TEST

Software Systems

course code: 201500111
date: January 23rd, 2017
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.
- The number of points scored on this test will be taken into account while calculating the final grade. The formula used to do so can be found in the reader.
- 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 (30 points)

We are programming a ticket agency. The following class is given:

```
/** Location of a show. */
public class Location {
    /** Address of the location. */
    public final String name;
    /** Number of seats of the location. */
    public final int capacity;

    /** Constructs a new location. */
    public Location(String name, int capacity) {
        this.name = name;
        this.capacity = capacity;
    }
}
```

a. *(4 points)* You have been told in the module that fields should always be declared as **`private`**.

- Why is this a good practice in general?
- The fields of `Location` are **`public`**. Discuss reasons why this may be appropriate in this case.

b. *(4 points)* Define an interface `Show` with the following methods (you may answer this together with the next subquestion):

- A method `getPrice` that returns the price of a ticket (a **`double`**);
- A method `getDiscount` that returns a price discount percentage for frequent customers (an **`int`** between 0 and 100);
- A method `setDiscount` that sets the discount percentage to a new value;
- A method `getDiscountPrice` that returns the discounted price;
- A method `getLocation` that returns the `Location` where the show takes place;
- A method `getCapacity` that returns the location capacity, i.e., it's number of seats (an **`int`**).

c. *(8 points)* Provide JML specifications for the methods above. Be as complete as you can. (You may answer this together with the previous subquestion.)

d. *(4 points)* Define an enumerated type `Genre` with values for comedy, musical, western and romance.

e. *(4 points)* Define an interface `TypedShow` that extends `Show` by additionally maintaining a set of genres of a show, and providing the following *JML-specified* methods:

- `addGenre` which adds a `Genre` (provided as parameter) to the set;
- `getGenres` which returns the set of all genres of the `TypedShow`.

Do not forget to add the requested JML specification!

f. *(6 points)* Define a class `Movie` that implements `TypedShow`. The constructor should initialise (only) the location and price of a movie.

Answer to question 1

a. *(4 points.)*

- Making fields **`private`** is good for encapsulation. The fields cannot be changed from outside the object except by calling methods, meaning that consistency of values can be guaranteed better. Also, if the implementation of a field changes, this is invisible outside the class. *(At least one of these answers thould be given; -2 otherwise.)*

- The fields in question are **`final`**, meaning they cannot be changed even though they are **`public`**. The second point above still argues against this — changes to the field implementation are now visible outside the class. *(−2 if **`final`** is not mentioned. No deduction if the point about changes in implementation being visible is not made.)*

b. *(4 points; −1 per independent syntax error, including argument and return types.)*

```
public interface Show {
    /*@ pure
      @ ensures \result >= 0 */
    public double getPrice();

    /*@ pure
      @ ensures \result == (100-getDiscount()) * getPrice() / 100 */
    public double getDiscountPrice();

    /*@ requires discount >= 0 && discount <= 100
      @ ensures getDiscount() == discount */
    public void setDiscount(int discount);

    /*@ pure
      @ ensures \result >= 0 && \result <= 100 */
    public int getDiscount();

    /*@ pure
      @ ensures \result != null */
    public Location getLocation();

    /*@ pure
      @ ensures \result == getLocation().capacity */
    public int getCapacity();
}
```

c. *(8 points:*

- *−2 if `pure` is not used anywhere, −1 if it is used but forgotten in one or more places;*
- *−1 if the `ensures` clauses of `getPrice` or `getLocation` are omitted*
- *−1 if the `ensures` of `getDiscountPrice` is not precise enough;*
- *No deduction if the bounds 0 and 100 for the discount are non-inclusive*
- *−2 for other (independent) errors).*

For the answer see above.

d. *(4 points; −1 if not all-caps (violation of code convention), −3 if **`enum`** is not used*

```
public enum Genre {
    COMEDY,
    MUSICAL,
    WESTERN,
    ROMANCE,;
}
```

e. *(4 points; −2 if **`extends`** is missing, −1 if **`implements`** is used instead, −1 for Java errors up to −3, −1 for JML errors up to −2)*

```
public interface TypedShow extends Show {
    /*@ requies genre != null
      @ ensures getGenres().contains(genre) */
    public void addGenre(Genre genre);

    /*@ pure
      @ ensures \result != null */
    public Set<Genre> getGenres();
}
```

f. *(6 points; −1 per independent Java error.* ***final*** *does not have to be used.)*

```
public class Movie implements TypedShow {
    private final Set<Genre> genres;
    private final double price;
    private final Location location;
    private int discount;

    public Movie(Location location, double price) {
        this.genres = new HashSet<>();
        this.location = location;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public double getDiscountPrice() {
        return (100 - getDiscount()) * getPrice() / 100;
    }

    public void setDiscount(int discount) {
        this.discount = discount;
    }

    public int getDiscount() {
        return discount;
    }

    public Location getLocation() {
        return location;
    }

    public int getCapacity() {
        return getLocation().capacity;
    }

    public void addGenre(Genre genre) {
        genres.add(genre);
    }

    public Set<Genre> getGenres() {
        return genres;
    }
}
```

