TEST

Software Systems: Programming

course code: 201700117
date: 31 January 2019
time: 13:45 – 16:45

SOLUTIONS

General

- When doing this test, you may use the following (unmarked) materials :
  - the module manual;
  - the slides of the lectures;
  - two Java books of your preference, and
  - a dictionary.

  You may *not* 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.
- 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.
- Your grade for this test is used to calculate the final grade of the module. The formula used to calculate the final grade of the module can be found in the module manual.

Question 1 (25 points)

In this test, we consider the design of a computer game. To start we define a class `Screen` to keep track of where figures (players and animations) are positioned on a certain screen area.

```
public class Screen {
    public static final int XMAX = 80;
    public static final int YMAX = 80;

    private static List<Player> players;
    private static List<Animation> animations;

    public static List<Player> getPlayers() { return players; }

    public static List<Animation> getFigures() { return animations; }

    // Methods added to allow the game to be initialised
    // with players and figures
    public static void setPlayers(List<Player> l) { players = l; }

    public static void setFigures(List<Animation> l) { animations = l; }

}
```

In order to share the screen more easily with all players, the `Screen` methods and fields are defined as **`static`**.

a. (*5 points*) What is the consequence of defining methods as **`static`** for the code that will call these methods? Illustrate your answer with a code example in which method `getPlayers()` is called.

In addition, we programmed a class `Position` that defines where a figure is located when the game is played, as shown in the sequel.

```
public class Position {
    private int x, y;
    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }

    public int getY() { return y; }

    public void setX(int x) { this.x = x; }

    public void setY(int y) { this.y = y; }

    // given a valid position (bx, by), it returns the
    // number of steps necessary to reach this position
    public int getDistance(int bx, int by) {
        // To be implemented
    }

    // returns the number of steps to reach the given position
    public int getDistance(Position p) {
        // To be implemented
    }
    // returns an arbitrary neighbouring position or
    // the same position
    public Position getNeighbourPosition() {
        // To be implemented
    }
}
```

b. (*4 points*) Program the body of the methods `getDistance(int bx, int by)` and `getDistance(Position p)`. In these methods, we assume that the distance is defined as the number of steps necessary to go from one position to another. For example, 3 steps are necessary to go from position (1, 2) to position (2, 4), namely 1 horizontal step and 2 vertical steps.

Hint: In order to program these methods, you can use method `static int Math.abs(int x)`, which returns the absolute value of `x`, i.e., $|x|$.

c. (*6 points*) Program the body of method `getNeighbourPosition()`, which returns an arbitrary (*random*) neighbouring position or the same position, with uniform probability distribution. Neighbouring positions have a distance of 1. For example, position (1, 2) is neighbour of position (2, 2).

Hints: In order to program this method, you can use method `static double Math.random()`, which returns a random positive value bigger than 0.0 and smaller or equal to 1.0 with uniform probability distribution. Don't forget to consider the boundaries of the screen!

d. (*10 points*) Give the *public invariant* that holds for the x and y coordinates, and the complete JML specification (i.e., both pre- and post-conditions) of methods `setX(int x)` and `getDistance(int bx, int by)`.

Answer to question 1

a. (*5 points*: 3 for the explanation; 2 for the call.)

`static` means that the method belongs to a class, not to an object. This also means that you don't need to create an object to call this method, i.e., you can call it on the class. In this specific case the call looks like `Screen.getPlayers()`.

b. (*4 points*: 2 for each method; it is acceptable if the code of the first method is duplicated in the second if it is correct; 1 if the value generated is almost correct.)

```
public int getDistance(int bx, int by) {
    return Math.abs(x - bx) + Math.abs(y - by);
}

public int getDistance(Position p) {
    return getDistance(p.getX(), p.getY());
}
```

c. (*6 points*: 3 for returning five values, 3 for values calculated (almost) correctly from a random value.)

```
public Position getNeighbourPosition() {
    int r = (int) (Math.random() * 5);
    int new_x = x;
    int new_y = y;
    if (r == 0 && x > 0) {
        // go to the left
        new_x = x - 1;
    }
    else if (r == 1 && x < Screen.XMAX) {
        // go to the right
        new_x = x + 1;
    }
    else if (r == 2 && y > 0) {
        // go down
        new_y = y - 1;
    }
    else if (r == 3 && y < Screen.YMAX) {
        // go up
        new_y = y + 1;
    }
    return new Position(new_x, new_y);
}
}
```

d. (*10 points*: 4 for the invariant, 3 for both preconditions, 3 for both postconditions.)

