TEST

Software Systems: Programming

course code: 201700117
date: 20 January 2020
time: 8:45 – 11:45

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.
- 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 real state offers.

a. *(5 points)* Define an interface called `Property` with methods to access the following attributes of a *real estate property* (e.g., a house or an apartment): *price*, *address*, *number of rooms*, *living area* and *total area*. Describe minimal reasonable preconditions and postconditions for these methods and explain your choices in the comments.

b. *(5 points)* Suppose an *apartment* is defined as a real estate property where the living area is equal to the total area (i.e., no garden, like a flat apartment). Implement a class called `Apartment` that implements `Property` and define a suitable constructor for this class.

c. *(5 points)* You should know that a **`public`** instance variable can be accessed and modified by the entire program. What is the visibility scope of an instance variable with the **`protected`** modifier, i.e., which objects are allowed to access this variable?

d. *(5 points)* A *studio* is defined as an apartment with a single room. Implement a class called `Studio` that extends `Apartment` and define a constructor for this class.

Answer to question 1

a. (5 points) Grading criteria:

- 1 point: no instance variables (because **`interface`**).
- 2 points: getters for all five property with reasonable return types.
- 2 points: postconditions close to standard answers. Java boolean expressions are not required with postconditions are properly defined informally (in text).

```
public interface Property {

    /**
     * @ensures result >= 0
     */
    public double getPrice();

    /**
     * @ensures result != null
     */
    public String getAddress();

    /**
     * @ensures result >= 1
     */
    public int getRooms();

    /**
     * @ensures result >= 0
     */
    public double getTotalArea();

    /**
     * @ensures result >= 0 &&
     *      result <= this.getTotalArea()
     */
    public double getLivingArea();
```

```
}
```

b. (5 points) Grading criteria:

- 1 point: four or five **`private`** instance variables. Also acceptable to have separate instance variables for total and living areas, if the methods are properly implemented.
- 3 points: all getters of `Property` are properly implemented.
- 1 point: a **`public`** constructor with parameters that sets all instance variables.

```
public class Apartment implements Property {

    private double price;
    private String address;
    private int rooms;
    private double area;

    public Apartment () {
    }

    public Apartment (double price, String address, int rooms, double area) {
        this.price = price;
        this.address = address;
        this.rooms = rooms;
        this.area = area;
    }

    public double getPrice() {
        return price;
    }

    public String getAddress() {
        return address;
    }

    public int getRooms() {
        return rooms;
    }

    public double getLivingArea() {
        return area;
    }

    public double getTotalArea() {
        return area;
    }

}
```

c. (5 points) Protected instance variables of a class `C` are accessible for: all subclasses of `C` and all classes in the same package as `C`.

Grading criteria:

- 3 points: mentioned subclasses.
- 2 points: mentioned same package.

d. (5 points) Grading criteria:

- 1 point: class declaration is correct (with **`extends`**).
- 1 point: no new instance variables are defined
- 1 point: a constant for the number of rooms is defined.
- 2 points: **`super`** is properly used in the constructor. Give only 1 point if all original instance variables of `Apartment` are properly initialised in the constructor instead of using **`super`**.

```
public class Studio extends Apartment {
    private static final int NUMBER_OF_ROOMS = 1;

    public Studio(double price, String address, double area) {
        super(price, address, NUMBER_OF_ROOMS, area);
    }
}
```

Question 2 (25 points)

Below you can find a class `Portfolio` that keeps track of real estate offers:

```
public class Portfolio {
    private Set<Property> offers = new HashSet<Property>();
    private int value;

    /** Returns all properties from the offer. */
    public Set<Property> getOffers() {
        return offers;
    }

    /** Returns the value of the portfolio. */
    public int getValue() {
        return value;
    }

    /**
     * Returns a map containing the properties grouped according
     * to the number of rooms.
     */
    public Map<Integer, Set<Property>> groupInRooms() {
        // To be implemented
    }

    /**
     * Returns a sorted list of Properties in this portfolio
     * from low to high price.
     */
    public List<Property> sortByPrice() {
        // To be implemented
    }

    public void addProperty (Property p) {
        offers.add(p);
        value += p.getPrice();
    }
}
```

a. *(10 points)* Implement the method `groupInRooms`. This method returns a `Map` relating the number of rooms to the properties, i.e., given a key `rooms` representing the number of rooms, with `groupInRooms().get(rooms)` all properties with `rooms` rooms can be obtained.