Question 2 (15 points)

Consider the following (incomplete) classes `Profile`, representing a customer profile at a ticket agency

```
public class Profile {
    /** Number of booked shows required to be frequent customer. */
    public static final int FREQUENT_CUSTOMER = 10;
    /** Name of this profile. */
    public final String name;
    /** Chosen shows. */
    private final Set<Show> choices = new HashSet<>();
    /** Preferential genres, i.e., which occur in at least one chosen show. */
    private final Set<Genre> prefs = new HashSet<>();

    /** Constructs a profile for a given name. */
    public Profile(String name) {
        this.name = name;
    }

    /** Adds a show to the choices and preferences of this profile.
     * @return true if the show is new for this profile, false otherwise */
    public boolean addChoice(Show show) {
        // To be programmed
    }

    /** Returns the set of chosen shows. */
    public Set<Show> getChoices() {
        return choices;
    }

    /** Returns the collected preferences in the chosen shows. */
    public Set<Genre> getPrefs() {
        return prefs;
    }

    /** Tests if this profile represents a frequent customer. */
    public boolean isFrequent() {
        // To be programmed
    }
}
```

a. (*4 points*) Program the missing methods `addChoice` and `isFrequent` of `Profile`, taking the following into account:

- The preferences are those genres of which the profile has ever booked a `TypedShow`;
- A profile represents a frequent customer if the number of chosen shows is as least as large as `FREQUENT_CUSTOMER`.

Now consider the (incomplete) class `Agency`, representing a ticket agency that has a given set of available shows:

```
public class Agency {
    /** Map from shows to number of tickets sold for that show. */
    private final Map<Show, Integer> soldMap = new HashMap<>();

    /** Adds a show to the ones available through this agency. */
    public void addShow(Show show) {
        // To be programmed
```

```
    }

    /** Returns all shows available through this agency. */
    public Set<Show> getShows() {
        // To be programmed
    }

    /** Books a number of tickets for a given show. */
    public void bookShow(Show show, int tickets) {
        soldMap.put(show, soldMap.get(show) + tickets);
    }

    /** Returns the number of tickets left for a given show. */
    /*@ ensures \result >= 0 */
    public int getSpace(Show show) {
        // To be programmed
    }
}
```

b. (*5 points*) What can go wrong in the method `bookShow`? Add a JML specification that restricts the usage of the method to those cases where no error can ensue.

c. (*6 points*) Program the missing methods `addShow`, `getShows` and `getSpace` of `Agency`, taking into account that the number tickets left for a show equals the capacity of that show (see the method `capacity` of `Show`) minus the tickets sold.

Answer to question 2

a. (*4 points; 2 per method. Note that the original version of the Javadoc for this method did not have the* `@return` *clause. This has been announced during the exam. If there are any errors that appear to have been caused by this mistake, be lenient.*) The method `addChoice`:

```
public boolean addChoice(Show show) {
    boolean result = choices.add(show);
    if (show instanceof TypedShow) {
        prefs.addAll(((TypedShow) show).getGenres());
    }
    return result;
}
```

The method `isFrequent`:

```
public boolean isFrequent() {
    return choices.size() >= FREQUENT_CUSTOMER;
}
```

b. (*5 points; up to* -2 *if explanation is missing or unclear, up to* -3 *for missing up wrong JML spec; of course max is* -5) Two things may go wrong. First, if `show` has never been registered for the agency, then `soldMap.get(show)` will return **`null`**. This will give riste to a `NullPointerException` upon the attempt to unbox it and turn in into an **`int`**. Second, the method does not test if the show is booked full.

