TEST

Software Systems: Programming

course code: 201700117
date: 30 January 2020
time: 13:45 – 16:45

SOLUTIONS

General

- You may use the following (unmarked) materials when making this test:
  - Module manual.
  - Slides of the lectures.
  - The book
    David J. Eck. *Introduction to Programming Using Java*. Version 8.1, July 2019.
  - A dictionary of your choice.

  The module manual, the book and the slides can be found at `https://www.utwente.nl/en/telt/learning/intranet/books/`
- You are *not* allowed to use any of the following:
  - Solutions of any exercises published on Canvas (such as recommended exercises or old tests);
  - Your own materials (copies of (your) code, solutions of lab assignments, notes of any kind, etc.).
- When you are asked to write Java code, follow code conventions where they are applicable. Failure to do so may result in point deductions. You do *not* have to add annotations or comments, unless explicitly asked to do so. Invariants, preconditions and postconditions should be given only when they are explicitly asked. When asked to implement a method, give the *full* method definition, including the method signature and the method body.
- No points will be deducted for minor syntax issues such as semicolons, braces and commas in written code, as long as the intended meaning can be made out from your answer.
- This test consists of 6 exercises for which a total of 100 points can be scored. The minimal number of points is zero. Your final grade of this test will be determined by the sum of points obtained for each exercise.

Question 1 (20 points)

In this test, we consider the interfaces, classes and methods necessary to implement an application for managing a database of movies and TV series similar to IMDb (Internet Movie Database, `imdb.com`).

a. *(5 points)* Define an interface `Entry` that models an entry (a movie or TV series) of our database. For each entry, this interface should have methods to query the *title*, the *average rating*, which a decimal value from 1 to 10 and the *number of ratings*, which indicates how many people rated this item. This interface should also have a method that allows a user to enter a rating for (*rate*) this entry, which is an integer value ranging from 1 to 10. Define reasonable *preconditions and postconditions* for these methods.

b. *(10 points)* Entries in the database have some functionality in common that can be implemented in an abstract class. Program an abstract class `AbstractEntry` that implements the interface `Entry`. This class should have the following instance variables:

- `String title` to store the title of the entry.
- `List<Integer> ratings` to store all ratings given to this entry.
- **`int`** `sum` to store the sum of all ratings given to this entry.

Define an appropriate constructor for this class. Finally, since entries should be *comparable* to each other, class `AbstractEntry` should also implement the `Comparable<Entry>` interface. The generic interface `Comparable<T>` has one method with the signature **`int`** `compareTo(T o)`. This method is defined as follows:

> **`int`** `compareTo(T obj)`
>
> Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
>
> **Parameters:**
> `obj` – the object to be compared.
>
> **Returns:**
> a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object `obj`.

This comparison is made based on the average rating, i.e., entries are ordered based on their ratings (entries with higher ratings are 'greater').

c. *(5 points)* Explain why it might be convenient to implement an abstract class like `AbstractEntry` and why this benefit cannot be achieved with interfaces, even if we use default methods like it is allowed since Java 8.

Answer to question 1

a. (5 points)

```
public interface Entry {

    /**
     * Get title of the entry
     * @ensures result != null
     */
    public String getTitle();

    /**
     * Get average rating
     * @ensures 1.0 <= result <= 10.0
```

```
     */
    public double getAverageRating();

    /**
     * Get number of ratings
     * @ensures result >= 0
     */

    public int getNumberOfRatings();

    /**
     * Add a rating
     * @requires 1 <= result <= 10
     *
     */
    public void rate(int rate);
}
```

Grading criteria:

- 1 point: no instance variables (because **`interface`**).
- 2 points: getters for all three properties with reasonable return types (**`int`** for integer and **`double`** for decimal) and well as a proper method to add a rating.
- 2 points: postconditions close to standard answers. Pay attention that the integer and double values have reasonable postconditions, with boolean conditions involving numeric values and not things like `result != null`. Java boolean expressions are not required if postconditions are properly (clearly) defined informally (in text).

