TEST

Software Systems: Programming

course code: 202001024
date: 20 January 2023
time: 8:45 - 11:45

SOLUTIONS

General

- You may use the following (unmarked) materials when making this test:
  - Module manual.
  - Slides of the Programming topics.
  - The book
    David J. Eck. *Introduction to Programming Using Java*. Version 9.0, May 2022.
  - A dictionary of your choice.
  - IntelliJ and/or Eclipse.
  - Javadoc documentation of Java 11
    `https://docs.oracle.com/en/java/javase/11/docs/api/`

  You are *not* allowed to use any of the following:
  - Solutions of any exercises or old tests published on Canvas;
  - 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. It is recommended that you use IntelliJ and/or Eclipse to write code.
- You do *not* have to add Javadoc or comments, unless explicitly asked to do so. Invariants, preconditions and postconditions should be given only when they are explicitly asked.
- You are not allowed to leave the room during the first 30 minutes or the last 15 minutes of the exam.
- Place your student ID card on the table as well as documentation that grants extra time (if applicable).

Question 1 (20 points)

The questions in the rest of this exam are related to the scenario presented here.

A software company recently discovered that not all their code is great. Some of the code is good, some of it is bad, and some code is just ugly. You are given a task to implement software to record code quality scores of each programmer. An AI is used to classify code into the three categories: the good, the bad and the ugly. After classifying a piece of code of a programmer, the software will invoke the appropriate method of a class that you are going to develop.

Write an interface `CodeScores` that has three commands `good(String name)`, `bad(String name)`, and `ugly(String name)`. These commands are given as the parameter the name of a programmer who wrote some good, bad, or ugly code. Furthermore, add a query `getScore(String name)` that returns the current score of a programmer, or zero if the programmer has not been scored yet, and a query `getCount(String name)` that returns the number of times a programmer was scored, or zero if the programmer has not been scored yet. Finally, a query `getProgrammers()` should return all programmers in the system.

The rules for updating the scores are as follows:
- Good: add 4 points to the score
- Bad: divide the score by 2
- Ugly: subtract 7 points from the score

All implementations of `CodeScores` have to follow these rules.

a. *(5 pts)* Define the interface `CodeScores`, using appropriate keywords and return types.

b. *(6 pts)* Include appropriate Javadoc for the interface and all methods.

c. *(9 pts)* Write JML postconditions for all commands. Write postconditions that describe what changes and postconditions that describe what stays the same.

Answer to question 1 Example code, only including the JML for `good`.

```
/**
 * Interface for recording code quality scores for programmers.
 */
public interface CodeScores {
    /**
     * Record that a programmer has written good code.
     * This adds 4 points to the score of the programmer.
     *
     * @param name The name of the programmer
     */
    //@ensures getScore(name) == \old(getScore(name)) + 4;
    //@ensures getCount(name) == \old(getCount(name)) + 1;
    //@ensures getProgrammers().contains(name);
    //@ensures (\forall String n; getProgrammers().contains(n);
                        n.equals(name) || getScore(i) == \old(getScore(i));
    //@ensures (\forall String n; getProgrammers().contains(n);
                        n.equals(name) || getCount(i) == \old(getCount(i));
    //@ensures (\forall String n; \old(getProgrammers()).contains(n);
                        getProgrammers().contains(n));
    void good(String name);

    /**
     * Record that a programmer has written bad code.
     * This divides the score of the programmer by 2.
     *
```

```
     * @param name The name of the programmer
     */
    void bad(String name);

    /**
     * Record that a programmer has written ugly code.
     * This subtracts 7 points from the score of the programmer.
     *
     * @param name The name of the programmer
     */
    void ugly(String name);

    /**
     * Get the current score for the programmer with the given name.
     *
     * @param name The name of the programmer
     * @return The current score for the programmer,
               or 0 if the programmer has not yet been scored
     */
    int getScore(String name);

    /**
     * Get the number of times the programmer has been scored.
     *
     * @param name The name of the programmer
     * @return The number of times the programmer has been scored,
               or 0 if the programmer has not yet been scored
     */
    int getCount(String name);

    /**
     * Get all programmers in the system.
     * @return All programmers in the system.
     */
    List<String> getProgrammers();
}
```

a. - (+2) Correct interface and requested methods are present
   - (+2) Correct return types for queries: double/int for `getScore`, int for `getCount`, and some list or set or collection of `String` for `getProgrammers`
   - (+1) The commands have **`void`** as return type