A JML specification stating that `show` must be known within the agency and not booked full is, for instance: *(The requirement on* `ticket` *being positive need not be given; if there, give* $+1$ *as compensation to any deductions.)*

```
/*@ requires show != null && getShows().contains(show)
  @ requires getSpace(show) >= tickets
  @ requires tickets > 0 */
public void bookShow(Show show, int tickets) {
    soldMap.put(show, soldMap.get(show) + tickets);
}
```

c. (*6 points; 2 per method*) The method `addShow`:

```
public void addShow(Show show) {
    soldMap.put(show, 0);
}
```

The method `getShows`:

```
public Set<Show> getShows() {
    return soldMap.keySet();
}
```

The method `getSpace`:

```
public int getSpace(Show show) {
    if (soldMap.containsKey(show)) {
        return show.getCapacity() - soldMap.get(show);
    } else {
        return 0;
    }
}
```

Question 3 (15 points)

Consider the following abstract class:

```
public abstract class Suggestion {
    /** The agency to which this suggestion belongs. */
    protected final Agency agency;
    /** Constructs a suggestion for a given agency. */
    protected Suggestion(Agency agency) {
        this.agency = agency;
    }
    /** Returns a suggestion for a given customer. */
    abstract public Set<Show> getShows(Profile cust);
}
```

You are asked to program two (concrete) classes extending `Suggestion`.

a. *(2 points)* What does the keyword **`protected`** in front of the declarations of the field `agency` and the constructor `Suggestion` mean?

b. *(5 points)* Program a class `GenreOverlap` extending `Suggestion`, in which `getShows` returns all shows available through the agency that satisfy all of the following conditions:

- The customer hasn't yet booked the show;
- The show has at least one of the genres of the customer's preference;
- The show is not fully booked.

For instance, if the agency offers the movie *Serendipity* with genres *comedy* and *romance*, and *Once upon a Time in the West* with genre *western*, then a customer with preference *comedy* and *western* would be suggested both of these movies under the `GenreOverlap` policy.

Shows that are not `TypedShow`s will never be suggested, as no genre is known for them.

c. *(5 points)* Program a class `GenreInclusion` extending `Suggestion`, in which `getShows` returns all shows available through the agency that satisfy all of the following conditions:

- The customer hasn't yet booked the show;
- The show's genres *completely* fall within customer's preference;
- The show is not fully booked.

For instance, in the same example as above, only *Once upon a Time in the West* will be suggested under the `GenreInclusion` policy.

d. *(3 points)* `Suggestion` is an abstract class. How would the solutions to the previous subquestions have to change if it had been an interface? *(You only have to describe the change in words, not give the full code.)*

Answer to question 3

a. *(2 points)* This means that the field and constructor can be accessed, respectively invoked, from any subclass (and also from all classes in the same package; no deduction if this last point is not mentioned), but not from arbitrary other classes.

b. *(5 points. Do not deduct points twice in case the same error is made in the next subquestion; essentially, you may regard these two subquestions as a single question for 10 points. Note that there are other correct solutions, including the use of streams.*

- -2 *for a missing* **`super`** *call;*
- -2 *if the previous choices are not subtracted;*

- -2 *if the test for space is missing;*
- -2 *if the* **`instanceof`** *test is missing;*
- -1 *if* **`instanceof`** *is there bu the case to* `TypedShow` *is missing)*

Note that the original question mentioned `getShow` *rather than* `getShows`*. Be lenient if this seems, for whatever reason, to have caused confusion.*

```
public class GenreOverlap extends Suggestion {
    public GenreOverlap(Agency agency) {
        super(agency);
    }

    public Set<Show> getShows(Profile cust) {
        Set<Show> result = new HashSet<>();
        for (Show show : agency.getShows()) {
            if (agency.getSpace(show)>0 && show instanceof TypedShow) {
                for (Genre genre : ((TypedShow) show).getGenres()) {
                    if (cust.getPrefs().contains(genre)) {
                        result.add(show);
                        break;
                    }
                }
            }
        }
        result.removeAll(cust.getChoices());
        return result;
    }
}
```

c. *(5 points; see above for point deductions.)*

```
public class GenreInclusion extends Suggestion {
    public GenreInclusion(Agency agency) {
        super(agency);
    }

    public Set<Show> getShows(Profile cust) {
        Set<Show> result = new HashSet<>();
        for (Show show : agency.getShows()) {
            if (agency.getSpace(show) > 0 && show instanceof TypedShow
                    && cust
                        .getPrefs()
                        .containsAll(((TypedShow) show).getGenres())) {
                result.add(show);
            }
        }
        result.removeAll(cust.getChoices());
        return result;
    }
}
```

d. *(3 points. Essentially three changes;* -1 *for any that is omitted.)*

- Instead of **`extends`**, the implementing classes should specify **`implements`**
- The field declaration in `Suggestion` should be copied to all implementing classes;
- The **`super`** call in the implementing classes should be omitted, instead an assignment to `agency` should be added.