- Invariant:

```
//@ public invariant 0 <= getX() && getX() <= Screen.XMAX;
//@ public invariant 0 <= getY() && getY() <= Screen.YMAX;
```

- Method specifications (preconditions and postconditions):

```
//@ requires 0 <= x && x <= Screen.XMAX;
//@ ensures getX() == x;
public void setX(int x) { this.x = x; }
```

```
//@ pure
//@ requires 0 <= bx && bx <= Screen.YMAX;
//@ requires 0 <= by && by <= Screen.YMAX;
//@ ensures \result == Math.abs(x - bx) + Math.abs(y - by);
public int getDistance(int bx, int by) {
    return Math.abs(x - bx) + Math.abs(y - by);
}
```

Question 2 (25 points)

In our game, figures can move on the screen. The following interface defines a `Figure`:

```
public interface Figure {
    //@ ensures 0 <= \result.getX() && \result.getX() <= Screen.XSIZE;
    //@ ensures 0 <= \result.getY() && \result.getY() <= Screen.YSIZE;
    public Position getPosition();

    //@ ensures \result != null;
    public Colour getColour();

    //@ ensures getColor().equal(c);
    public void setColour(Colour c);
}
```

In addition to a position, each figure has a colour, which is defined with RGB codes, as programmed in the class below.

```
public class Colour {
    private int r, g, b;

    public Colour(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

We distinguish two types of figures in our game: *players* and *animations*.

- Players represent the humans who play the game. A player has a method called `goTo(Position p)` that gets a *valid neighbouring position* `p` (i.e., a neighbouring position inside the screen area) and moves the player to this position. Each player has also a *credit points* attribute (an **`int`** value) and methods to get these credits and to decrease the credits by some points. We assume that players are created with some initial credit points.

- Animations can move and attack players. An animation has a method called `takeStep()` to take a step to a neighbouring position, and a method called `attack(Player p)` to attack a player. An animation can only attack a player if they are at neighbouring positions or on the same position.

a. (*3 points*) Define interfaces `Player` and `Animation` to represent players and animations, respectively.

b. (*4 points*) Give the JML specifications of the methods of the `Animation` interface.

c. (*8 points*) Program a class `Monster` that properly implements the `Animation` interface. This class has a constructor that gets the colour of the monster as an argument. A monster chooses randomly to which neighbouring position it moves when it takes a step. Each 4th attack of a monster is successful, while the other attacks are unsuccessful. When an attack is successful the credits points of the attacked player are decreased by 10 points.

d. (*10 points*) Program a class `Ogre` that extends `Monster` and represents an *ogre*. An ogre is a green monster (the RGB code for green is (1, 128, 0).) The ogre always moves in the direction of the first player in the list. Override method `takeStep()` in the `Ogre` class to represent this behaviour.

Hint: Define the following helper methods:

- `Player selectPlayer()` to select the player that the ogre will follow[1]; and

[1] You can assume that the set of players is not empty during the game.

- `goTo(Player p)` to determine the next step to a neighbouring position to be taken to reach player `p`.

Answer to question 2

a. (*3 points*: 2 for `Player` and 1 for `Animation`)

```
public interface Player extends Figure {


    //@ requires getLocation().getDistance(x, y) == 1;
    //@ ensures getLocation().getX() == x;
    //@ ensures getLocation().getY() == y;
    public void goTo(Position p);

    public int getCredits();

    public void decreaseCredits(int value);
}

public interface Animation extends Figure {

    //@ ensures getPosition().getDistance(\old(getPosition())) <= 1;
    public void takeStep();

    //@ requires s != null;
    //@ requires getPosition().getDistance(s.getPosition()) == 1 ||
    //              getPosition().equals(s.getPosition());
    public boolean attack(Player s);
}
```

b. (*4 points*: 2 for each method specification, where the first `requires` in method `attack` can be neglected because it is more or less implied by the second condition.)

See above.

c. (*8 points*: 2 for proper attributes, 1 for constructor, 1 for all getters and setters, 1 for `takeStep` and 3 for `attack`, where 2 for the proper calculation of the successful attack and 1 for decreasing the credit of the player.)

