar
19	Alternative Dice constructor	Programmering
20	Testing of the Dice class	Programmering

JavaFX controller and FXML

Oppgave	Tittel	Oppgavetype
21	JavaFX controller	Programmering

Denne delen av oppgaven omhandler **Dice**-klassen, som brukes til å representere (verdien av) én eller flere terninger (engelsk: dice = terninger, die = terning), og en mulig poengverdi (score) for terningene. En slik klasse kan være nyttig i mange typer terningspill. I f.eks. Yatzy kan en Dice-instans brukes til både å representere et helt kast før det er gitt poeng, og terningene en ender opp med etter en runde, med poengene satt.

Dice-klassen er vist under, ... erstatter kodefragmenter som det spørres om i oppgaveteksten. For oversiktens skyld er det også oppgitt et grensesnitt som brukes i senere oppgaver.

```
/**
* Represents a set of die values. A die has six possible values 1-6,
* but the number of dice may vary from Dice instance to Dice instance.
* In addition, a Dice-instance can have a score.
*/
public class Dice implements Iterable<Integer> {

/**
* @param dieCount
* @return a collection of random integer values in the range 1-6
*/
public static Collection<Integer> randomDieValues(int dieCount) {
   ...
}

... fields (part 1) ...

/** (part 1)
* Initializes this Dice with the values in dieValues, and a score.
* @param dieValues
* @param score the score to set, may be -1 for yet unknown
* @throws a suitable exception if the die values are outside the valid range
*/
public Dice(Collection<Integer> dieValues, int score) {
   ...
}

/** (part 1)
* Initializes this Dice with dieCount random values (using Math.random())
* @param dieCount
*/
public Dice(int dieCount) {
   ...
}

/** (part 1)
* Initializes this Dice with the values in dice, and a score
* @param dieValues
* @param score the score to set, may be -1 for yet unknown
*/
public Dice(Dice dice, int score) {
   ...
}

/** (part 2)
* Format: [die1, die2, ...] = score (score is omitted when < 0)
*/
... method for generating a String representation of a Dice instance ...

/** (part 2)
* Parses a string using the toString format (see above) and
* returns a corresponding Dice.
```

```
     * @param s
     * @return a new Dice instance initialized with die values and score from the String argument
     */
     public static Dice valueOf(String s) {
        ...
     }

     /** (part 3)
     * @return the number of die values
     */
     public int getDieCount() {
        ...
     }

     /** (part 3)
     * @param dieNum
     * @return the value of die number dieNum
     */
     public int getDieValue(int dieNum) {
        ...
     }

     /** (part 3)
     * @param value
     * @return the number of dice with the provided value
     */
     public int getValueCount(int value) {
        ...
     }

     /** (part 4)
     * @return the current score
     */
     public int getScore() {
        ...
     }

     /** (part 4)
     * Sets the score, but only if it isn't already set to a non-negative value
     * @param score
     * @throws a suitable exception if score already is set to a non-negative value
     */
     public void setScore(int score) {
        ...
     }

     ... Iterable methods (part 5) ...

     /** (part 6)
     * @param dice
     * @return true if all die values in the argument appear in this Dice
     */
     public boolean contains(Dice dice) {
        ...
     }

     /** (part 6)
     * @param dices a Collection of Dice
     * @return a new Dice instance with the all the die values this Dice and
```

```
     * all Dice in the argument, without any specific order
     */
     public Dice add(Dice dice) {
       ...
     }

     /** (part 6)
     * @param dice
     * @return a new Dice instance with the die values from this Dice, but
     * without those from the argument, without any specific order
     */
     public Dice remove(Dice dice) {
       ...
     }
}

/** (part 7)
* Interface for scoring rules, i.e.
* logic for computing a score for a subset of dice in a Dice
*/
public interface DiceScorer {
  /**
   * Computes a score for (a subset of) the dice in the Dice argument.
   * The return value includes those dice that gives the score, and
   * of course the score itself.
   * @param dice
   * @return The dice for which the rule computes a score, and the score itself, or
   * null, if this rule isn't applicable
   */
   Dice getScore(Dice dice);
}
```

1 Fields and constructors

En **Dice**-instans skal ha informasjon om (verdien til) et *visst antall terninger* og en *poengverdi* (score). Terningverdiene settes ved initialisering (på ulike måter), mens poengverdien kan settes ved initialisering eller senere.

Skriv kode for felt og konstruktører, samt metoden **randomDieValues** iht. API-beskrivelsen. Bruk **random**-metoden i **Math**-klassen, som er beskrevet som følger:

public static double random()

Returns a **double** value with a positive sign, greater than or equal to **0.0** and less than **1.0**. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.

Skriv ditt svar her...

```
Import math.random;

public class Dice implements Iterable<Integer> {
    Private Collection<Integer> diceList;
    Private int score;

    public Dice(int dieCount) {
        diceList = randomValues(dieCount);
        score = -1;
    }

    Public Dice(Collection<Integer> dieValues, int score){
        if (dieValues.stream().anyMatch(p->p<1 || p>6)){
            throw new IllegalArgumentException;
        }
        this.score=score;
        diceList = new ArrayList();
        diceList.addAll(dieValues);
    }

    public static Collection<Integer> randomDieValues(int dieCount) {
        Collection<Integer> diceList = new ArrayList()
        Math math = new Math();
        for (int k=0; k<amountOfDice; k++){
            die = Math.random() * 6;
            diceList.add(die.roundUp());
        }
        return diceList;
}
```

Maks poeng: 6

2 Dice as text

Skriv *standardmetoden* som brukes for å lage en tekstlig representasjon av et **Dice**-objekt på formatet som er angitt i API-beskrivelsen. Skriv også **valueOf**-metoden, som brukes for å lage et nytt **Dice**-objekt med spesifikke terningverdier og evt. en poengverdi fra en **String** på samme format.