b. - (+1) Decent Javadoc on the interface
   - (+5) Decent Javadoc on every method, including `param` with description and `@return` with description (-1 per bad/missing method, half score for Javadoc without `@param` or `@return`)

c. - (+3) 1 point for the change to `getScore`, 1 point for `getCount`, 1 point for `getProgrammers` (postconditions on the commands)
   - (+6) 2 points for `\forall` on a `String`, 2 points for using `getProgrammers().contains()` in some reasonable way, and 2 points for formalizing that all other programmers have the same score and/or count and/or that programmers that were in the system are still in the system.
     The idea was to formalize that after a command, all other programmers' scores and counts would remain the same.

Question 2 (15 points)

a. *(10 pts)* Write a class `CodeScoresImpl` that implements the `CodeScores` interface. Give the full implementation as your answer. Javadoc and JML are not needed.
Hint: first think about what data your class needs to store, i.e., what fields will you need, and then implement the methods.

b. *(5 pts)* Sometimes the keywords **final** and/or **static** are used on fields of a class. Is it appropriate to use one or both of these keywords on the fields you defined in your implementation? Explain your answer for both keywords.

Answer to question 2

```
public class CodeScoresImpl implements CodeScores {
    private final Map<String, Integer> scores = new HashMap<>();
    private final Map<String, Integer> counts = new HashMap<>();

    @Override
    public void good(String name) {
        scores.put(name, getScore(name) + 4);
        counts.put(name, getCount(name) + 1);
    }

    @Override
    public void bad(String name) {
        scores.put(name, getScore(name) / 2);
        counts.put(name, getCount(name) + 1);
    }

    @Override
    public void ugly(String name) {
        scores.put(name, getScore(name) - 7);
        counts.put(name, getCount(name) + 1);
    }

    @Override
    public int getScore(String name) {
        return scores.getOrDefault(name, 0);
    }

    @Override
    public int getCount(String name) {
        return counts.getOrDefault(name, 0);
    }

    @Override
    public Set<String> getProgrammers() {
        return new HashSet<>(scores.keySet());
    }
}
```

The `@Override` annotations are optional. Instead of `getOrDefault` one can also check with `containsKey`. One could explicitly maintain a list of programmers but this is not strictly necessary, since `scores.keySet()` or `count.keySet()` gives the same result.

Example explanation:

- It is appropriate to use the **`final`** keyword on the fields I defined in my implementation. Using **`final`** on the fields `scores` and `counts` would indicate that the reference to the `Map` objects

cannot be reassigned, ensuring that the object that the reference variable is pointing to cannot be changed. This would ensure that the class will always use the same `Map` objects, and prevent any accidental changes to the state of the class.
- It is not appropriate to use the **`static`** keyword on the fields `scores` and `counts` in this implementation. Static fields are shared among all instances of a class, meaning that all instances of the class would share the same `Map` objects, which would cause the class to lose its ability to maintain separate scores and counts if there are multiple objects of the class. Therefore, the fields should not be static.

a.
- (+2) Correct fields and types to store for each name the score and the number of scores: `Map<String,Double>` and `Map<String,Integer>` depending on the chosen return types of the getters. Wrong is using several lists or arrays for score and count with matching indices. A class `Programmer` storing the name, score and count of each programmer could be used, but is not required.
- (+2) All fields are **`private`**.
- (+1) All fields are correctly initialized.
- (+3) Correct implementations of `good`, `bad` and `ugly`.
- (+2) Correct implementations of the queries.

b.
- (+2) Correct choice (+1) and explanation (+1) of **`static`**.
- (+3) Correct choice (+2 for using **`final`**, +1 for not using **`final`** but with a good explanation, 0 points for not using **`final`** without a good reason) and explanation (+1) of **`final`**.

Question 3 (10 points)

It is important that the data collected by your `CodeScoresImpl` class can be stored in a file. Implement a method `void readFromFile(String filename)` that reads a file in the following file format:

```
50,5,Clint Eastwood
-5,13,Lee Van Cleef
30,20,Eli Wallach
```

Here, Lee has a score of -5 and has been scored 13 times.

You do not need to provide a method for writing to a file. You may assume files are correctly written following the file format. Your method should pass exceptions on to the caller. You may assume no scores exist yet in the `CodeScoresImpl` object.