b. *(8 points)* On page 420 of Eck's book (Section 8.5) you can find the `simpleBubbleSort` algorithm to sort an array. Implement the method `sortByPrice`, which returns an `ArrayList` with the contents of the portfolio, but where the portfolio is sorted by price from lowest to highest. You are expected to use methods like `get(int i)` and `set(int i, E element)` of the `List<E>` interface to move elements around according to the algorithm. On page 499 of Eck's book you can find information on these methods.

c. *(7 points)* Write a JUnit test `testSortByPrice` for the method `sortByPrice` that tests whether the result list is sorted and that each offer in the portfolio is also in the sorted result list. Your test case should detect that the following implementation of `sortByPrice` is incorrect:

```
public List<Property> sortByPrice() {
    List<Property> result = new ArrayList<Property>();
    Property oneProperty = getOffers().iterator().next();
    for (int i=0; i < getOffers().size(); i++) {
        result.add(oneProperty);
    }
    return result;
}
```

Answer to question 2

a. (10 points) Grading criteria:

- 1 point: correct return type: `Map<Integer, Set<Property>>`
- 1 point: instantiated correct implementation: `HashMap<Integer, Set<Property>>` or `HashMap<>` (modern Java) or for example `HashMap<Integer, List<Property>>` if they made a mistake with the return type.
- 2 points: defined a loop to iterate over the `offers` instance variable.
- 2 points: properly added the element in each iteration (in both cases).
- 1 point: created new entry if entry for `rooms` did not exist.
- 2 points: new entry is a suitable implementation of the value type of the map, that is, `HashSet<Property>` (or if they made an error before, then `LinkedList<Property>` or `ArrayList<Property>`).
- 1 point: properly returned the `Map`.

```
public Map<Integer, Set<Property>> groupInRooms() {
    Map<Integer, Set<Property>> result =
            new HashMap<Integer, Set<Property>>();
    for (Property p : offers) {
        int rooms = p.getRooms();
        if (result.containsKey(rooms)) {
            result.get(rooms).add(p);
        } else {
            Set<Property> properties = new HashSet<Property>();
            properties.add(p);
            result.put(rooms, properties);
        }
    }
    return result;
}
```

b. (8 points) Grading criteria:

- 1 point: correct return type: `List<Property>` or `ArrayList<Property>`
- 1 point: correct initialization of the list `result` (initialisation and copying of `Property` objects from `getOffers()`)
- 2 points: two nested for loops (as in the referenced code of the book)
- 1 point: using `get` and `getPrice` to obtain the price for the comparison
- 1 point: the comparison has the correct operator `>` (as in the referenced code)
- 2 points: correct implementation of swapping positions `j` and `j+1`

```
    public List<Property> sortByPrice() {
        List<Property> result = new ArrayList<Property>();
        result = new ArrayList<Property>();
        for (Property o : getOffers())
            result.add(o);
        for (int i = 0; i < result.size(); i++) {
            for (int j = 0; j < result.size() - 1; j++) {
                if (result.get(j).getPrice() > result.get(j + 1).getPrice()) {
                    Property temp = result.get(j);
                    result.set(j, result.get(j + 1));
                    result.set(j + 1, temp);
                }
            }
        }
        return result;
    }
```

c. (7 points) Grading criteria:

- 1 point: answer uses the `@Test` annotation and sensible `assert*` methods
- 1 point: reasonable setup of a portfolio with at least 3 properties with different prices
- 1 point: answer does not use the instantiated Property objects directly to check the result. A wrong answer would be storing the objects in local variables and checking if the local variables are in the correct place in `testResult`.
- 2 points: correctly test if the result is sorted using a **`for`** loop checking each successive pair
- 2 points: correctly test that the result list is a permutation of the set of offers, that is, the same size (1 point) and the same items (1 points). Award no points if they only test that each element of the list is in the set; this is wrong.

```
class PortfolioTest {

    Portfolio pf;

    @BeforeEach
    void setUp() throws Exception {
        pf = new Portfolio();
        pf.addProperty(new Studio(12, "", 10));
        pf.addProperty(new Apartment(23, "", 4, 15));
        pf.addProperty(new Apartment(4, "", 2, 3));
        pf.addProperty(new Studio(0, "", 12));
        pf.addProperty(new Apartment(12, "", 3, 20));
    }


    @Test
    public void testSortByPrice() {
        List<Property> testResult = pf.sortByPrice();

        // Initially test if the two lists the same size
        assertEquals(testResult.size(), pf.getOffers().size());

        // Now test if it is sorted
        for (int i= 1 ; i < testResult.size(); i++) {
            assertTrue(testResult.get(i - 1).getPrice() <=
                    testResult.get(i).getPrice());
        }
```

```
            // And test if every offer in the Portfolio is in the List
            for (Property p : pf.getOffers()) {
                assertTrue(testResult.contains(p));
            }
        }
    }
```