Formatet er "**[** *t1*, *t2*, ... *tn* **] =** *poeng"*, hvor *t1* - *tn* er terningverdier og *poeng* er poengverdien. Poeng-delen, altså " **=** *poeng*" skal ikke være med hvis poeng er -1 (ennå ikke satt).

Eksempel: Hvis et **Dice**-objekt har terningverdiene 1, 1 og 3 og poengene ikke er satt (= -1), så vil den tekstlige representasjonen være "**[1, 1, 3]"**. Dersom terningene er tre 6-ere som har gitt 600 poeng, så vil teksten være " **[6, 6, 6] = 600"**.

Skriv ditt svar her...

```
1
2     public String toString(){
3         String string = ""
4         string += dicelist.toString;
5         if (score<0){
6             string += " = " + String.valueOf(score); //tror ikke value of trengs her, men tas med for
                  sikkerhetsskyld.
7         }
8         return string;
9     }
10
11    public static Dice valueOf(String s){
12        Collection<Integer> dice = new ArrayList();
13        String[] sl= s.split(" = ");
14        Int score = Integer.valueOf(sl[1])
15        String s1 = sl[1].strip('[',']').split(", ")
16        for (int k = 0; k<s1.length(); k++){
17            dice.add(s1.charAt(k) - '0');
18        }
19        return Dice(dice, score);
20    }
```

Maks poeng: 8

3 Dice values

Skriv metodene **getDieCount**, **getDieValue** og **getValueCount** iht. API-beskrivelsen.

Skriv ditt svar her...

```
1  public int getDieCount(){
2      return this.diceList.size();
3  }
4
5  public int getDieValue(int dieNum) {
6      return diceList.get(dieNum);
7  }
8
9  public int getValueCount(int value) {
10     return diceList.stream().filter(p->p==value).count;
11 }
```

Maks poeng: 5

4 Score

Skriv metodene **getScore** og **setScore** iht. API-beskrivelsen.
Skriv ditt svar her...

```java
public int getScore() {
    return score;
}

public void setScore(int score) {
    if (this.score < 0){
        throw new IllegalArgumentException();
    }
    this.score=score;
}
```

Maks poeng: 3

5 Iterable

Dice-klassen implementerer **Iterable<Integer>**-grensesnittet. Implementer metoden(e) som da er nødvendig. Skriv også kode som eksemplifiserer hvordan man ved bruk av **Dice** kan dra nytte av at den implementerer nettopp dette grensesnittet.
Skriv ditt svar her...

```java
public boolean hasNext(){
    return diceList.hasNext();

public int next(){
    return dicelist.next();
}

//Ettersom DiceList er en Collection som implementerer Iterable,vil den ha hasNext() og next() som
    metoder i deg.
Vi kan bruke det i f.eks:

public void sumValues(Collection<Integer> coll){
    Int sum = 0;
    sum+= getValue;
    while (coll.hasNext()){
        next();
        sum+= getValue();
    }
    return sum;
}

//Dette er ikke en fullstendig eller kjørbar kode, men er et eksempel på hvordan det kan
    implementeres.
```

Maks poeng: 3

6 add and remove

Skriv metodene **add** og **remove**, som alle tar et **Dice**-objekt som eneste argument. Merk at ingen av disse endrer på verken **this**-objektet eller argumentet, og poengverdien(e) benyttes ikke.

Her er noen eksempler på bruken av disse metodene, hvor tekst-formatet i API-beskrivelsen brukes for å representere **Dice**-objekter:

[1, 2].add(**[1, 4]**) returnerer **[1, 2, 1, 4]** // merk at rekkefølgen ikke spiller noen rolle

[1, 1, 2].remove(**[1, 4]**) gir **[1, 2]** // merk at rekkefølgen ikke spiller noen rolle

Merk at **remove** *ikke* har samme logikk som **Collection** sin **removeAll**-metode.

Skriv ditt svar her...

```
public Collection<Integer> getDiceList(){
    return thisdiceList;
}

public Dice add(Dice dice){
    Integer = this.score + dice.getScore;
    Collection<Integer> thisdielist= new ArrayList();
    thisdielist.addAll(dice.getDicelist);
    thisdielist.addAll(this.diceList);
    Dice thisdie = New Dice(thisdielist, score);
    return thisdie;
}

public Dice remove(Dice dice){
    Collection<Integer> paramDiceList = dice.getDiceList;
    Collection<Integer> dL = this.dicelist;
    paramDiceList.stream().forEach(p->dL.remove(p));
    newScore = this.score - dice.getScore;
    return new Dice(dL, newScore);
}
```

Maks poeng: 8

Denne deloppgaven handler om poengberegning basert på et sett terningverdier. Felles for mange terningspill er at man kaster terninger og så finner ut hvor mange poeng en får basert på terningverdiene. Visse kombinasjoner gir mer elle mindre poeng, og noen gir ingen. I Yatzy har en mange poenggivende kombinasjoner, f.eks. ett par (to like), to par, tre og fire like, liten (1-5) og stor (2-6) straight, hus (ett par og tre like) og Yatzy, og poengene en får er stort sett summen av terningverdiene som inngår i kombinasjonen. I Farkle, derimot, så får en 100 poeng for enere, 50 poeng for femmere, og for de andre får en bare poeng hvis en får tre eller flere i slengen (se vedlegg for detaljene).