```
public class Monster implements Animation {

    private Position pos;
    private Colour colour;
    private int attack;
    public static final int SUCCESSFACTOR = 4;
    public static final int POWER = 10;

    public Monster(Colour c) {
        pos = new Position(0, 0);
        colour = c;
        attack = 0;
    }

    @Override
    public Position getPosition() {
        return pos;
    }
```

```
    @Override
    public Colour getColour() {
        return colour;
    }

    @Override
    public void setColour(Colour c) {
        colour = c;
    }

    @Override
    public void takeStep() {
        pos = pos.getNeighbourPosition();
    }

    @Override
    public boolean attack(Player p) {
        boolean result = (attack == SUCCESSFACTOR - 1);
        attack = (attack + 1) % SUCCESSFACTOR;
        if (result) {
            p.decreaseCredits(POWER);
        }
        return result;
    }
}
```

d. (*10 points*: 2 for proper initialisation of colour, 8 for `takeStep`. If hints were followed then 2 for `selectPlayer`, 4 for `goTo` and 2 for calling them properly in `takeStep`. If hints are not followed check the logic to verify whether it makes sense.)

```
public class Ogre extends Monster

    public static final Colour green = new Colour(0, 128, 0);

    public Ogre() {
        super(green);
    }

    @Override
    public void takeStep() {
        Player s = selectPlayer();
        goTo(s);
    }

    public Player selectPlayer() {
        return Screen.getPlayers().get(0);
    }

    public void goTo(Player s) {
        Position loc = getPosition();
        Position sloc = s.getPosition();
        if (sloc.getX() > loc.getX()) {
            loc.setX(loc.getX() + 1);
        }
        else if (sloc.getX() < loc.getX()) {
```

```
                loc.setX(loc.getX() - 1);
            }
            else if (sloc.getY() > loc.getY()) {
                loc.setY(loc.getY() + 1);
            }
            else {
                loc.setY(loc.getY() - 1);
            }
        }
    }
```

Question 3 (15 points)

Players are classified according to the number of credits they have at some time, and if they belong to a particular class they get some privileges (not further specified here). In this exercise we define three levels, which are represented by the following enumeration:

```
public enum Level {
    gold, silver, bronze;
}
```

Class `PrivilegeMap` keeps track of the groups of players for each level.

```
public class PrivilegeMap {

    public static final int BRONZE_MAX = 50;
    public static final int SILVER_MAX = 100;

    private Map<Level, Set<Player>> map;

    public void classifyPlayers() {
        // To be implemented
    }

    public int levelDifference(PrivilegeMap other, Level l) {
        // To be implemented
    }

}
```

a. (*5 points*) Implement methods `classifyPlayers()`, which fills in the attribute `map` by assigning each player from `Screen` (see Question 1) to the set that corresponds to its level.

b. (*10 points*) Implement a method called `levelDifference` that gets another `PrivilegeMap` and a level as argument, and returns the difference of the total credits of the players for this level. For example, if we call this method with another `PrivilegeMap` and level `silver`, it will calculate the total number of credits of the silver players in each `PrivilegeMap` and return the difference.

Answer to question 3

a. (*5 points*: 3 proper initialisation of the `map`, 2 proper classification of the players.)