b. *(10 points)*

```
abstract public class AbstractEntry
            implements Entry, Comparable<Entry> {
    private String title;
    private List<Integer> rankings;
    private int sum;

    protected AbstractEntry(String title) {
        this.title = title;
        this.rankings = new ArrayList<>();
        this.sum = 0;
    }

    public String getTitle() {
        return this.title;
    }

    public double getAverageRating() {
        return ((double)sum ) / this.rankings.size();
    }

    public int getNumberOfRatings() {
        return this.rankings.size();
    }


    public void rate(int rate) {
        this.rankings.add(rate);
```

```
            sum = sum + rate;
        }

        public int compareTo(Entry other) {
            double myAverage = this.getAverageRating();
            double otherAverage = other.getAverageRating();
            return Double.compare(myAverage, otherAverage);
        }
    }
```

Grading criteria:

- 2 points: proper class declaration with **`abstract`** and **`implements`**.
- 2 points: instance variables with reasonable types as indicated in the question.
- 2 points: constructor properly defined to initialise all instance variables.
- 2 points: all interface methods properly implemented (1 point if not all methods are correctly implemented, but 2 or 3).
- 2 points: method `compareTo` properly implemented. No problem if `Double.compare` is not used as long as the implementation is correct.

c. *(5 points)* Although an abstract class cannot be instantiated, the functionality defined in an abstract class can be reused in multiple concrete subclasses of this class, so that their code can be simpler. Reusability in the case of interfaces is rather limited to (fixed) default values of method returns, because instance variables cannot be defined in interfaces like it is possible with abstract classes. Grading criteria:

- 3 points: mentioned reusability as the main benefit.
- 2 points: mentioned that it is not possible to define instance variables in interfaces (or limited reusability of interfaces).

Question 2 (15 points)

a. *(3 points)* Implement a class called `Movie` that extends the abstract class `AbstractEntry`. A `Movie` has a *release year*. In your implementation, define proper methods to *retrieve the release year*, as well as a *constructor* to instantiate a `Movie` given a title and a release year as arguments.

b. *(7 points)* Implement a class `TVSeries` that extends the abstract class `AbstractEntry`. A `TVSeries` has a *number of episodes*. In your implementation, define methods to *retrieve the number of episodes* and to *increment this number by one*, as well as a *constructor* to instantiate a `TVSeries` with a given title. The number of episodes of a new TV series is always one. Define one reasonable *invariant* for the `TVSeries` class. Define *preconditions and postconditions* for all methods and the constructor of the `TVSeries` class.

c. *(5 points)* Implement a test class `TVSeriesTest` for `TVSeries` that tests the following functionality:
   - Whether after 4 rates are added, the average rating and the number of ratings are correctly given.
   - Whether the number of episodes is correctly initialized and correctly given after it is increased twice.

   For simplicity, you may implement all tests in a single test method, but do not forget to give the proper annotations.

Answer to question 2

a. *(3 points)*

```
public class Movie extends AbstractEntry {
    private int releaseYear;

    public Movie(String title, int releaseYear) {
        super(title);
        this.releaseYear = releaseYear;
    }

    public int getReleaseYear() {
        return this.releaseYear;
    }

}
```

Grading criteria:

- 1 points: added an instance variable for release year with and proper getter.
- 2 points: constructor properly sets all instance variables, including release year, using **`super`**. Give only 1 point if all instance variables are properly initialised but **`super`** is not used.

b. *(7 points)*

```
/**
 * Class that represents TV series
 * Each series has at least one episode
 *
 * @invariant episodes >= 1
 */
public class TVSeries extends AbstractEntry {

    private int episodes;
```

```
    /**
     * Constructor for a TV series
     *
     * @requires title != null
     * @ensures getTitle() == title && getEpisodes() == 1
     */
    public TVSeries(String title) {
        super(title);
        episodes = 1;
    }

    /**
     * Adds an episode to the series
     *
     * @ensures getEpisodes() == old.getEpisodes() + 1
     */
    public void addEpisode() {
        episodes++;
    }

    /**
     * Gets the number of episodes of the series
     *
     * @ensures result > 0
     */
    public int getEpisodes() {
        return episodes;
    }

}
```