For å støtte (minst) begge disse spillene, så introduseres et **DiceScorer**-grensesnitt, som representerer en poengregel generelt, og én implementasjonsklasse for hver (type) regel. **DiceScorer**-grensesnittet har kun én metode, **getScore**, som tar inn et **Dice**-objekt med alle terningverdiene som skal vurderes samlet og returnerer et nytt **Dice**-objekt med akkurat de terningverdiene som dekkes av regelen og med poengverdien satt riktig for akkurat disse terningverdiene:

```
/**
* Interface for scoring rules, i.e.
* logic for computing a score for a subset of dice in a Dice
*/
public interface DiceScorer {
  /**
   * Computes a score for (a subset of) the dice in the Dice argument.
   * The return value includes those dice that gives the score, and
   * of course the score itself.
   * @param dice
   * @return The dice for which the rule computes a score, and the score itself, or
   * null, if this rule isn't applicable
   */
  Dice getScore(Dice dice);
}
```

En implementasjon av **DiceScorer** som tilsvarer "tre like"-regelen i Yatzy, vil altså returnere et **Dice**-objekt med de tre like terningverdiene den finner i **Dice**-argumentet, eller null hvis den ikke finner tre like verdier. I tillegg settes poengene i **Dice**-objektet som returneres til riktig verdi. Hvis f.eks. argumentet er **[1, 2, 2, 2, 5]** så vil den iht. Yatzy-logikk returnere **[2, 2, 2] = 6**. En tilsvarende implementasjon for Farkle vil ha samme logikk for å finne tre like, men ha annen logikk for å sette poengene (= 2 * 100).

7 Three or more of a kind

Skriv kode for klassen **ThreeOrMoreOfAKind** (tre eller flere like), som implementerer **DiceScorer** med følgende logikk:

For et **Dice**-argument uten tre eller flere like returneres null. Ellers returneres et (nytt) **Dice**-objekt med alle de like terningverdiene, altså tre eller flere like verdier. For tre like får en 100 ganger verdien det er tre eller flere like av, f.eks. 300 poeng for tre 3-ere. For hver ekstra terning med samme verdi så *dobles* poengene, f.eks. får en 1200 (= 300 * 2 * 2) poeng for fem 3-ere.

Merk at hvis en har mange nok terninger, så kan en få tre eller flere like for flere terningverdier. En skal da velge den terningverdien som gir *flest poeng*. Hvis en f.eks. har terningene [2, 2, 2, 2, 5, 5, 5] så skal en velge de tre 5-erne, siden de gir 500 poeng, mens de fire 2-erne gir bare 400. Hvis flere terningverdier gir samme, høyeste poengverdi, så skal den med flest like terninger velges.

Skriv ditt svar her...

```
1  public Class ThreeOrMoreOfAKind implements DiceScorer{
2
3
4      Public Dice getScore(Dice dice){
5          int Score;
6          if (!checkForTuples){
7              return null;
8          }
9          if (checkForTrips(dice) != null){
10             if (!checkForMults){
11                 Score = scoreTrips(dice);
12                 return score;
13             }
14             return i*100;
15
```

```
        }
        if (checkForQuads(dice) != null){
            if (!checkForMults){
                Score = scoreQuads(dice);
                return score;
            }
            return i*200;
        }
        if (checkForQuints(dice) != null){
            if (!checkForMults){
                Score = scoreQuints(dice);
                return score;
            }
            return i*400;
        }
        if (checkForHexes(dice) != null){
            if (!checkForMults){
                Score = scoreHexes(dice);
                return score;
            }
            return i*800;
        }
            }

    }


    public int scoreTrips(Dice dice){
        int n = checkfortrips(dice);
        return n*100
    }
    public scoreQuads(Dice dice){
        return scoreTrips(dice)*2;
    }
    public scoreQuints(Dice dice){
        return scoreTrips(dice)*2*2;
    }
    public scoreHexes(Dice dice){
        return scoreTrips(dice)*2*2*2;
    }

    public int checkForHexes(Dice dice){
       for (int i=1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count==6){
                return i;
            }
        }
        return null;
    }
     public int checkForQuints(Dice dice){
       for (int i=1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count==5){
                return i;
            }
        }
        return null;
    }
     public int checkForQuads(Dice dice){
       for (int i=1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count==4){
                return i;
            }
        }
        return null;
    }
     public int checkForTrips(Dice dice){
       for (int i=1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count==3){
                return i;
            }
        }
        return null;
    }

     public int checkForTuples(Dice dice){
        for (int i=1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count>2){
                return i;
            }
        }
        return null;
     }
     public int checkForMults(Dice dice){
        int n = checkForTuples();
        for (int i=n+1; i<7; i++){
            if (dice.diceList.stream().filter(p->p==i).count>2){
                return i;
            }
        }
        return null;

     }
```

```
108
109
110
111 //Hei, fra starten av antok jeg at det var maks 6 terninger, derfor ble denne delen rotete ettersom
        jeg regnet med man kunne bare på flere triple, ikke friple. Derfor ble mye av oppgaven å tette
        lekkasjer. sorry.
```

Maks poeng: 6

8 Total score computation

Hvis en har mange terninger og er heldig, så kan én eller flere **DiceScore**-regler kombineres flere ganger. En får da summen av poengene som hver bruk av en regel gir. Én terning kan selvsagt ikke være med i mer enn én kombinasjon. Hvis en f.eks. har "tre eller flere like"-regelen implementert av **ThreeOrMoreOfAKind**-klassen og får fire 3-ere og tre 4-ere så vil en få 600 poeng for 3-erne og 400 poeng for 4-erne og dermed 1000 til sammen.