Question 3 (15 points)

a. *(7 points)* Add a method `removeProperty(Property p)` to the class `Portfolio` of Question 2, which removes a property from the portfolio, throwing an exception if the property is not found. Define a new exception for this purpose. You can use the `contains(Object o)` and `remove(Object o)` methods of the `Collection<E>` interface to implement this method. On page 492 of Eck's book (Section 10.1.4) you can find information on these methods.

b. *(8 points)* Add a method to the `Portfolio` class with the following signature

```
public Set<Property> removeProperties(Set<Property> ps)
```

that tries to remove all properties in `ps` from the portfolio, and returns a set with the properties that were not found in the portfolio and therefore have not been removed. Your method *must* use (i.e., catch) the exception you defined above, otherwise it is be considered incorrect.

Answer to question 3

a. (7 points) Grading criteria:

- 2 points: answer defines a `RemovePropertyException` that extends the `Exception` class
- 2 points: method signature includes **`throws`** `RemovePropertyException`
- 1 point: correctly throws (using **`throw new`** `...`) the exception if `offers` (or `getOffers()`) does not `contain` the property
- 1 point: removes the property if `offers` (or `getOffers()`) does `contain` the property
- 1 point: if the property is removed, decreases the value of `value` with the price of the removed property

```
public class RemovePropertyException extends Exception {
    // An empty body is enough for this question
}

public void removeProperty(Property p) throws RemovePropertyException {
    if (offers.contains(p)) {
        offers.remove(p);
        value = value - (int) p.getPrice();
    } else {
        throw new RemovePropertyException();
    }
}
```

b. (8 points) Grading criteria:

- 2 point: correct initialisation of the result `HashSet` variable, and correctly returning this variable at the end.
- 1 point: a **`for`** loop over all items of the `ps` parameter.

- 3 points: using a **try-catch** block around the `removeProperty` method that adds the property to the return set if the exception is thrown. No points if a too general exception is caught, e.g., if `Exception` is caught.
- 2 point: correctly adding non-removed properties to the `HashSet` result in the **catch** block.

```
public Set<Property> removeProperties(Set<Property> ps) {
    Set<Property> result = new HashSet<Property>();
    for (Property p : ps) {
        try {
            removeProperty(p);
        } catch (RemovePropertyException ex) {
            result.add(p);
        }
    }
    return result;
}
```

Question 4 (15 points)

The following Java interface makes it possible to select properties of a portfolio based on wishes of the customer:

```
public interface CustomerWish {
    /** Returns all properties of Portfolio p that match
     * the wishes of the customer */
    public Set<Property> matches(Portfolio p);
}
```

a. *(5 points)* Implement a class `AreaWish` that implements the `CustomerWish` interface by internally keeping a minimal and a maximal total area (as instance variables) for selecting the properties, so that a property only satisfies the `AreaWish` if its total area is between these bounds.

b. *(5 points)* Implement a method `bestPricedProperties` that, given a `Set` of properties, returns a `Set` of those properties with the lowest price of all the properties in the given `Set`.

c. *(5 points)* Implement a class `CombinedWish` that implements the `CustomerWish` interface by keeping a set of wishes in a `Set<CustomerWish> wishes` field, so that a property only satisfies the `CombinedWish` if it satisfies all the wishes in the `wishes` attribute.
Hint: use the `retainAll` method that is discussed on page 492 of Eck's book (Section 10.1.4).