Question 4 (10 points)

Consider the following class `Ticket`, representing a ticket to a show, with a given named owner and price:

```
public class Ticket {
    private final String owner;
    private final Show show;
    private final double price;

    // Constructor to be programmed

    public String getOwner() {
        return owner;
    }

    public Show getShow() {
        return show;
    }

    public double getPrice() {
        return price;
    }
}
```

a. *(3 points)* Define a class `TicketException` that extends `Exception`.

b. *(7 points)* Program the constructor of `Ticket`, taking the following into account:

- It takes an `Agency`, a `Show` and a `Profile` as parameters;
- It throws a `TicketException`, with an appropriate message, if the show is booked full;
- It also throws a `TicketException`, with an appropriate message, if the profile already has booked the show;
- It determines the ticket's `owner` and `price` based on the profile.

Answer to question 4

a. *(3 points)*

```
public class TicketException extends Exception {
    public TicketException(String arg0) {
        super(arg0);
    }
}
```

b. *(7 points; −2 if the* **`throws`** *clause is omitted; −2 for any incorrectly thrown* `TicketException`*, −1 if there is no appropriate message; −2 if the discount is not applied, or the show is not booked, or the show is not added to the profile.)*

```
public Ticket(Agency agency, Show show, Profile cust)
        throws TicketException {
    // check if there are tickets available
    if (agency.getSpace(show) == 0) {
        throw new TicketException("Show␣is␣booked␣full");
    }
    // add show to customer choices
    if (!cust.addChoice(show)) {
        throw new TicketException("Customer␣already␣booked␣this␣show");
    }
    // add ticket to nr. of tickets sold
    agency.bookShow(show, 1);
    // compute price
    double price = cust.isFrequent()
            ? show.getDiscountPrice()
            : show.getPrice();
    this.show = show;
    this.owner = cust.name;
    this.price = price;
}
```

Question 5 (15 points)

Suppose that we have two concurrent bookings at the same agency, as follows:

```
public class Booking extends Thread {
    private static final Agency TICKET_MASTER = new Agency();
    private static final Location KINEPOLIS = new Location("Kinepolis␣1", 150);
    private static final Show ASSASSINS_CREED = new Movie(KINEPOLIS, 12.0);

    public static void main(String[] args) {
        TICKET_MASTER.addShow(ASSASSINS_CREED);
        Booking agnes = new Booking(2);
        Booking bert = new Booking(5);
        // start two bookings concurrently
        agnes.start();
        bert.start();
        try {
            agnes.join();
            bert.join();
        } catch (InterruptedException exc) {
            // do nothing
        }
        System.out.println("Tickets␣left:␣"
                + TICKET_MASTER.getSpace(ASSASSINS_CREED));
    }

    /** Number of tickets to be booked. */
    private final int count;

    /** Constructs a booking for a given number of tickets. */
    public Booking(int count) {
        this.count = count;
    }

    /** Runs the booking. */
    public void run() {
        TICKET_MASTER.bookShow(ASSASSINS_CREED, count);
    }
}
```

a. (*4 points*) What are the possible values printed by the `main` method? Elaborate your answer

b. (*3 points*) If we remove the entire try-block

```
try {
    agnes.join();
    bert.join();
} catch (InterruptedException exc) {
    // do nothing
}
```

do the possible printed values change, and if so, how? Explain your answer.

c. (*4 points*) If we change the `run` method into

```
public synchronized void run() {
    TICKET_MASTER.bookShow(ASSASSINS_CREED, count);
}
```

how does this affect the program? In particular, do the possible printed values change, and if so, how? Explain your answer.

d. (*4 points*) If we change the `run` method into

```
public void run() {
    synchronized (TICKET_MASTER) {
        TICKET_MASTER.bookShow(ASSASSINS_CREED, count);
    }
}
```

how does this affect the program? In particular, do the possible printed values change, and if so, how? Explain your answer.