Det blir mer komplisert hvis flere regler "konkurrerer" om de samme terningene. F.eks. vil jo de samme fire 3-ere og tre 4-ere utgjøre hus (to 3-ere og tre 4-ere) og ett par (to 3-ere), hvis tilsvarende poengregler var med.

Poengberegningen for et spesifikt spill gjøres generelt som følger:

- En har en oversikt over alle **DiceScorer**-objektene som gjelder, f.eks. i en tabell eller liste. Gjør nødvendige antagelser om hvor disse er lagret og kan hentes ut.
- En prøver alle **DiceScorer**-objektene etter tur og det som gir flest poeng brukes først. Terningene som er i den tilsvarende kombinasjonen fjernes, og så gjentar en dette til alle poenggivende terninger er "brukt opp". Resultatet er (en **Collection** med) alle **Dice**-objektene som tilsvarer de kombinasjonene som ble brukt med poengverdien satt.

For eksemplet over, med bare **ThreeOrMoreOfAKind**-regelen og terningene **[1, 3, 3, 3, 3, 4, 4, 4, 5]** så vil en få som resultat (en **Collection** med) **Dice**-objektene **[3, 3, 3, 3] = 600** og **[4, 4, 4] = 400**. To av terningene, 1 og 5, ble i dette tilfellet ikke poenggivende og dermed ikke med i resultatet.

*/**
** Computes a set of Dice with scores for the provided Dice.*
** @param dice*
** @return the set of Dice with die values and corresponding scores.*
**/*
public Collection<Dice> computeDiceScores(Dice dice) {
...
}
Skriv ditt svar her...

```
1
2  public Collection<Dice> computeDiceScores(Dice dice){
3      Collection<Dice> dicescore;
4      Collection<Int> anslist;
5      int res;
6      for (int k = 1; k<7; k++){
7          res = dice.getDicelist.stream().filter(p->p==k).count
8          if (res>2){
9              for (i=0; i<res; i++)
10                 ansList = new ArrayList();
11                 anslist.add(k)
12             )
13             dicescore.add(new Dice(anslist, anslist.getScore));
14         )
15     }
16     return ansList;
17 }
18
19
```

Maks poeng: 6

9 Summing with the Stream technique

Vis hvordan du kan beregne totalpoengsummen som en **Collection** av **Dice**-objekter tilsvarer, vha. Stream-teknikken. Du har altså en **Collection** av **Dice**-objekter med poengverdier og skal summere disse poengverdiene:

Collection<Dice> diceObjectsWithScore = computeDiceScores(...);

int totalPoints = ... compute total points using the Stream technique ...

Skriv ditt svar her...

```
1  public int getTotalFromCollection(Collection<Dice> diceObjectsWithScore){
2
3      int totalPoints;
4      diceObjectsWithScore.stream().forEach(p->{
5      totalPoints+=p.getScore;})
6      return totalPoints;
7  }
```

Maks poeng: 3

Det finnes flere måter å implementere **Dice**-klassen og alle dens metoder, med ulike fordeler og ulemper. En måte å tillate bruk av flere implementasjoner er å gjøre om **Dice** til et *grensesnitt* og så ha en eller flere implementasjonsklasser, hvor den eksisterende **Dice**-klassen blir en av disse:

```
public interface Dice {
   ...
}
public class DiceImpl1 implements Dice { ... tilsvarer løsningen i deloppgavene 1-6 ... }
public class DiceImpl2 implements Dice { ... alternativ løsning ... }
```

Navnene på implementasjonsklassene kan selvsagt være mer forklarende.

10

The Dice interface

Tre alternative grensesnitt er foreslått, basert på den nåværende **Dice**-klassen:

```
// Alternativ 1, alle metoder og konstruktører:
public interface Dice ... {
   static Collection<Integer> randomDieValues(int dieCount);
   Dice(Collection<Integer> dieValues, int score);
   Dice(int dieCount);
   Dice(Dice dice, int score);
   static Dice valueOf(String s);
   int getScore();
   void setScore(int score);
   int getDieCount();
   int getDieValue(int dieNum);
   int getValueCount(int value);
   boolean contains(Dice dice);
   Dice add(Dice dice);
   Dice remove(Dice dice);
}

// Alternativ 2, alle metoder:
public interface Dice ... {
   static Collection<Integer> randomDieValues(int dieCount);
   static Dice valueOf(String s);
   int getScore();
   void setScore(int score);
   int getDieCount();
   int getDieValue(int dieNum);
   int getValueCount(int value);
   boolean contains(Dice dice);
   Dice add(Dice dice);
   Dice remove(Dice dice);
}

// Alternativ 3, utvalgte metoder:
public interface Dice ... {
   int getScore();
   void setScore(int score);
   int getDieCount();
   int getDieValue(int dieNum);
   int getValueCount(int value);
   boolean contains(Dice dice);
   Dice add(Dice dice);
   Dice remove(Dice dice);
}
```

Hvilket av alternativene over bør anbefales?

- Grensesnittalternativ 3
- Grensesnittalternativ 2
- Grensesnittalternativ 1

Den opprinnelige **Dice**-klassen implementerer **Iterable<Integer>**. Spørsmålet er hvordan dette skal håndteres ved overgangen til et **Dice**-grensesnitt.

Velg ett alternativ

- Dice-grensesnittet *må* utvide (extends) Iterable<Integer>.
- Dice-grensesnittet *må* utvide (extends) Iterable<Integer> *og kan* liste opp metoden(e) fra Iterable.
- Dice-grensesnittet *må* liste opp metoden(e) fra Iterable.
- Dice-grensesnittet *må* både utvide (extends) Iterable<Integer> *og* liste opp metoden(e) fra Iterable.