Grading criteria:

- 2 points: added an instance variable for episode with and proper getter and incrementer.
- 2 points: constructor properly sets all instance variables, including episodes.
- 1 points: proper invariant, close enough to the standard answer.
- 2 points: proper precondition (1) and postconditions (3), close enough to the standard answer. Give only 1 point if 2 or 3 of them are properly defined.

c. *(5 points)*

```
class TVSeriesTest {

    private TVSeries greatSeries;

    @BeforeEach
    void setUp() throws Exception {
        greatSeries = new TVSeries("Wheel of Time");
    }

    @Test
    void test() {
        // adds four rates
        greatSeries.rate(10);
        greatSeries.rate(7);
        greatSeries.rate(7);
        greatSeries.rate(6);
```

```
            // test number of ratings and average
            assertEquals(4, greatSeries.getNumberOfRatings());
            assertEquals((10.0 + 7.0 + 7.0 + 6.0) / 4,
                    greatSeries.getAverageRating());

            // test episodes
            assertEquals(1, greatSeries.getEpisodes());
            greatSeries.addEpisode();
            greatSeries.addEpisode();
            assertEquals(3, greatSeries.getEpisodes());

        }

    }
```

Grading criteria:

- 1 point: `TVSeries` object properly instantiated.
- 2 points: number of ratings and average properly tested with 4 values (1 point each).
- 2 points: initialisation and increment of episodes properly tested (1 point each).

Question 3 (20 points)

Below you can find an incomplete class `SSMDB`, which defines the so-called *Software Systems Movie Database*.

```
public class SSMDB {
    private Set<Entry> entries;

    public SSMDB() {
        entries = new HashSet<Entry>();
    }

    /**
     * Returns all entries of the database.
     * @ensures result != null
     */
    public Set<Entry> getEntries() {
        return entries;
    }

    /**
     * Returns a Movie with the title and release year
     * or null if the Movie is not found
     */
    public Movie getMovie(String title, int year) {
        // TO BE IMPLEMENTED !
    }

    /**
     * Returns a map that contains all the movies that were released
     * in a given year.
     */
    public Map<Integer, Set<Movie>> getMovies() {
        // TO BE IMPLEMENTED !
    }

    /**
     * Returns the entry with the beste (highest) rating that is smaller
     * or equal to a certain given rating or null if no entry satisfies
     * this condition.
     */
    public Entry getBest(double highest) {
        // TO BE IMPLEMENTED !
    }

    public void addEntry(Entry entry) {
        entries.add(entry);
    }
}
```

a. *(5 points)* Implement the method `getMovie`, which given a title and year, returns a `Movie` with this title and release year, or **`null`** if this `Movie` does not exist in the database.

b. *(7 points)* Implement the method `getMovies`, which returns a `Map` that relates each release year with the set of movies released in that year. This means that given a certain key `year`, `getMovies().get(year)` returns a set with all movies released in `year`.

c. *(8 points)* Implement the method `getBest(int highest)`, which returns one `Entry` that has a rating that is the best in our SSMDB but is still lower than or equal to `highest`, or **`null`** if no such `Entry` exists. For example, `getBest(5.0)` returns the best entry in SSMDB with rating lower than or equal to 5.0 or **`null`** if all entries in SSMDB have a rating higher than 5.0. In this question, you should assume that `highest >= 1 && highest <= 10` is a precondition of `getBest(int highest)`.