Hint: you can use the `split` method of the `String` class.

Answer to question 3

Example solution:

```
/**
 * Reads data from a file with the scores and counts of the programmers.
 * @param filename The name of the file to read from.
 * @throws FileNotFoundException If the file is not found.
 * @throws IOException If there is an error reading from the file.
 */
public void readFromFile(String filename) throws FileNotFoundException, IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
        String line;
        while ((line = reader.readLine()) != null) {
```

```
            String[] parts = line.split(",");
            int score = Integer.parseInt(parts[0]);
            int count = Integer.parseInt(parts[1]);
            String name = parts[2];
            scores.put(name, score);
            counts.put(name, count);
        }
    }
}
```

Also fine is only throwing `IOException`, since `FileNotFoundException` is a subclass of `IOException`.

This implementation uses a try-with-resources statement to open and read the file, line by line. Each line is split into parts using the `split` method of the `String` class and the comma `,` as the delimiter. The first part is parsed into an integer as the score, the second part as the count, and the third part as the name. Then, it updates the scores and counts of the programmers. The method also throws `FileNotFoundException` and `IOException` on to the caller if these are thrown while reading the file.

- (+2) Use a `FileReader` and/or some other method of parsing the file line by line.
- (+2) Use a method to close the file after reading (either with try-with-resources or explicit close).
- (+2) Correctly parse each line to obtain the three parts.
- (+2) Correctly set the fields of the class according to the read file.
- (+2) Correctly throw an exception if the file is not found and/or if there is an IOException thrown.

Question 4 (15 points)

The company wants to fire the worst programmer, i.e., the programmer with the lowest score.

a. *(5 pts)* Write a method `getWorstProgrammer()` that returns the name of the worst programmer, or `null` if there is no programmer scored yet.

b. *(5 pts)* Add a postcondition to the `getWorstProgrammer` method that describes that the returned name (if not `null`) is the name of the worst programmer.

c. *(5 pts)* One of your friends bribes you to change the software so their name is never returned as the worst programmer. Instead, if your friend is the worst programmer, then the name of the second worst programmer should be returned. Modify the method `getWorstProgrammer` to accommodate their wishes. Store the name of the friend as a constant in the class. Finally, also modify the postcondition.

Answer to question 4

```
//@ ensures (\result != null) ==>
//@                 (\forall String n; getProgrammers().contains(n);
//@                         scores.get(\result) <= scores.get(n));
public String getWorstProgrammer() {
    int minScore = Integer.MAX_VALUE;
    String worstProgrammer = null;
    for (Map.Entry<String, Integer> entry : scores.entrySet()) {
        if (entry.getValue() < minScore) {
            minScore = entry.getValue();
            worstProgrammer = entry.getKey();
        }
    }
```

```
    return worstProgrammer;
}
```

Also fine are solutions that explicitly sort, for example using `Collections.sort` or `List.sort`.

This implementation first initializes the variables `minScore` to the maximum value of an integer and `worstProgrammer` to **`null`**. Then it iterates over the scores of all programmers and compares the score of each programmer to the current minimum score. If the score of the current programmer is lower than the current minimum score, it updates the minimum score and the name of the worst programmer. If the scores map is empty, the method will return **`null`**.

Example solution for part c:

```
private static final FRIEND = "Jimmy";

//@ ensures (\result != null) ==>
//@             (\forall String n; getProgrammers().contains(n);
//@                  !n.equals(FRIEND) ==> getScore(\result) <= getScore(n));
public String getWorstProgrammer() {
    int minScore = Integer.MAX_VALUE;
    String worstProgrammer = null;
    for (Map.Entry<String, Integer> entry : scores.entrySet()) {
        if (!entry.getKey().equals(FRIEND) && entry.getValue() < minScore) {
            minScore = entry.getValue();
            worstProgrammer = entry.getKey();
        }
    }
    return worstProgrammer;
}
```

a. (+5) Return the correct result, e.g. some loop over all scores and remember the worst one, or just use a sort function and return the lowest. Can give partial points depending on how close the answer is to a correct solution.

b.
- (+1) for the `\result != null` guard
- (+2) for use of a `\forall` on either the result of `getProgrammers()` or directly on the keys of one of the maps
- (+2) for `getScore(\result) <= getScore(n))` or something equivalent

c.
- (+1) for the constant: **`static`** and **`final`** (may be **`public`** or **`private`**)
- (+2) for the implementation that excludes returning the name of the friend
- (+2) for the modification to the JML, e.g., adding another guard `!n.equals(FRIEND)`