Maks poeng: 4

11 Inheritance

Hvis en har flere implementasjoner av det nye **Dice**-grensesnittet, så kan en regne med at visse deler av disse vil bli nokså eller helt like.

Ett aspekt som typisk vil bli (nokså) likt er håndtering av poengene (score). Forklar med tekst og/eller kode hvordan du vil håndtere det vha. arvingsmekanismen, slik at løsningen tillater stor grad av gjenbruk av kode i subklasser og blir ren og ryddig.

Skriv ditt svar her...

```
1  Her kan det være lurt å bruke en abstrakt superklasse som er en superklasse for alle
       terningkastespill. Denne klassen vil være abstrakt ettersom det ikke gir mening å instansiere en
       slik klasse. Denne abstraket klassen vil være en slags mal for de som arver av den.
2
3  Arvemekansimen tillater meget god gjenbruk av kode, som for eksempel i konstruktøren, hvor man ville
       kalt super(param) først, der super() er konstruktøren for superklassen. Dermed vil Dice()
       konstruerer i bilde av sin superklasse og kan etterpå gjøre videre endringer i konstruktøren som
       er spesifikke til denne diceklassen og ikke til andre som bruker samme superklasse. Validering er
       mye lettere med arv da i superklassen kan man gjøre validering som er felles til alle subklassene
       , og så kan man gjøre tweakinger i subklassen for å gjøre validering som ikke er generellt for
       alle subklassene. F.eksempel i Dice-konstruktøren ville valideringen av at dice-veridene i lista
       er over 1, blitt gjort i superklassen da dette er felles for alle dicespill. I denne Dice klassen
       er maksimum for terning-verdi 6, dermed ville denne valideringen skje subklassen. Dette er fordi
       andre terningspill spiller med terninger som går høyere enn 6. Derfor ville vi overridet den
       valideringsmetoden i subklassen og kalt på metoden i superklassen men også lagt til en liinje om
       at alle terningene skulle maks være 6.
4
```

Maks poeng: 6

12 Delegation

Dice sin **add**-metode skal lage en ny **Dice**-instans (altså instans av en klasse som implementerer **Dice**) som kombinerer terningverdier fra to andre **Dice**-instanser (this og argumentet). En kan tenke seg at metoden returnerer en instans av en ny **Dice**-implementasjon kalt **DiceAdder**, som bruker *delegering*. Den vil ha to **Dice**-felt og en konstruktør som tar inn to **Dice**-instanser:

DiceAdder(Dice dice1, Dice dice2) { ... feltene settes her ... }

Hver **Dice**-metode kan da bruke/delegere til disse to Dice-instansene i sin løsning, f.eks. vil **getDieCount()** returnere summen av **getDieCount()** for hver av de to **Dice**-instansene:

```
public int getDieCount() {
  return dice1.getDieCount() + dice2.getDieCount();
}
```

Forklar med tekst og/eller kode hvordan delegeringsteknikken vil bli brukt i følgende metoder i en slik **DiceAdder**-klasse: **getDieValue**, **getValueCount**, **contains**, **add** og **remove**. Kommenter spesielt hvis delegeringsteknikken ikke passer for en spesifikk metode!

Skriv ditt svar her...

```
1
2  public int getDieValue(int n){
3      if (n<dice1.getDicecount){
4          return dice1.getDieValue(int n);
5      }
6      return dice2.getDieValue(n);
7  }
8
9
10 Public int getDieValueCount(){
11     return dice1.getDieValueCount() + dice2.getDieValueCount
12 }
13
14 public boolean contains(dice dice){
15     return (dice1.contains(dice)||dice2.contains(dice));
16 }
17 public Dice add(Dice dice){
18     Dice newDice = new DiceAdder(this, Dice);
19     return newdice;
20 }
21
22 public Dice remove({
23
24     if (dice1.remove(Dice) != dice1){
25         return dice1.remove + dice2;
26     }
27     return dice1+dice2.remove;
28 }
29
```

Maks poeng: 10

Så langt har koden vært relativt uavhengig av typen terningspill. I denne deloppgaven handler det om en runde i Farkle-spillet. I en slik runde så akkumuleres det poeng ved at en gjentatte ganger kaster terninger, og sparer på en eller flere poenggivende terningkombinasjoner.

Reglene for hvilke kombinasjoner som gir poeng varierer noe mellom ulike varianter av Farkle, men en får typisk poeng for tre eller flere like, og for femmere (50 poeng pr. stk) og enere (100 poeng pr. stk). En tenker seg at dette beregnes av en metode tilsvarende **computeDiceScores** fra deloppgave 8 om totalpoengberegning.

Etter hvert kast har en tre tilfeller/muligheter:

1. Hvis ingen terningkombinasjoner gir poeng, så er runden over og en mister alle poengene en evt. har akkumulert.
2. En kan velge å stoppe og får poeng for terningkombinasjonene en har spart så langt og for de gjenværende som en akkurat kastet.
3. En kan velge å legge til side en eller flere poenggivende terningkombinasjoner, og få poeng for dem, og så kaste resten. Hvis en beholder alle terningene en kastet så det ikke er noen igjen å kaste, så kan en kaste alle på nytt.

For hvert kast så får en de samme tre mulighetene. Avveiningen i spillet er altså om en skal beholde poengene en har fått så langt, eller kaste på nytt for å få flere poeng, men risikere å miste alle.