Answer to question 3

a. *(5 points)*

```
public Movie getMovie(String title, int year) {
    Movie movie = null;
    for (Entry entry : getEntries()) {
        if ((entry instanceof Movie)
            && ((Movie) entry).getTitle().equals(title)
                && ((Movie) entry).getReleaseYear() == year) {
            movie = (Movie) entry;
            break;
        }
    }
    return movie;
}
```

Grading criteria:

- 1 point: define a variable of type `Movie` and return the value of this variable.
- 1 point: iterate with a loop over the set of entries.
- 1 point: check if entry is a `Movie` using lazy evaluation `&&`, or avoid a type exception in some way.
- 1 point: check title and year.
- 1 point: return **`null`** if not found.

b. *(7 points)*

```
public Map<Integer, Set<Movie>> getMovies() {
    Map<Integer, Set<Movie>> result =
            new HashMap<Integer, Set<Movie>>();
    for (Entry entry : getEntries()) {
        if (entry instanceof Movie) {
            Movie film = (Movie) entry;
            Set<Movie> sameDate = result.get(film.getReleaseYear());
            if (sameDate == null) {
                sameDate = new HashSet<Movie>();
                result.put(film.getReleaseYear(), sameDate);
            }
            sameDate.add((Movie) entry);
        }
    }
    return result;
}
```

Grading criteria:

- 1 point: define a proper variable to hold the return value (instantiated with a concrete class).
- 1 point: iterate with a loop over the set of entries.
- 3 points: create new set and add it to the result map if date is not there yet.
- 2 points: add each `Movie` to the proper set.

c. *(8 points)*

```
public Entry getBest(double highest) {
    Entry result = null;
    double best = 1; // stores the best rating
    for (Entry entry : getEntries()) {
        double rating = entry.getAverageRating();
        if (rating >= best && rating <= highest) {
            result = entry;
            best = rating;
        }
    }
    return result;
}
```

Grading criteria:

- 1 point: define a proper variable to hold the return value (of type `Entry`).
- 1 point: define and properly initialise a variable to hold the best rating during calculation
- 2 points: iterate with a loop over the set of entries.
- 2 points: properly query the average rating.
- 2 points: properly update values if a better entry is found.

Question 4 (15 points)

In our database, we do not want to have multiple objects for each specific movie or series, because this would complicate the calculation of the average ratings (we would have to combine ratings from different objects). This can achieved by having so called factory methods, like
`Movie createMovie(String name , int year)` and
`TVseries createTVSeries(String name , int episodes)`
to create a movie and a TV Series in the database, respectively. In order to indicate that a movie or TV series already exists and will not be created, these methods throw exceptions.

a. *(5 points)* Define an exception `MovieAlreadyExists` that has to be thrown by `createMovie` if the movie to be created already exists. This exception should hold enough information *in itself* for a developer or maintainer to understand which movie already existed.

b. *(10 points)* Implement method `createMovie` that creates and returns a movie if it does not exist, otherwise throws an `MovieAlreadyExists` exception. Movies are considered to be the same if they have the same title and the same release year.

Answer to question 4

a. *(5 points)*

```
public class EntryAlreadyExists extends Exception {

    public EntryAlreadyExists (String name, int year) {
        super("Movie:␣" + name + "(" + year + ")␣already␣exists");
    }
}
```

Grading criteria:

- 2 points: exception extends `Exception`.
- 3 points: exception keeps title and year somehow (in the message or as instance variables) and there is a way to view this information, either from the message content or with appropriate getters.

b. *(10 points)*

```
public Movie createMovie(String title, int year)
        throws MovieAlreadyExists {
    if (this.getMovie(title, year) != null ) {
        throw new MovieAlreadyExists(title, year);
    }
    else {
        Movie result = new Movie(title, year);
        this.entries.add(result);
        return result;
    }
}
```

Grading criteria:

- 2 points: exception is declared with **`throws`** in the declaration of `createMovie`.
- 2 points: properly create and throw the exception if `Movie` is found.
- 2 points: properly create a new `Movie`.
- 2 points: properly update the database.
- 2 points: properly return the created `Movie`.

