TEST

Software Systems: Programming

course code: 201700117
date: 22 January 2018
time: 8:45 – 11:45

SOLUTIONS

General

- While making this, you may use the following (unmarked) materials:
  - the reader;
  - the slides of the lectures;
  - the books which are specified as course materials for the module (or copies of the required pages of said books);
  - a dictionary.

  You may *not* use any of the following:
  - solutions of any exercises published on Blackboard (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.
- 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.
- The grade for this test is used in calculating the final grade of the module. The formula used to do so can be found in the reader.

Question 1 (10 points)

Consider the following (standard Java) interface:

```
public interface Comparable<T> {
    /**
     * 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 object given as parameter.
     * @param o the object to be compared.
     */
    int compareTo(T o);
}
```

For instance, the standard Java class `String` implements `Comparable`, and so, for instance,

- `"abc".compareTo("xyz")` returns a negative number, because the `"abc"` comes before `"xyz"` in the standard lexicographical ordering;
- `"one".compareTo("one")` returns zero.

a. *(3 points)* What does the `T` stand for, in the first line and in the declaration of `compareTo`? What is its purpose?

b. *(5 points)* Modify the class `Date`, given below, so that it implements `Comparable`, in such a way that one date is considered smaller than another if it occurred earlier than that other. *(You only have to indicate what needs to be changed or added to the declaration below, not copy the entire class.)*

```
public class Date {
    private int year, month, day;

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

    public int getYear() {
        return this.year;
    }

    public int getMonth() {
        return this.month;
    }

    public int getDay() {
        return this.day;
    }
}
```

c. *(2 points)* Give a (**boolean**) expression, using your extended `Date` class, that tests whether 22 January 2018 lies before 1 February 2018. *(You do not have to give the outcome, just the expression itself.)*

Answer to question 1

a. *(3 points; up to −2 if the term "type parameter" or similar is not used; up to −2 if it is not mentioned that `T` is substituted by an actual type.)* `T` is a *type parameter*. When `Comparable` is used, a concrete type will be specified that takes the place of `T` in that instance. For instance, in

the type `Comparable<String>`, `T` is instantiated by `String` and so the method has the actual signature `compareTo(String)`.

b. *(5 points; −2 if the header change is omitted, −1 if only the type parameter `<Date>` is omitted, up to −3 for errors in the method body.)* The header should be changed into

```
public class Date implements Comparable<Date> {
```

and the following method should be added:

```
    public int compareTo(Date o) {
        int result = this.year - o.year;
        if (result != 0) {
            return result;
        }
        result = this.month - o.month;
        if (result != 0) {
            return result;
        }
        result = this.day - o.day;
        return result;
    }
```

c. *(2 points; no points if the comparison with 0 is omitted; 0 or 1 point (depending on severity) if the `Date`s are constructed incorrectly or `compareTo` is invoked incorrectly.)*

```
new Date(22,1,2018).compareTo(new Date(1,1,2018)) < 0
```

Question 2 (20 points)

Consider the following class:

```
/** Test within a course or module. */
public class Test {
    private final String topic;
    private final Date date;

    public Test(String topic, Date date) {
        this.topic = topic;
        this.date = date;
    }

    /** Returns the topic that was tested. A topic is part of a course or module;
      * two different courses never have topics with the same name.
      */
    public String getTopic() {
        return this.topic;
    }

    /** Returns the date this test was taken. */
    public Date getDate() {
        return this.date;
    }
}
```

a. *(12 points)* Define an interface `Grade` that provides the following *JML-specified* methods:

- `getSNumber`, which returns the 7-digit s-number (as an **`int`**) of the student who obtained the grade;
- `getDate`, which returns the date that this grade was assigned;
- `getValue`, which returns the point value of the grade, which is a **`double`** between 1.0 and 10.0 inclusive, with a fractional part consisting of at most one decimal. *(To test whether a* **`double`** *value has a non-zero fractional part, you can compare it with the same value coerced to an* **`int`***; for instance,* `x == (int) x` *for a* **`double`** *variable* `x` *returns* **`true`** *if the value of* `x` *has no decimal.)*

Do not forget to add the requested JML specification (which should be as precise as possible)!

b. *(8 points)* Define a class `TestGrade` that implements `Grade` and additionally maintains the `Test` for which this is a `Grade`, and (besides implementing the methods of `Grade`) provides the following *JML-specified* elements:

- A constructor, which should initialise all fields.
- A method `getTest`, which returns the `Test` for which this is a `Grade`.
- A method `setValue`, which changes the point value of the grade. A grade should only be changed to a higher value.