```
public void classifyPlayers() {
    map = new HashMap<Level, Set<Player>>();
    map.put(Level.bronze, new HashSet<Player>());
    map.put(Level.silver, new HashSet<Player>());
    map.put(Level.gold, new HashSet<Player>());
    List<Player> players = Screen.getPlayers();
    for (Player s : players) {
        int credits = s.getCredits();
        if (credits <= BRONZE_MAX) {
            map.get(Level.bronze).add(s);
        }
        else if (credits <= SILVER_MAX) {
            map.get(Level.silver).add(s);
        }
        else {
            map.get(Level.gold).add(s);
        }
    }
}
```

b. (*10 points*: 6 if proper values are found, 4 if an attempt is made to send the difference.)

```
public int levelDifference(PrivilegeMap other, Level l) {
    int this_sum = 0, other_sum = 0;

    for (Player s : map.get(l)) {
        this_sum = this_sum + s.getCredits();
    }
    for (Player s : other.map.get(l)) {
        other_sum = other_sum + s.getCredits();
    }
    return Math.abs(other_sum - this_sum);
}
```

Question 4 (10 points)

Figures can only move to positions that are inside the screen area. This creates restrictions to the positions to which players can try to move.

In order to indicate that a player tried to move to a position outside the screen we define the following exception:

```
public class IllegalMoveException extends Exception {

    public IllegalMoveException(int x, int y) {
        super("(" + x + "," + y + ") is not on the screen");
    }
}
```

a. *(3 points)* Program a method called `tryToMove` that gets a neighbouring position and tries to move the player to this position. This method checks if the position is inside the screen, and if this is not the case it throws a `IllegalMoveException` exception, otherwise it moves the player to the new position.

b. (*7 points*) Define method `goToRightDown()` that uses method `tryToMove()` to move in single steps to the right of the screen and then to the bottom of the screen, without testing the actual position (i.e., without testing if the position is bigger or equal to `Screen.XMAX` and `Screen.YMAX`, respectively). Explain why it is in general not a good idea to implement method `goToRightDown()` in this way.

Answer to question 4

a. *(3 points):* 2 *for properly throwing the exception,* 1 *for the condition.*

```
public void tryToMove(int x, int y) throws IllegalMoveException {
    if (x < 0  || x > Screen.XMAX || y < 0 || y > Screen.YMAX) {
        throw new IllegalMoveException(x, y);
    }
        else {
            goTo(new Position(x, y));
    }
}
```

b. (*7 points*: 2 for each **try** clause, 1 for stopping at the right position, 2 for the explanation.)

```
public void goToRightDown() {
    try {
        while (true) {
            tryToMove(pos.getX() + 1, pos.getY());
        }
    }
    catch (IllegalMoveException e) {
    }
    try {
        while (true) {
            tryToMove(pos.getX(), pos.getY() + 1);
        }
    }
    catch (IllegalMoveException e) {
    }
}
```

This is not a good idea because exceptions are meant to represent error conditions (violations of invariants or preconditions, for example), and not to be used in the implementation of normal behaviour like this. This is bad and uncommon programming practice.

Question 5 (10 points)

Ogres and players can run in their own thread. The ogre thread could have a `run()` method like this:

```
public void run() {
    while (true) {
        takeStep();
        for (Player s : Screen.getPlayers()) {
            if (getPosition().getDistance(s.getPosition()) == 1 ||
                    getPosition().equals(s.getPosition())) {
                attack(s);
            }
        }
    }
}
```

The player thread could have a `run()` method like this, where for the sake of simplification we don't do much in the **`catch`** clause:

```
public void run() {
    while (true) {
        int new_x = 0, new_y = 0;
        try {
            new_x = System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            new_y = System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (new_x >= 0 && new_x < Screen.XMAX &&
            new_y >= 0 && new_y < Screen.YMAX &
            getPosition().getDistance(new_x, new_y) == 1) {
                goTo(new Position (new_x, new_y));
        }
    }
}
```

a. (*6 points*) Explain what can go wrong with this implementation if methods `attack` and `goTo` are performed concurrently. Explain how this problem can be solved.

b. (*4 points*) Suppose that a `LazyMonster` is added to the game that waits until a player is at a certain distance and then tries to quickly move to this player. Explain how this monster could be implemented so that it would not check the position of all players at all times, but would only check when the position of a player changes. Explain how the program needs to be modified to achieve this.

Answer to question 5

a. (*6 points:* 3 for the problem, max 2 in case the below specific case is not mentioned; 3 for the solution, where 1 for mentioning synchronization, 2 for a more detailed description including synchronizing on a common object or a lock object.)

- *Problem*: The monster can start the attack when a player is in neighbouring position, and the player can move before the attack is finished, violating the rules of the game.

- *Possible solution*: Make sure a move of the player does not happen during the attack by synchronising these pieces of code using a **`synchronized`** block on a common object or a `lock` object.

b. (*4 points:* 2 for mentioning `wait-notify`, 2 if solution makes sense.)

Use a `wait`/`notify` combination. Define a `lock` for each player, and whenever a player moves it calls `notify`. The Lazy monster then waits for all players (with `wait` on the same `lock`) and checks only then if the player is in a neighbouring position.

Question 6 (15 points)

Suppose this game was developed further so that it can be played online as well. To make the game even more interesting, all kinds of features have been added. For example, players can become owner of items and can hold special in-game coins. These coins can then be used to buy more (virtual) items to use in the game. And because this virtual money can also be traded for "real" money, the users of the game will want a system that is properly secured.

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 this online game, 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.

As a player, you of course do not want the messages and commands you send to the game server are modified along the way.

d. *(2 points)* What security-mechanism (if properly implemented) can allow the server to check whether the messages it receives haven't been modified along the way by a malicious party?

Suppose now that a variant of this game has been running for a while. In an attempt to earn more money, the company that developed this online version introduces special premium levels. As a player you can only enter these levels if you have enough coins (which you have to buy) or if you have a lot of experience. 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 rumors that it is possible to enter premium levels without needing to buy coins or gaining a lot of experience points. 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 and on the next page.

e. *(5 points)* Explain, using the code shown, how a malicious player can visit the premium levels without buying coins or gaining experience. You can assume that a user can set his or her name and tagline at will.

```
public class OnlinePlayer {

    private String name;
    private String description;
    private int experiencePoints;
    private int totalCoins;

    // ... [Some methods that are not relevant for the question]

    /**
     * Give information about the object as a String.
```

```
     * Returns the name, experience points and total coins,
     * separated by "::"
     * @return
     */
    public String getInfo() {
        return name + "::" +
                description + "::" +
                Integer.toString(experiencePoints) + "::" +
                Integer.toString(totalCoins);
    }

   // ... [Some methods that are not relevant for the question]

}
```

```
public class PremiumLevel {
   // ... [Some methods that are not relevant for the question]

    private static final int MINIMAL_EXPERIENCE = 40;
    private static final int MINIMAL_TOTAL_COINS = 15;
    /**
     * Check whether the player is allowed to enter this level
     * @param player
     * @return true if player is allowed to enter this level
     */
    public boolean checkPremium(OnlinePlayer player) {
        String playerInfo = player.getInfo();
        String[] infoItems = playerInfo.split("::");
        if (Integer.parseInt(infoItems[2]) > MINIMAL_EXPERIENCE ||
                Integer.parseInt(infoItems[3]) > MINIMAL_TOTAL_COINS) {
            return true;
        }
        return false;
    }
}
```

De javadoc 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)* The most straightforward approach is to use a message authentication code (MAC) [full points]. Mentioning HMAC (a MAC that uses a hash function) also gets full points. Only saying "cryptographic hash function" or one of the existing hash functions (SHA-256, MD5) gets only 1 point: since computing a hash does not require a key, an attacker can simply recompute the hash for the modified message. Although suboptimal in this case, a digital signature (using asymmetric cryptography, for example RSA) also gets full points. Saying "encryption" gets no points.

e. *(5 points)* A malicious user can simply add a string in the tagline (or even the name) that contains two times "::" and some large numbers. These numbers are then (wrongly) interpreted as the experience and number of coins of the player, after which the player is accepted to the premium level. An example tagline is: "Experience is everything!::60::12". Full points if a similar explanations 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.