Question 5 (15 points)

Study the following code fragment. Assume that the started threads are *not* daemon threads, i.e., they do not run continuously but are started when necessary.

```
class Stream implements Runnable {
    private Object obj;
    private char token;
    private static final int REPETITIONS=100;

    public Stream(char token) {
        this.obj = new Object();
        this.token = token;
    }

    public void run() {
        synchronized(obj) {
            for (int i = 0; i < REPETITIONS; i++) {
                System.out.print(token);
            }
        }
    }

    public static void main(String[] args) {
        new Thread(new Stream('x')).start();
        new Thread(new Stream('y')).start();
        new Thread(new Stream('z')).start();
    }
}
```

a. *(5 points)* What is the output produced by this code fragment? Properly explain your answer by describing how this piece of code works and why.

b. *(5 points)* How does this output change if **`synchronized`** block around the **`for`** loop is removed? Properly explain your answer by discussing why the output changes or remains the same.

c. *(5 points)* Suppose now that `obj` is made **`static`**, i.e., the declaration of `obj` is changed to **`private static`** `Object obj;` What is the output produced by this code fragment in this new case? Properly explain your answer by explaining why the output changes (or not) with respect to items a. and b.

Answer to question 5

a. *(5 points)* The threads will run concurrently in the sense that they will get turns to run for a while until they finish. Since each thread writes a character `token` 100 times, this means that characters `x`, `y` and `z` will be printed out in an arbitrary order, each one of them 100 times.

Grading criteria:

- 2 points: mention the correct output. Only 1 point if mention that this sequence can stop somewhere because the program may terminate.
- 3 points: explain why this is the case (mention turns or scheduling).

b. *(5 points)* Nothing changes because the **`synchronized`** block had no effect any way. Grading criteria:

- 2 points: mention the correct output (nothing changes, or the answer to a.).
- 3 points: explain why this is the case (there is no restriction, threads get turns).

c. *(5 points)* This question didn't go exactly as we planned. In the original version of the `Stream` class, variable `obj` was initialised with the declaration, like in

```
private Object obj = new Object();
```

Unfortunately we made a mistake when copying the code to the test. In the original version, making `obj` **`static`** would result in sequences of 100 times `x`, `y` and `z` being printed (no interleaving). However, because in the code in the test `obj` is associated to different object instances every time a `Stream` object is created, the mutual exclusion of the **`for`** loops will not occur since each reference refers to a different object. Therefore, in the grading we will accept both answers:

- Same as before because `obj` is initialised with different object, the **`synchronized`** block does not provide mutual exclusion.
- Sequences of 100 times `x`, `y` and `z` are printed without interleaving, because the **`synchronized`** block mutually excludes the **`for`** loops of each `Stream` instance. In principle, we cannot guarantee that these blocks of 100 characters are printed out in this specific order ($X \rightarrow Y \rightarrow Z$), but this is highly probable. The only certainty is that the blocks will not interleave.

Grading criteria:

- 2 points: mention the correct output.
- 3 points: explain why this is the case.

Question 6 (15 points)

Nowadays people can subscribe to streaming services and once they have paid an affordable fee they can watch videos (typically movies and TV shows) online, or, depending of their subscription, even download these videos to watch them offline (without Internet connection). This means that these services should keep track of these users and their subscriptions, so that they only use those services facilities for which they paid.

a. *(3 points)* What are the three main security properties/attributes that, when violated for a (software) system, indicate there has been a security incident?

In a streaming service, users have accounts and a password system is used to make sure only the rightful owners of the accounts can login.

b. *(3 points)* It is common practice to first apply a hash function to a password before storing it. Explain why it is a good a idea for such online systems to apply a hash function to their users' passwords instead of storing them as-is.

c. *(2 points)* If you were to choose between SHA-256 and PBKDF2 to store the passwords, which one would you choose? Explain your answer.