Her er noen eksempler som illustrerer reglene:

1. En kaster 2, 3, 3, 4 og 4. Runden stopper opp av seg selv, med 0 poeng som resultat :-(
2. En kaster 2, 3, 4, 5 og 5. En legger til side en 5-er for 50 poeng, og kaster de fire gjenværende terningene. En får 1, 3, 5 og 6. En legger til side 1-eren og 5-eren for 150 nye poeng, altså 200 til sammen så langt, og kaster de to terningene som er igjen. En får da 4 og 4, og mister alle poengene :-(
3. **En kaster som i pkt. 3, men stopper før det siste kastet og får altså 200 poeng for runden :-)**
4. En kaster som over, men er heldig og får to 5-ere for 50+50 poeng i de siste kastet og kan kaste alle på nytt. En får da 1, 1, 1, 4 og 6, sparer de tre 1-erne for 1000 poeng (spesialregel for poenggiving i Farkle) og stopper. Runden gir da 200+50+50+1000 poeng :-)

Et forslag til realisering av en klasse som håndterer en Farkle-runde, er vist under. Noen deler er imidlertid utelatt...

```
/**
 * Represents a round of Farkle, where a player throws and keeps dice,
 * until s/he either "bank", i.e. save your points, or
 * farkle, i.e. get no points in a throw and hence looses all gathered points.
 * During and after a finished round, kept sets of dice and their scores are available.
 * During a round, the remaining dice are also available.
 */
public class FarkleRound {
    private int dieCount;
    private Collection<Dice> kept = new ArrayList<>();
    private Dice currentDice;

    /**
     * Initializes a new round, where dieCount number of dice is immediately rolled.
     * Note that the round may be immedielately finished, if the initial roll give no points.
     * @param dieCount the number of dice rolled
     * @param scoring the object responsible for computing the score
     */
    public FarkleRound(int dieCount) {
        this.dieCount = dieCount;
        roll(dieCount);
    }
    private void roll(int dieCount) {
        this.currentDice = new Dice(Dice.randomDieValues(dieCount), -1);
        if (computeDiceScores(currentDice).isEmpty()) {
            this.kept.clear();
            this.currentDice = null;
        }
    }
    private Collection<Dice> computeDiceScores(Dice dice) {
       ... implemented earlier in this exam ...
    }
```

```
    /**
     * @return true of the round is finised, false otherwise
     */
    public boolean isFinished() {
        return this.currentDice == null;
    }

    /**
     * Called when the player decides to stop and secure points.
     * Finishes the round, by keeping all scoring Dice, as computed by the scoring object.
     */
    public void bank() {
        if (isFinished()) {
            throw new ... which exception class ? ...
        }
        this.kept.addAll(computeDiceScores(currentDice));
        ... put this object in a state that indicates this round is finished ...
    }

    /**
     * Called when the player decides to keep specific dice and roll the rest.
     * All the dice kept must be scoring ones, but not all scoring dice need to be kept.
     * @param dice the dice to keep, the rest of the current dice should be thrown.
     */
    public void keepAndRoll(final Dice dice) {
        if (isFinished()) {
            throw new ... which exception class ? ...
        }
        if (! currentDice.contains(dice)) {
            throw new ... which exception class ? ...
        }
        final Collection<Dice> scores = computeDiceScores(dice);
        if (scores.stream().mapToInt(Dice::getDieCount).sum() != dice.getDieCount()) {
            throw new SomeKindOfException("You can only set aside dice that contribute to the score");
        }
        kept.addAll(scores);
        currentDice = currentDice.remove(dice);
        // roll remaining dice or all, if there are none left
        ... what code needs to be here ? ...
    }
}
```

13

final

Hvilke, om noen, av feltene øverst i **FarkleRound**-klassen kan ha modifikatoren **final**?

Velg 0-3 alternativer

- [ ] currentDice
- [x] dieCount
- [ ] kept

Maks poeng: 2

14 Code in bank

Nederst i **bank**-metoden er en kodelinje erstattet med "... put this object in a state that indicates this round is finished ...". Skriv inn den korrekte kodelinjen. Koden må være korrekt, komplett og konsis, uten kommentarer, siden den sjekkes automatisk.

```
currentDice = null;
```

Maks poeng: 2

15 Exception classes

I **keepAndRoll**-metoden utløses unntak i tre tilfeller.

Hvilken av disse unntakstypene passer i det første tilfellet?
Velg ett alternativ

- IllegalArgumentException
- IllegalStateException
- Exception
- RuntimeException

Hvilken av disse unntakstypene passer i det andre tilfellet?
Velg ett alternativ

- Exception
- RuntimeException
- IllegalArgumentException
- IllegalStateException

I det tredje tilfellet bruker vi unntakstypen **SomeKindOfException**, som er tenkt som en egendefinert unntakstype. Hvilken unntakstype bør brukes som superklassen til **SomeKindOfException**?

Velg ett alternativ

- Exception
- RuntimeException
- IllegalStateException
- IllegalArgumentException

Maks poeng: 3

16 Code in keepAndRoll

Nederst i **keepAndRoll**-metoden er én eller flere kodelinjer erstattet med "... what code needs to be here ? ...".
Skrive linjen(e) som mangler. Koden vil bli lest av sensor.