Question 5 (10 points)

a. *(5 pts)* If the person does not exist when invoking the `getScore` or `getCount` methods, an exception `ProgrammerNotFound` should be thrown. Create this exception and modify `getScore` and `getCount` to correctly throw the exception. Give the exception and the new implementations of those two methods as your answer.

b. *(5 pts)* Exceptions in Java can be "checked" and "unchecked". Explain what the difference is between checked and unchecked, explain whether the exception you made is checked or unchecked, and explain your design decision of a checked or unchecked exception.

Answer to question 5

Example code:

```
public class ProgrammerNotFound extends Exception {
    public ProgrammerNotFound(String name) {
        super("Programmer␣" + name + "␣not␣found.");
    }
}

@Override
public int getScore(String name) throws ProgrammerNotFound {
    if (!scores.containsKey(name)) {
        throw new ProgrammerNotFound(name);
    }
    return scores.get(name);
}

@Override
public int getCount(String name) throws ProgrammerNotFound {
    if (!counts.containsKey(name)) {
        throw new ProgrammerNotFound(name);
    }
    return counts.get(name);
}
```

This implementation defines a new `Exception` class called `ProgrammerNotFound` that takes the name of the programmer as a parameter in its constructor. The `getScore` and `getCount` methods check if the programmer is in the system by checking if the name is present in the scores and counts `Map`s, respectively. If the programmer is not found, it throws the `ProgrammerNotFound` exception.

The difference between checked and unchecked exceptions in Java is that checked exceptions must be handled or declared by the programmer, whereas unchecked exceptions do not have to be. Checked exceptions are typically used for exceptional conditions that a well-written program should anticipate and be able to handle, such as a missing file. Unchecked exceptions are typically used for programming errors or unexpected conditions that are outside the control of the program, such as a null pointer exception. The exception that I made, `ProgrammerNotFound`, is a checked exception. The decision to make the exception checked was based on the idea that this exception is thrown when the programmer is not found, but it is a condition that the program can anticipate and handle.

In this case, making the exception checked allows the developer to handle the exception by catching it and providing an appropriate action such as notifying the user or trying to recover the data. Also, this allows the developer to see the potential issue in the code by seeing the exception in the signature of the methods and prepares them to handle it.

a. - (+2) A class with the correct name extending the `Exception` class (or `RuntimeException` if unchecked).
   - (+1) There is a sensible message, either in the constructor or defined in the `getScore` and `getCount` methods
   - (+2) Correct modification of `getScore` and `getCount`.

b. - (+2) Correct explanation of difference between checked and unchecked. A common mistake is to believe that a thrown exception is unchecked and a caught exception is checked; this is of course wrong.
   - (+1) Correct choice for checked exception.
   - (+2) Correct explanation of the design decision. This should answer “why choose to force the programmer to handle the exception”.

Question 6 (10 points)

The company wants to use multiple AIs to classify code simultaneously, i.e., using multiple threads.

a. *(5 pts)* Consider that there are multiple threads of AIs scoring programmers. They will invoke methods of the same `CodeScores` object from different threads. Give a concrete example of what can go wrong, and in your answer, explicitly mention the invoked methods, the expected result and a possible wrong result, explaining how this result can occur.

b. *(5 pts)* Make your `CodeScoresImpl` implementation thread safe such that no race conditions can occur. Explain why the example in the previous question is no longer possible.

Answer to question 6

Example answers:

a. For example, consider two threads, Thread A and Thread B, both invoking the `good(String name)` method of the same `CodeScores` object on the same programmer, "John Smith", at the same time. The expected result is that the score of "John Smith" increases by 4 points. However, a possible wrong result is that the score of "John Smith" increases by only 2 points. This can occur because Thread A and Thread B both retrieve the current score of "John Smith" at the same time, both add 4 to the score, and then both update the score at the same time. The problem is that the score of "John Smith" is only updated once instead of twice, resulting in the score only increasing by 4 points instead of 8 points.

b. One way to make the `CodeScoresImpl` implementation thread safe is to use the **`synchronized`** keyword on the methods that update the scores and counts. This ensures that only one thread can access the methods at a time, preventing race conditions from occurring. That is, only one thread can read and write to the fields of the class at a time, preventing all race conditions on the fields of the class.