Answer to question 5

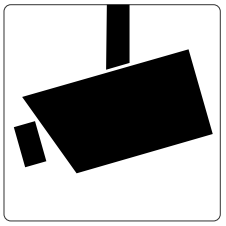
a. (*4 points;* -1 *if the number of booked tickets is reported rather than the remaining space. Max. 1 point for a correct answer with no or insufficient explanation.*) The possible answers are 143, 145 and 148. The first is the expected outcome where the bookings are done one after the other (in which case the order does not matter, as addition is commutative). The last two are caused by interleavings in which both `agnes` and `bert` read the number of booked tickets as 0, after which either `agnes` books first (resulting in 2 booked tickets and hence 148 remaining seats) or `bert` books first (resulting in 5 booked tickets and hence 145 remaining reats).

b. (*3 points; max. 1 point for a correct answer without explanation.*) Now it is possible that the `println` is reached before either `agnes` or `bert` have recorded their bookings; so in addition to the answers of the previous subquestion, 150 may be printed as well.

c. (*4 points. No deduction if it has been assumed that the* ***`try`*** *block is still absent, as in subquestion b. Max. 1 point for a correct answer without explanation.*) The `run` method is synchronised on the `Thread` object itself. This is obviously a different object for each thread, and so the synchronisation does not do any good. The possible answers are the same as in subequestion a.

d. (*4 pointss. No deduction if it has been assumed that the* ***`try`*** *block is still absent, as in subquestion b. Max. 1 point for a correct answer without explanation.*) Now the critical section is properly synchonised on one and the same object, which means that the interleaving causing the different answers in subquestion a is absent. The only possible remaining outcome is therefore 143.

Question 6 (15 points)



Imagine there's a company selling an internet-connected surveillance camera targeted at consumers. It is advertised as being easy to use: "simply connect it to your home network and you can manage your camera from all over the world in our cloud-based service!". To be able to access their camera, consumers need to go to the cloud service's website and log in with the password printed at the back of the camera.

a. *(3 points)* Suppose you managed to get hold of a couple of such cameras and you started to notice a pattern in the 12-character default passwords:

   - First, all passwords start with the text "IPC",
   - this is followed by either an "n" or a "k",
   - after which come 6 characters from the set {"a", "b", "c", "d"},
   - which is followed by 2 characters chosen from the whole lowercase (latin) alphabet ("a"-"z") and numbers (0-9).

   How many different passwords are possible with such a scheme? Explain your answer.

As people start to own more than one camera, having to log in separately for each of them is not very convenient. Consumers start to ask for a nice dashboard to manage and view the output of their cameras. To be able to support this, the company allows users to create a personal account at the company's cloud service. After choosing a password, consumers can then add the cameras to their accounts.

The company realizes it should not save the consumers' passwords in their database as is and decides to apply a hash to the password before storing it.

b. *(3 points)* Explain how a hashed password can still be used to authenticate users and explain why it is a good idea to apply a hash function to passwords before storing them.

c. *(3 points)* The company has the option to use either SHA1 or scrypt as the hash function to apply for storing the password. Which hash function would you advice the company to choose? Elaborate your answer.

The camera itself also contains software (it's so-called "firmware") and the company would like to be able to update this software over the internet. However, they of course do not want any changes to be made to this new firmware while in transit (intentionally or otherwise). In the initial version they send a cryptographic hash of the firmware alongside the firmware to attempt be able to check whether the firmware had been modified.

d. *(5 points)* Explain why this approach is not sufficient to prevent a "man-in-the-middle" attack. In addition, mention at least one cryptographic technique that does allow the detection of malicious changes.

Answer to question 6

a. *(3 points)* $1 * 2^1 \times 4^6 \times (26+10)^2 (= 10616832)$

b. *(3 points)* Many people (unfortunately) re-use their passwords. If the database with the plaintext passwords is ever leaked, then these users can be impersonated at such other services. If a

password is hashed, then it is nearly impossible to revert this process. One can still authenticate users by first storing the hashed password for this user, then when a user tries to log in with his/her password, this password is again hashed and compared to the value stored for the user. If they match, the user gave the same password as the in the setup phase, so he/should be allowed to continue. Note that passwords could still be recovered using brute-force approaches, see also next question.

c. *(4 points)* Scrypt. Should the database with hashed passwords ever be compromised, an attacker can try to launch a so called brute-force attack by simply trying many different passwords and comparing it with the hash values. Scrypt is made deliberately slow to make this approach unfeasible. Standard hash functions such as SHA1 on the other hand are designed to be fast and would make it possible to try many more passwords per second.

d. *(5 points)* An attacker can simply modify the data received and recompute the hash value. This way the receiver would not notice that something had changed. One should use Message Authentication Codes (MACs) to be able to detect malicious modifications. When a MAC is used, an attacker would not be able to compute a new "tag" for the modified data because it does not know the secret key shared between the sender and the receiver. Alternatively, digital signatures based on asymmetric cryptography (e.g., RSA, ECC) could be used.