Skriv ditt svar her...

```
if(currentDice.getDieCount == 0){
  roll(dieCount);
}
else{
  roll(dieCount - currentDice.getDieCount);
}
```

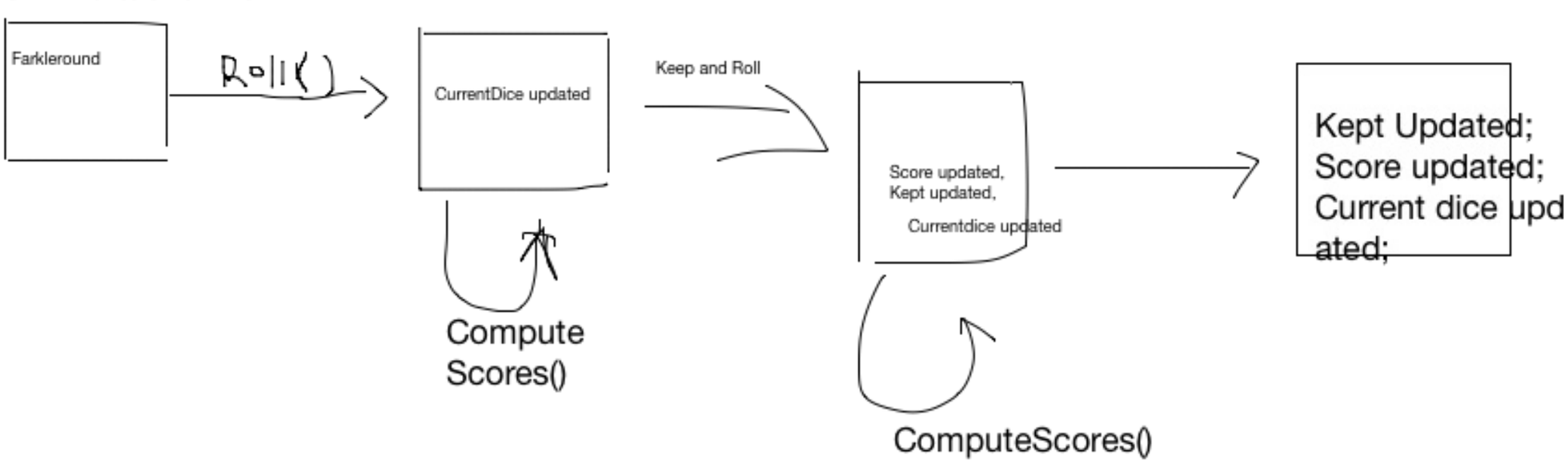
Maks poeng: 2

17 FarkleRound object diagram

Underveis i og etter en Farkle-runde vil tilstanden til runden være representert ved hjelp av flere objekter, inkludert et **FarkleRound**-objekt. Tegn objektdiagram for tilstanden *etter* runden som er beskrevet i det *tredje* eksemplet (fremhevet med **fet** skrift) for Farkle-reglene.

Bruk blyantikonet til høyre på verktøylinja for tekstfeltet, for å aktivere tegneverktøyet. Det kan være lurt å kladde på forhånd, siden tegneverktøyet er litt begrenset...

Skriv ditt svar her...



Maks poeng: 6

Dice-klassen har en konstruktør som initialiserer den med tilfeldige terningverdier, og det gjør den vanskelig å teste, fordi en jo ikke vet hvilke verdier random-metoden genererer. Et alternativ er en konstruktør som henter verdier fra en **Supplier<Integer>** (se nedenfor). Til vanlig kan en bruke en **Supplier**-implementasjon som leverer tilfeldige tall vha. **Math.random()**, mens til testing kan en lage en som gir ut bestemte verdier:

```
/**
 * Initializes this Dice with n die values provided by the supplier argument.
 * @param dieCount the number of dice to "throw"
 * @param supplier provides the die values
 */
public Dice(int dieCount, Supplier<Integer> supplier) { ... }
```

Supplier er deklarert som følger:

```
public interface Supplier<T> {
    /**
     * Gets a result.
     * @return a result of the type T
     */
    T get();
}
```

18

Functional interface?

Er **DiceScorer** et funksjonelt grensesnitt? Forklar hvorfor/hvorfor ikke!

Skriv ditt svar her...

Ja, Supplier er et funskjonelt grensesnitt. Den har kun en metode og grensesnittet har en funksjon til bruk. Supplier kan brukes i Lambda-funksjoner på streams.

Maks poeng: 3

19

Alternative Dice constructor

Implementer den alternative **Dice**-konstruktøren.

Skriv ditt svar her...

```
1  public Dice(int dieCount, Supplier<Integer> supplier){
2      Collectiion<Integer> suk = new Arraylist();
3
4      for (i=0; i<dieCount; i++){
5         suk.add(supplier.get)
6      }
7      return suk;
8
9  }
```