Do not forget to add the requested JML specification (which should be as precise as possible)!

Answer to question 2 *The division of points is, with hindsight, a bit out of balance: it would have been better for both parts to get 10 points.*

a. *(12 points, of which 4 for correct Java and 8 for correct JML.*

- -1 *for each Java error, e.g. wrong parameter or return types. Do not deduct points for minor errors such as semicolons.*
- -1 *for each omitted* `pure` -1 *for errors;*
- -2 *for each omitted* `ensures`*, except the last (of* `getValue`*)*
- -3 *for omitting the last* `ensures` *(of* `getValue`*)*
- *Max.* -3 *for spurious JML*
- -2 *for other (independent) errors).*

```
public interface Grade {
    /*@
     * pure
     * ensures \result >=0 && \result < 10000000;
     */
    int getSNumber();

    /*@
     * pure
     * ensures \result != null;
     */
    Date getDate();

    /*@
     * pure
     * ensures \result >= 1.0 && \result <= 10.0;
     * ensures \result * 10 == (int) (\result * 10);
     */
    double getValue();
}
```

b. *(8 points, of which 2 for correct Java and 6 for correct JML.*

- -1 *for each Java or JML error or omission.*
- *Do not deduct points for minor errors such as semicolons, or for repeat errors, i.e., ones that were essentially already made in the first part.*
- *Max.* -2 *for Java errors,* -3 *for errors in the constructor contract,* -2 *in the contract of* `getTest` *and* -3 *in the contract of* `setValue`
- *In the constructor contract, deduct at most* -1 *if the idea is essentially there and one or two conditions were forgotten.*

```
public class TestGrade implements Grade {
    private final Test test;
    private final int sNumber;
    private double value;

    /*@
     * requires test != null;
     * requires sNumber >= 0 && sNumber < 10_000_000;
     * requires value >= 1.0 && value <= 10.0;
     * requires 10 * value == (int) (10 * value);
     * ensures getTest() == test && getDate() == test.getDate();
     * ensures getSNumber() == sNumber && getValue() == value;
     */
    public TestGrade(Test test, int sNumber, double value) {
        this.test = test;
        this.sNumber = sNumber;
```

```
        this.value = value;
    }

    public Date getDate() {
        return getTest().getDate();
    }

    /*@
     * pure
     * ensures \result != null
     */
    public Test getTest() {
        return this.test;
    }

    public int getSNumber() {
        return this.sNumber;
    }

    public double getValue() {
        return this.value;
    }

    /*@
     * requires value > \old.getValue() && value <= 10.0;
     * requires 10 * value == (int) (10 * value);
     * ensures getValue() == value;
     */
    public void setValue(double value) {
        this.value = value;
    }
}
```

Question 3 (25 points)

a. *(3 points)* Program an enumerated type `TestMode`, with values `WRITTEN` (which stands for a written test), `ORAL` (which stands for an oral examination) and `PROJECT`.

b. *(7 points)* A *course* consists of a name and a set of topics. Every topic has an associated test mode (of type `TestMode`), and an associated relative weight (a positive **`int`**) that determines which fraction of the final course grade is made up by that topic. (See subquestion c below for an example.) Part of `Course` is given below; complete it by programming the constructor and the method `addTopic`, including JML specifications that are as complete as possible. *(Your answer need only consist of the constructor and method with JML-specifications; you can omit the predefined part of the class.)*

```
public class Course {
    private final String name;
    private final Map<String, Integer> weights;
    private final Map<String, TestMode> modes;

    /** Constructs a course with a given name, initially without topics. */
    /*@
     * requires name != null;
     * ensures getName() == name;
     * ensures weights.isEmpty() && modes.isEmpty();
     */
    public Course(String name) {
        // fill in
    }

    /** Returns a map from topics to their relative weight. */
    public Map<String, Integer> getWeights() {
        return this.weights;
    }

    /** Returns a map from topics to their corresponding test mode. */
    public Map<String, TestMode> getModes() {
        return this.modes;
    }

    /** Returns the name of this course. */
    public String getName() {
        return this.name;
    }

    /** Adds a topic to this course. The topic should be fresh. */
    public void addTopic(String topic, int weight, TestMode mode) {
        // fill in
    }

}
```

c. *(5 points)* Program an additional method `getModeTopics` for `Course`, which has no parameters and returns a map from `TestMode`s to those topics tested in those modes. For instance, course `"Software␣Systems"` has topics