Answer to question 4

a. (5 points): Grading criteria:

- 1 point: correct class definition including the **implements** keyword.
- 1 point: two **private** instance variables of an appropriate type (as in Question 1 (i.e.,**double** or **int**) and a **public** constructor that initializes these variables.
- 1 point: correct method signature of `matches`.
- 2 points: correct implementation of `matches`, using `getTotalArea` (1 point) and a **for** loop over all offers obtained via `getOffers()` (1 point).

```
public class AreaWish implements CustomerWish {
    private double min, max;

    public AreaWish (double min, double max) {
        this.min = min;
        this.max = max;
    }

    public Set<Property> matches(Portfolio pf) {
        Set<Property> result = new HashSet<Property>();
        for (Property p : pf.getOffers()) {
            if (p.getTotalArea() >= min && p.getTotalArea() <= max) {
                result.add(p);
            }
        }
        return result;
    }
}
```

b. (5 points): Grading criteria:

- 1 point: method definition: must return `Set<Property>` and have one parameter `Set<Property>`. Method can be **`static`** or not.
- 1 point: use a **`for`** loop over all properties in the given set.
- 1 points: keeps the lowest price in a local variable.
- 2 points: adds each property with the lowest price to the `Set` to be returned.

A correct implementation of this method with two loops, one to look for the lowest price and another one to populate the set, is also acceptable.

```
public static Set<Property> bestPricedProperties(Set<Property> ps) {
    Set<Property> result = new HashSet<Property>();
    double lowestPrice = -1;
    for (Property p : ps) {
        if (lowestPrice < 0 || p.getPrice() < lowestPrice) {
            lowestPrice = p.getPrice();
            result.clear();
        }
        if (p.getPrice() == lowestPrice) {
            result.add(p);
        }
    }
    return result;
}
```

c. (5 points): Grading criteria:

- 1 point: correct class definition using the **`implements`** keyword and implementing the method `matches` correctly.
- 1 point: a **`private`** field `wishes`.
- 3 points: the implementation of `matches` instantiates a result `Set` (1 point), populates it either via the constructor (as below) or with a **`for`** loop (1 point), and then uses a **`for`** loop over all wishes to keep only those offers that match each wish as below (1 point).

```
public class CombinedWish implements CustomerWish {
    private Set<CustomerWish> wishes = new HashSet<CustomerWish>();

    public void addWish (CustomerWish wish) {
        this.wishes.add(wish);
    }

    public Set<Property> matches(Portfolio pf) {
        Set<Property> result = new HashSet<Property>(pf.getOffers());
        for (CustomerWish w : wishes) {
            result.retainAll(w.matches(pf));
        }
        return result;
    }
}
```

Question 5 (10 points)

Suppose now that a multithreaded application that uses class `Portfolio` is implemented so that different concurrent threads can make calls to the methods of this class simultaneously. For example, these threads may try to add properties with `addProperty` or remove properties with `removeProperty`.

a. *(5 points)* Explain and give an example of what can go wrong if two threads try to add or remove a property at the same time in this program.

b. *(5 points)* Give a solution to this problem and explain how and why this solution works.

Answer to question 5

a. (5 points):

- The `value` instance variable may get a value that does not correspond to the sum of the values of all properties in the portfolio, because the addition or subtraction can go wrong. This happens if another thread starts after the value has been read and before it is modified in a thread (inside the addition or subtraction statement).
- According to the documentation, `HashSet` is not thread-safe, so that if multiple threads manipulate the structure of a `HashSet` by adding and removing elements at the same time, then the outcome is non-deterministic.

Grading criteria:

- 5 points: mentioned one of the problems and explained well what can happen.
- 3 points: mentioned vaguely one of the problems, with no explanation of how it could go wrong.

b. (5 points): Example of correct examples are:

- Use the **`synchronized`** keyword on the `addProperty` and `removeProperty` methods.
- Use a **`synchronized`** `()` block inside each method that modifies the `value` field and the `offers` set, synchronized on some shared object (can be the `Portfolio` object).
- Use explicit locking to protect both fields `offers` and `value` fields.
  In all these solutions, object locks will avoid that multiple threads execute certain code sections simultaneously (mutual exclusion).

Grading criteria:

- 5 points: mentioned one of the solutions and how it works.
- 3 points: mentioned one of the solutions, but not how it works.
- 1 point: only gave keywords with no explanation.