Maks poeng: 3

20 Testing of the Dice class

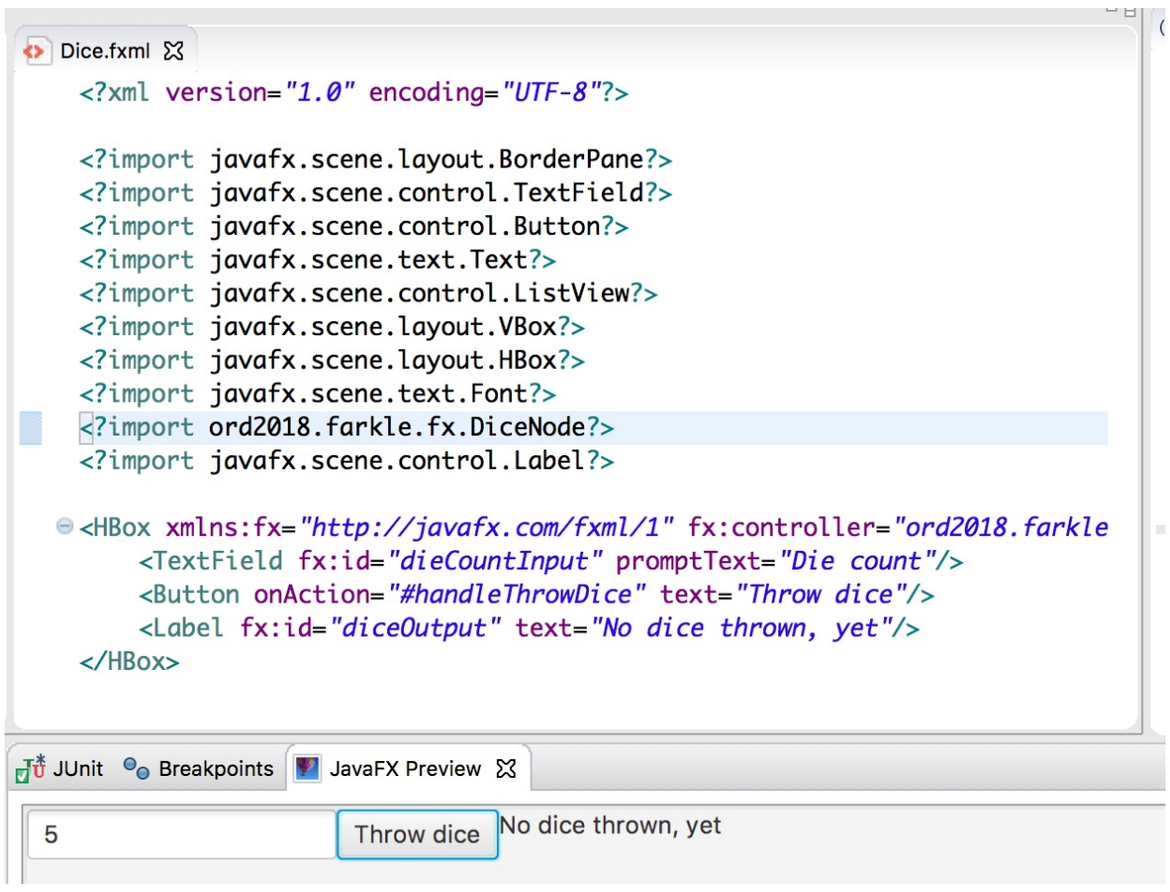
Skriv en eller flere test-metoder for **Dice** sin **valueOf**-metode. Du kan anta det finnes en metode kalt **assertDieValues** som sjekker terningverdiene til en **Dice**:

assertDieValues(dice, 1, 2, 3) // utløser assert-unntak hvis dice *ikke* har terningverdiene 1, 2, 3 og bare disse.

Skriv ditt svar her...

```
1  public ValueOfTest(Dice dice){
2
3      assertDieValues(Dice.ValueOf("[2, 2, 2] = 200") ,2,2,2);
4
5  public ValueOfDieTest{
6      try{
7          assertDieValues(Dice.valueOf("[5, 3, 1] = 200") ,2,2,2);
8          fail()
9      }
10     catch(assertException e){
11
12     }
13     catch(exception e){
14         Print(e);
15         //dette skal da ikke skje gitt at assertDieValues fungerer(?);
16     }
17
18     assertDieValues(Dice.valueOf("[5, 3, 2] = 200") ,2,5,3);
19
20 }
21
22 public ValueOfScoreTest(Dice dice){
23
24     assertEquals(100, Dice.valueOf(["1,1,1] = 100").getScore, 0.2); //Må ha med Delta her
25     assertFalse(200, Dice.valueOf(["1,1,1] = 100").getScore, 0.2);
26 }
27
28 public valueOfNoScore(Dice dice){
29     assertEquals(-1, Dice.valueOf(["1,1,1]").getScore, 0.2);
30     assertFalse(-1, Dice.valueOf(["1,1,1] = 10").getScore, 0.2);
31 }
32
```

Maks poeng: 5



Du skal lage en liten JavaFX-app for terningspill basert på **Dice**-klassen. I første omgang skal du støtte terningkast og poengberegning:

- Brukeren skal kunne fylle inn et tall **n** i et tekstfelt (TextField)
- Når en knapp (Button) trykkes, så skal det lages et **Dice**-objekt med **n** tilfeldige tall.
- Poengverdien til **Dice**-objektet settes vha. metoden **void computeFarkleScore(Dice)**, som du kan anta finnes.
- Den tekstlige representasjonen til **Dice**-objektet skal så vises som en tekst (Label)

Følgende FXML er skrevet for appen:

```
<HBox xmlns:fx="http://javafx.com/fxml/1" fx:controller="ord2018.farkle.fx.DiceController">
  <!--  text field for inputing the die count -->
  <TextField fx:id="dieCountInput" promptText="Die count"/>
  <!--  button for throwing the dice, i.e. create a Dice object and computing the score -->
  <Button onAction="#handleThrowDice" text="Throw dice"/>
  <!--  label for outputing the textual representation of the Dice object -->
  <Label fx:id="diceOutput" text="No dice thrown, yet"/>
</HBox>
```

Se også illustrasjonen.

21 JavaFX controller

Skriv en JavaFX-kontroller som implementerer ønsket oppførsel. Den skal passe til den oppgitte FXML-en og bruke klasser og metoder beskrevet tidligere i oppgavesettet.

Hvis det er detaljer du er usikker på, så forklar med tekstkommentarer i koden.

Skriv ditt svar her...

```
public TextField InputDieCount
public Label OutputText;



@FXML
public Button handleThrowDice{
    roll(InputDieCount);
    UpdateDiceOutput();
}

@FXML
public void DiceOutput(Dice dice){
    OutputText = Label.ValueOf(computeFarkleScore(Dice));
}
```

Maks poeng: 6