New users of a service are sent an initial password upon registration. Some services offers a subscription for free for a period of time (typically one month), after which the user has to start paying. After making several accounts in the system in an unrelated effort to try to cheat the system later, you may have noticed that the initial passwords are always of the same length and only contain a very specific set of characters. Some example passwords you find are:

- `pw342abEE`
- `pw222pqAA`
- `pw923pqD6`
- `pw123ab5A`

You are starting to see a pattern here! You guest the following scheme is used:

- Password always starts with "pw"
- then come 3 digits,
- followed by either "ab" or "pq"
- followed by 2 hexadecimal characters (uppercase)

d. *(2 points)* What is the worst-case number of passwords you need to try to guess a password from such a newly made account?

Suppose now that the streaming service offers two service levels, namely the *budget service* that is very cheap but the user can only watch movies when there is a network connection and the movies are 10 years old or older, and the *premium services* that is much more expensive and the users can watch all movies and also download them to watch them even when they do not have an Internet connection. Also suppose that it is possible to upgrade your subscription from budget to premium service, which means that the user has to pay a higher fee. Suppose this is working great for a while and the company is seeing a clear increase in profit. However, after a while the company is starting to hear rumours that it is possible to enter the premium services without needing to change the subscription. You are asked to figure out what is going on. You have a look at the source code of the game and you quickly realize it quite a mess. You for example notice that clumsy (and inefficient) programming constructions are being used in several places. You then find the code snippets printed below.

e. *(5 points)* Explain, using the code shown, how a malicious player can access the premium service without paying more. You can assume that users can set and change their names at will.

```
package test;

public class StreamingServiceUser {

    private String name;
    private boolean premium;
    private String bankAccount;

    // ... Other methods that are irrelevant for the question

    /**
     * Returns the user name and the subscription type separated by "::"
     */
    public String getInfo() {
        String text = this.premium ? "PREMIUM" : "BUDGET";
        return name + "::" + text;
    }

    // ... Other methods that are irrelevant for the question

}

public class PremiumService {
    // ... Other methods that are irrelevant for the question


    /**
     * Here we check the user is allowed to use this service
     * @param user
     * @return true if the user is allowed to use this service.
     */
    public boolean checkPremium(StreamingServiceUser user) {
        String info = user.getInfo();
        String[] infoItems = info.split("::");
        return infoItems[1].equals("PREMIUM");
    }
}
```

The JavaDoc documentation of the method `String.split` states the following:

> `public String[] split(String regex)`
>
> Splits this string around matches of the given regular expression.
>
> This method works as if by invoking the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.
>
> The string "boo:and:foo", for example, yields the following results with these expressions:
>
> | **Regex** | **Result** |
> |---|---|
> | `:` | `{ "boo", "and", "foo" }` |
> | `o` | `{ "b", "", ":and:f" }` |
>
> **Parameters**
> regex - the delimiting regular expression
>
> **Returns**
> the array of strings computed by splitting this string around matches of the given regular expression

Answer to question 6

a. *(3 points)* Confidentiality, Integrity, Availability (one point each). If only "CIA": 1 point. If three properties are provided that seem close enough: 2 points. But you can be pretty strict here.

b. *(3 points)* People often re-use password across domains; when site (and the database containing the passwords) is compromised, the accounts of the users at other sites can also be compromised.

c. *(2 points)* PBKDF2: this hash function is specifically designed to be slow, which makes brute-force attacks harder. SHA-256 is designed to be as fast as possible, making it unsuitable for this use-case. More grading guidelines:

   - Stating that "scrypt", "bcrypt", or "Argon" should be used instead: full points.
   - Only PBKDF2 without proper explanation: 1 point.

d. *(2 points)* $1 \times 10^3 \times 2 \times 16^2 = 512000$
Give 2 points if only calculation is shown, and 1 point if only the correct result is given (calculation was asked).

e. *(5 points)* A malicious user can simply add a string in the tagline (or even the name) that contains "::" followed by the text `PREMIUM`. This is then wrongly interpreted as the name plus premium level. Full points if a similar explanation is given. More grading guidelines:

   - 1 point if only something along the lines "change the name" is given as the answer.
   - 3 points if only a (working) example is given for the name or tagline, without an explanation.