Question 6 (15 points)

Modern houses will have more and more fancy sensors (Internet-of-Things!) to make living more pleasant. Now consider a seller of WiFi-connected digital temperature sensors. The idea is that consumers buy those sensors, connect them to a WiFi network, and go to a web-based online service to view the (graphs of the) temperature measurements from anywhere in the world. Suppose the seller wants to make sure that the messages sent from the sensor to the web server are not tampered with in their travel over the evil internet.

a. *(1 point)* Which of the following methods is most suitable to protect the *integrity* of the messages?

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

b. *(2 points)* Explain your answer to Question 6.a.

c. *(2 points)* Integrity is one of the three often used security properties that, when violated, indicate there is a security incident. What are the other two properties?

Of course the online service for viewing the temperature values has an account system: users can create an account and protect it with a password.

d. *(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 sites to apply a hash function to their users' passwords instead of storing them as-is.

Sometimes users forget their passwords and for this reason this online service has implemented a password-reset functionality. After filling in an e-mail address, users are sent a new password by email. You notice that these reset passwords are 9-11 characters long and have the following pattern:

- First, all passwords start with the text "tt", "bbb", or "gggg",
- after which come 3 characters from the set {"g", "k", "l", "o"} and
- this is followed by four digits.

e. *(2 points)* If an attacker would try to brute-force access to an account with such a reset password, how many attempts would it at most take? Show your calculation.

After a while, the owner of the service is starting to hear rumours that there is an easy way to get access to any account on the system. You are asked to investigate and solve this problem. You look around in the messy code base and find the code shown below. Apparently someone thought it would be a good idea to add some flexibility in the code handling the login process by combining the password with a string that describes which (cryptographic) hash-function should be used. So a password "bike" in combination with the MD5 hash function will be passed along as "MD5:bike".

```
private Map<String, String> passwordDB;

/**
 * Generates the hex-encoded hash of the password using a very flexible
 * scheme. The hash-function to use is embedded in the password: it is
 * separated by a colon. Example passwords: "MD5:abcd" or "SHA1:s3cr3t".
 * @throws NoSuchAlgorithmException
 */
public String getPWHash(String password) throws NoSuchAlgorithmException {
    String[] r = password.split(":");
    String prefix = r[0];
    String realPassword = r[1];
    MessageDigest md = MessageDigest.getInstance(prefix);
    md.update(realPassword.getBytes());
    byte[] digest = md.digest();
    return Hex.encodeHexString(digest);
}

public boolean login(String username, String password) {
    boolean result = true;
    if (passwordDB.containsKey(username)) {
        try {
            String passwordHash = getPWHash(password);
            if (!passwordDB.get(username).equals(passwordHash)) {
                result = false;
            }
        } catch (Exception e) {
            // Whatever, shouldn't happen, right?
        }
    } else {
        result = false;
    }
    return result;
}
```

De Javadoc documentation of the method `String.split` is 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

Suppose an attacker (somehow) has full control over the *content* of the variables `username` and `password` that are passed along to the `login` method. There are (at least) two ways for the attacker to get the `login` method to return true (and thus gaining access) without actually knowing a password.

f. *(5 points)* Describe at least one way an attacker can get access to any account. Also show how one can (easily) solve this issue in the code.

Answer to question 6

a. *(1 points)* HMAC.

b. *(2 points)* HMAC uses cryptographic hashing to help detect changes in data contents, making it extremely suitable to protect data integrity.

c. *(2 points)* Confidentiality, Availability.

d. *(2 points)* People often re-use password across domains; when site is compromised, the accounts of the users at other sites can also be compromised.

e. *(3 points)* $3 \times 4^3 \times 10^4 = 1920000$

f. *(5 points)* In the login method, the boolean `result` is not set to **`false`** in the catch statement. So whenever an exception is thrown, the login method will return true! There are (at least) two ways an exception can be thrown:

- If you provide an invalid hashing algorithm (e.g, "monkey:password"), an `NoSuchAlgorithmException` will be thrown.
- If you do not put an ":" in the password, an `IndexOutOfBoundsException` will be thrown.

A simple way to prevent this is to initialize the `result` variable to **`false`**.

- 2 points for mentioning that the login method will return **`true`** if an exception is thrown.
- 2 points for mentioning at least one way in which an exception gets thrown.
- 1 point for mentioning the solution (initialising the `result` variable to **`false`**).