Grading:

a. (+5) for good explanation, award points based on detail level. Full points when the explanation details that the value is first read by both threads, then locally updated, then the updated value is written by both threads, thus overwriting the first update by the second one. A (potential) problem that would also work for non-concurrency situations, like a different order good/bad/ugly resulting in different scores, is not race condition.

b. (+5) for an explanation and the updates to the code. Only give 2 points if **`synchronized`** is applied without any (good) explanation. If a `ReentrantLock` is used, then the `unlock()` call should occur inside a **`finally`** block, as otherwise the lock would stay locked if an exception occurred inside the method.

Question 7 (10 points)

The company wants to automatically fire anyone when they get a negative score. To do this, they run a separate thread that waits until a programmer has a negative score and then immediately fires them.

Add a method `String waitForNegative()` that returns the name of the next programmer that gets a negative score. This method will be called from a separate thread repeatedly. The method should not return if there is no programmer with a negative score (yet).

Give all changed methods and the new `waitForNegative` method as your answer.

Answer to question 7

Example code:

```
/**
 * Returns the name of the next programmer that gets a negative score.
 * The method will not return until there is a programmer with a negative score.
```

```
 * @return The name of the next programmer that gets a negative score.
 */
public synchronized String waitForNegative() {
    while (true) {
        for (Map.Entry<String, Integer> entry : scores.entrySet()) {
            if (entry.getValue() < 0) {
                return entry.getKey();
            }
        }
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

@Override
public synchronized void ugly(String name) {
    scores.put(name, scores.getOrDefault(name, 0) - 7);
    counts.put(name, counts.getOrDefault(name, 0) + 1);
    notifyAll();
}
```

This implementation uses a **`while`** loop that continually checks for a programmer with a negative score. If there is no programmer with a negative score, it uses the `wait()` method to wait for a notification that a score has been updated. When a score is updated, the `notifyAll()` method is called to notify all waiting threads that a score has been updated. Only `ugly` needs to call `notifyAll()`.

- (+2) Correctly use `notifyAll()` while in a mutex (in `ugly`)
- (+3) Use `wait()` inside a **`while`** loop
- (+3) Use `wait()` inside the same mutex as `notifyAll`
- (+2) Return the name of the programmer with a negative score.

The point here is that a student should recognize that the appropriate way to let a thread wait until a condition is true, is to use the wait-notify mechanism.

Question 8 (10 points)

The company wants a website where the programmers of the company can see their current scores. For this, user authentication is required.

a. *(4 pts)* Describe two approaches that can be used for user authentication; one sentence per approach is enough.

b. Below is some Java code that has a vulnerability.

```
@RequestMapping("/register")
public void registerUser(@RequestBody UserInfo user) throws DatabaseException {
    try {
        // open a connection with the database
        database.openConnection();
        // store the given information of the user
        database.saveUser(user.email, user.password);
    } catch (DatabaseException ex) {
        // just throw it again, but ensure the connection is closed
        throw ex;
```

```
    } finally {
        // ensure the connection is closed even after exceptions
        database.closeConnection();
    }
}
```

The above code is an endpoint that directly receives an email address and a password from the request body, i.e., from the browser of the user, via a secure connection.

(6 pts) Identify a vulnerability and give a method to overcome this vulnerability.

Answer to question 8

a. 1 point for mentioning the approach and 1 point for the explanation. A few sample solutions (listing any two is acceptable):
   - Passwords. The user types in her secret password to prove her identity.
   - Cryptographic authentication protocols. The user has a secret key and authenticates herself using a crypto protocol (e.g., SSL).
   - Biometrics. The user displays her fingerprint, hand geometry, iris scan, etc., so that the computer can verify that it is really her.
   - Hardware tokens. The user owns a hardware device which displays a new one-time password every minute, and the user types in the current one-time password so that the server can verify its correctness

b. The security risks in this code snippet are:
   - Weak password policy – There is no check on the length of a password or even to check if the password is strong enough.
   - Cleartext storage of sensitive data – A password is stored directly in the database without hashing or salting the password.
   - Password can be overwritten – There is no check whether the user already exist.
   - Input is not sanitized – Maybe SQL injection is present.

   3 points for the security risk and 3 points for a way to overcome it.