- `"Design␣theory"` with mode `WRITTEN` and weight 1
- `"Design␣practice"` with mode `PROJECT` and weight 2
- `"Programming␣theory"` with mode `WRITTEN` and weight 1
- `"Programming␣practics"` with mode `PROJECT` and weight 3

- `"Mathematics"` with mode `WRITTEN` and weight 2

(note that these are not the actual weights of the module!) then `getModeTopics().get(WRITTEN)`, when invoked on that `Course`, should return a `Set` containing the topics `"Design␣theory"`, `"Programming␣theory"` and `"Mathematics"`. *(No specification or comments need be provided for this method.)*

d. *(10 points)* Program a class `CourseGrade` that is constructed using a `Course` and a non-empty `Set` of `TestGrade`s. You may assume that all `TestGrade`s concern the same student, and that the set contains exactly one grade for each topic of the `Course`.

- `getValue` in this case should return a number without decimals, which is the rounded, weighted average of the topic grades, where the weights are given by the `Course`.
- `getSNumber` returns the student to whom all `TestGrade`s belong.
- `getDate` returns the date of the latest `TestGrade`.

(No specification or comments need be provided for this class.)

Answer to question 3

a. *(3 points. The question is quite simple, so an error will soon give rise to 0 points. Wrong interpunction (commas) should not be punished.)*

```
public enum TestMode {
    WRITTEN, ORAL, PROJECT;
}
```

b. *(7 points: 2 for the constructor, 3 for `addTopic` and 3 for the JML.)*

```
    /*@
     * requires name != null;
     * ensures getName() == name;
     * ensures weights.isEmpty() && modes.isEmpty();
     */
    public Course(String name) {
        this.name = name;
        this.weights = new HashMap<>();
        this.modes = new HashMap<>();
    }
```

Note that to be really complete, the contract of `addTopic` should also specify that the *rest* of the maps `weights` and `modes` is unchanged. However, this is beyond what we can expect the students to write.

```
    /*@
     * requires topic != null && weight > 0 && mode != null;
     * requires !weights.containsKey(topic);
     * ensures weights.get(topic) == weight && modes.get(topic) == mode;
     */
    public void addTopic(String topic, int weight, TestMode mode) {
        this.weights.put(topic, weight);
        this.modes.put(topic, mode);
    }
```

c. *(5 points. Since it is difficult to predict which errors will occur, a scale should be devised during correction.)* The solution below is not the only one possible: one may also choose to initialise `result` for a given `TestMode` value only when that value exists in `modes`. However, the result is different, for then `result` may be only partial: for instance, if no `ORAL` test mode occurs, then the solution below will map `ORAL` to the empty set, whereas the alternative will not have it as a key and hence map it to **`null`**. The question is not totally clear about which of these is asked for, so they are both correct.

```
public Map<TestMode, Set<String>> getModeTopics() {
    Map<TestMode, Set<String>> result = new HashMap<>();
    for (TestMode m : TestMode.values()) {
        result.put(m, new HashSet<>());
    }
    for (Map.Entry<String, TestMode> e : getModes().entrySet()) {
        result.get(e.getValue()).add(e.getKey());
    }
    return result;
}
```

d. *(10 points. Deduct at most -5 for errors in `getValue`, -3 for `getSNumber` and -4 for `getDate`. Since it is difficult to predict which errors will occur, a scale should be devised during correction.)*

```
public class CourseGrade implements Grade {
    private final Course course;
    private final Set<TestGrade> tests;

    public CourseGrade(Course course, Set<TestGrade> tests) {
        this.course = course;
        this.tests = tests;
    }

    public Course getCourse() {
        return this.course;
    }

    public double getValue() {
        double result = 0;
        int totalWeight = 0;
        for (TestGrade g : this.tests) {
            int weight = this.course.getWeights().
                    get(g.getTest().getTopic());
            result += g.getValue() * weight;
            totalWeight += weight;
        }
        return (int) (result / totalWeight + 0.5);
    }

    public Set<TestGrade> getTests() {
        return this.tests;
    }

    public int getSNumber() {
        return this.tests.iterator().next().getSNumber();
    }
```

```
    public Date getDate() {
        Date result = null;
        for (TestGrade g : this.tests) {
            Date d = g.getDate();
            if (result == null || result.compareTo(d) < 0) {
                result = d;
            }
        }
        return result;
    }
}
```

Question 4 (15 points)

a. *(3 points)* Declare a new exception class `TopicException`, which will be used to indicate that the same topic occurs twice in a `Course`. Reuse the existing mechanism to construct exceptions with dedicated messages.

b. *(7 points)* Change `Course.addTopic` as programmed in Question 3.b so that it throws

- a `TopicException` if the topic that is (attempted to be) added already occurs in the course;
- an `IllegalArgumentException` (which is a standard Java subclass of `RuntimeException`) if either the parameter `weight` or the parameter `mode` is not a legal value.

In either case, the thrown exception should have a decent error message. (The standard `IllegalArgumentException` has a constructor that takes the message as a `String` parameter.) *(Your answer should consist of the entire new `addTopic` declaration.)*

c. *(5 points)* Write a code fragment that constructs a course `"Software␣Systems"` with the topics as given in Question 3.c, handling (only) those exceptions that must be handled in order for the code to compile. When an exception is thrown, its message should be printed on the standard output. Explain your choice of error handling. *(To save writing, it is enough if in your answer you add only a single topic to the course, rather than all five.)*

Answer to question 4

a. *(3 points. The question is quite simple, so an error will soon give rise to 0 points. Wrong interpunction should not be punished.)*

```
public class TopicException extends Exception {
    public TopicException(String message) {
        super(message);
    }
}
```

b. *(7 points; −2 for a missing **`throws`** declaration, up to −3 for each thrown exception.)*

```
    public void addTopicX(String topic, int weight, TestMode mode)
            throws TopicException {
        if (this.weights.containsKey(topic)) {
            throw new TopicException("Topic␣" + topic +
                    "␣already␣exists");
        }
        if (weight < 1) {
            throw new IllegalArgumentException("Weight␣" + weight +
                    "␣should␣be␣positive");
        }
        if (mode == null) {
            throw new IllegalArgumentException("Test␣mode␣is␣null");
        }
        this.weights.put(topic, weight);
        this.modes.put(topic, mode);
    }
```

c. *(5 points; −2 if the `IllegalArgumentException` is also caught; up to −3 for errors in the **`try-catch`**-construct.)* You only need to catch the `TopicException`, as the other one is a `RuntimeException` that does not lead to a compilation error.

```
Course c = new Course("Software␣Systems");
try {
    c.addTopicX("Design␣Theory", 1, TestMode.WRITTEN);
} catch (TopicException e) {
    System.out.println(e.getMessage());
}
```

Question 5 (15 points)

Consider the following three classes:

```
public class Restaurant {
    private final int seats;
    private int reserved;

    public Restaurant(int seats) {
        this.seats = seats;
    }

    public boolean reserve(int number) {
        boolean result = this.seats - this.reserved >= number;
        if (result) {
            this.reserved = this.reserved + number;
        }
        return result;
    }

    public int getReserved() {
        return this.reserved;
    }
}
```

```
public class Group extends Thread {
    private final int size;
    private final Restaurant rest;
    private boolean success;

    public Group(Restaurant rest, int size) {
        this.size = size;
        this.rest = rest;
    }

    @Override
    public void run() {
        this.success = this.rest.reserve(this.size);
    }

    public boolean isSuccess() {
        return this.success;
    }
}
```

```
1  public class Booking {
2      public static void main(String[] args) {
3          Restaurant stek = new Restaurant(5);
4          Group yolo = new Group(stek, 4);
5          Group tegel = new Group(stek, 3);
6          yolo.start();
7          tegel.start();
8          try {
9              yolo.join();
10             tegel.join();
11         } catch (InterruptedException exc) {
12             // cannot occur
13         }
14         System.out.printf("Y:␣%s;␣T:␣%s;␣S:␣%s%n",
15                 yolo.isSuccess(), tegel.isSuccess(), stek.getReserved());
16     }
17 }
```

a. (*6 points*) What are the possible outcomes printed by the `main` method of `Booking`? Which of them are erroneous? Explain your answer.

b. (*4 points*) If we remove the entire **`try`**-block from `Booking` (lines 8–13), do the possible printed outcomes change, and if so, how? Explain your answer.

c. (*5 points*) In the original system (with **`try`** block), how can you modify `Restaurant` such that the erroneous outcomes can no longer occur? Explain why your solution works.

Answer to question 5

a. *(6 points; up to −3 for omitting or failing to explain any of the first three, −1 for omitting or failing to explain any of the last two.)* There are quite a number of possibilities:

- `Y: true; T: false; S: 4` if the Yolo transaction was processed completely before the Tegel transaction. This is correct behaviour.
- `Y: false; T: true; S: 3` if the Tegel transaction was processed completely before the Yolo transaction. This is correct behaviour.
- `Y: true; T: true; S: 7` if the calculation of `result` for both transactions was computed before the value of `reserved` was updated, and then the update was performed in proper interleaving fashion for both. This is wrong behaviour.
- `Y: true; T: true; S: 4` if the calculation of `result` for both transactions was computed before the value of `reserved` was updated, and the updates of `reserved` also interfered, where the one of Yolo came last. This is wrong behaviour.
- `Y: true; T: true; S: 3` if the calculation of `result` for both transactions was computed before the value of `reserved` was updated, and the updates of `reserved` also interfered, where the one of Tegel came last. This is wrong behaviour.

b. *(4 points; award full points if there is a decent explanation, even if the answer is incomplete. Repeat errors should not be counted twice.)* Yes, there are now even more options:

- `Y: false; T: false; S: 0`, if the main thread reaches the `printf` statement before anything happened in the spwaned threads;
- Any combination of `true` or `false` for `Y`, `true` or `false` for `T`, and `3`, `4` or `7` for `S`; i.e., $2 \times 2 \times 3 = 12$ options in total, of which 5 were already listed in a above. This is due to the fact that the assignment to `success` in either thread may not yet have occurred when the

`printf` is executed, in combination with the scenarios that led to the last three options in a.

c. *(5 points)* It is enough to make the method `reserve` in `Restaurant` **`synchronized`**, as both threads invoke this method on the same object and it covers the entire transaction.

Question 6 (15 points)

Imagine an online platform for table reservations at restaurants. Suppose now that after a few years of successful service, the owner of the platform is starting to worry a bit more about the security of his website.

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?

Both restaurant owners and customers can have accounts on this website and of course the accounts need to be properly protected; people should not be able to see or modify each other's reservations. A password is one of the ways a computer server can determine whether the person on the other end of the line is actually the person he or she claims to be. However, passwords do have some downsides.

b. *(2 points)* List at least two such downsides.

Authentication methods are typically subdivided into three categories, also called "authentication factors". The aforementioned password fits into the category, "something the user *knows*".

c. *(4 points)* What are the other two categories? Also provide at least one example for each of them.

Before customers can make a reservation, they need to create an account. Upon enrollment, new users are sent an initial password. You notice that the 12-13 character passwords have the following pattern:

- First, all passwords start with the text "pw",
- this is followed by two or three digits,
- after which come 8 characters from the set {"x", "p", "f"}.

d. *(3 points)* If an attacker would try to brute-force access to an account with such an initial password, how many attempts would it at most take? Show your calculation.

Of course the passwords of the users should not be stored in the database in "plaintext".

e. *(3 points)* Which of the following approaches is best for storing user passwords? Explain your answer.

A. Base64
B. PBKDF2
C. HMAC
D. SHA256

Answer to question 6

a. *(3 points)* CIA: Confidentiality, Integrity, Availability.

b. *(2 points)* Password re-use across domains, multiple passwords are hard to manage, passwords are easy to shoulder-surf or phish, people often choose easy to guess bad passwords.

c. *(4 points)*

- Something the user *has*. Examples: bank card, smart card, token, RSA token, yubikey.

- Something the user *is* (biometrics). Examples: fingerprint scanner, voice recognition, iris scan.

d. *(3 points)* $1 \times (10^2 + 10^3) \times 3^8 = 7217100$

e. *(3 points)* B. (Base64 is an encoding scheme, HMAC is a keyed hash function and not suitable, SHA256 is a hash function built for speed)