TEST
Software Systems: Programming

course code: 201700117
date: 01 February 2018
time: 13:45 – 16:45

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 (15 points)

Consider the following two Java classes:

```
/**
 * ShopItem is a base class, representing items to be sold
 * in a shop. Actual ShopItems will be represented by sub classes.
 */
public abstract class ShopItem  {
    private String title = "";
    private double price;

    //@ requires title != null
    public void setTitle(String title) {
        this.title = title;
    }

    //@ pure ensures \result != null
    public String getTitle() {
        return title;
    }

    //@ requires price >= 0.0
    public void setPrice(double price) {
        this.price = price;
    }

    //@ pure ensures \result >= 0.0
    public double getPrice() {
        return price;
    }

    // Subclasses are required to override toString with an implementation
    // that shows all relevant information about that ShopItem
    @Override
    public abstract String  toString();
}
```

```
public class Book extends ShopItem   {
    private String author;

    //@ requires author != null && title != null;
    public Book(String author, String title ) {
        this.author = author;
        setTitle(title);
    }

    //@ pure ensures \result != null
    public String getAuthor() {
        return author;
    }

    //@ pure ensures \result != null
    @Override
    public String toString() {
        return "Book:␣" + getTitle() + ",␣by␣" + author;
    }
}
```

a. *(5 points)* Assume we have Java variable declarations:

```
- ShopItem item, bookItem;
- Book bk;
```

Which of the following statements are actually legal Java code, where by "legal" we mean that the Java compiler will accept this code.

~~*(1)*~~ `item = new ShopItem();`

(2) `bk = new Book("Shakespeare", "Hamlet");`

(3) `bookItem = new Book("Cervantes", "Don␣Quixote");`

(4) `bk.setPrice(45.00);`

(5) `String auth = bookItem.getAuthor();`

b. *(5 points)* Have a look at the JML specifications. In particular, look at the JML specification for the `getAuthor` and the `toString` method from `Book`. Do you believe the specifications are actually correct? **Explain your answer.**

c. *(5 points)* In our shop, we want to sell more than just "books". We need to define some more extensions of `ShopItem` like, for instance, a class for representing "movies". Create a Class "Movie" that extends `ShopItem` and that can represent *movie* information. It must represent a title, a director, a year of publication, and a price. "title" and "director" are `Strings`, "year" is an **`int`**, "price" is a **`double`**. You just have to give the Java code, so in particular, no JML is required.

Question 2 (20 points)

In this question, we build on the `ShopItem`, `Book`, and the `Movie` classes defined in the previous Question. `ShopList` is a new class that we use to represent *lists* of books and movies:

```
/**
 * A java.util.List of ShopItem's,
 * with some extra utility methods added.
 */
public class ShopList extends ArrayList<ShopItem>{
    /**
     * Returns the concatenation of the toString() representations
     * of all ShopItem's on the List, separated by newline characters.
     */
    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        for (ShopItem item : this) { // "this" refers to the ArrayList instance
            result.append(item); // appends item.toString()
            result.append('\n'); // appends a newline character.
        }
        return result.toString();
    }
}
```

Here is a table containing some books (the first three items) as well as some movies (the last two items).

Title	Author	Director	Year	Price
Shakespeare	Hamlet			35.00
Cervantes	Don Quixote			27.00
Austen	Pride and Prejudice			14.95
The Godfather		Coppola	1972	25.00
Gladiator		Scott	2000	35.00

a. *(4 points)* Create a Java class "`MyShop`", containing a method to initialize a `ShopList`. The name and signature of the method should be: **`public`** `ShopList initInventory()`.
Within this method, create an instance of `ShopList`, and fill it with the book data and movie data from the table above.

Finally, write a few lines of code where you create and initialize a `ShopList` typed variable called "`inventory`" that you initialize using your `initInventory` method, and print the contents of the `inventory` to the standard output.

Next we consider the following interface "`Predicate`". It is a simplification of the "real" Java 8 `Predicate` interface.

```
public interface Predicate<T> {
    boolean test(T object);
}
```

We provide one possible implementation `BargainFilter` of this interface. It selects books and movies that are no more expensive than a certain specified `limit`:

```
public class BargainFilter implements Predicate<ShopItem> {
    private double limit;

    public BargainFilter(double limit) {
        this.limit = limit;
    }

    @Override
    public boolean test(ShopItem item) {
        return item.getPrice() <= limit;
    }
}
```

b. *(2 points)* Consider the following code fragment:

```
austen = new Book("Austen", "Pride␣and␣Prejudice");
BargainFilter bargain = new BargainFilter(15.0);
```

What is the type, and what is the value of the following expression:

```
bargain.test(austen)
```

c. *(5 points)* Write your own `Predicate<ShopItem>` filter class, that we will call `FavoriteMovies`. It should select *movies* of a specified `director`, produced before a certain specified `year`. (The specified year is included, so it is "up to and including" the specified year.)

d. *(7 points)* Finally, we enrich the `ShopList` class (from Question 2) with an extra method for "filtering" the list. The required new method is like this:

```
/**
 * Creates and returns a new ShopList, consisting of all ShopItems from
 * this ShopList that satisfy the test from the specified filter Predicate
 */
public ShopList select(Predicate<ShopItem> filter) {
        //  ... (provide the code)
}
```

Provide the Java code for the `select` method. (*Just* the code for this method, leave out the remainder of the `ShopList` class).

e. *(2 points)* In part (a) of this question you created a `ShopList` typed variable called `inventory`. Give an expression that selects all movies from "`inventory`" that Coppola produced until 1980, and that cost no more than 30.00 euro.

Question 3 (20 points)

We extend our `Book` class from the previous questions:

```
public class ReviewedBook extends Book  {
   private List<String> reviews;

   public ReviewedBook(String author, String title ) {
      super(author, title);
      reviews = new ArrayList<String>();
   }

   /*
    *  Adds a (non-null) text to the list of reviews.
    *  Duplicate reviews will not be added a second time.
    */
   public void addReview(String reviewText) {
      if ( ! reviews.contains(reviewText)) {
         reviews.add(reviewText);
      }
   }

   public List<String> getReviews() {
      return reviews;
   }

   @Override
   public String toString() {
      StringBuilder display =
      new StringBuilder("Book:␣" + getTitle() + "␣by␣" + getAuthor());
      for (String rev : reviews) {
          display.append("\nReview:\n");
          display.append(rev);
      }
      return display.toString();
   }
}
```

a. *(10 points)* **Provide JML specifications for the constructor and all methods included in the `ReviewedBook` code.** You do not have to specify the methods from the `Book` class. But of course you should take into account the "requires" and "ensures" from the JML specifications for that base class.

Outside our "Shop" a separate system has been developed for collecting book reviews. It just records book reviews for certain book titles. The system is described by the following `BookReviews` class:

```
public class BookReviews {
    private Map<String, Set<String>> reviews =
    new HashMap<String, Set<String>>();

    /*@
     * requires bookTitle != null && reviewText != null
     * ensures getReviewsFor(bookTitle)   ==
     *          \old(getReviewsFor(bookTitle)).add(reviewText)
     */
    public void addNewReview(String bookTitle, String reviewText) {
        Set<String> reviewsForTitle = reviews.get(bookTitle);
        if (reviewsForTitle == null) {
            reviewsForTitle = new HashSet<String>();
            reviews.put(bookTitle, reviewsForTitle);
        }
        reviewsForTitle.add(reviewText);
    }

    /*@ pure
     *  ensures \result != null
     */
    public Set<String> getReviewsFor(String bookTitle ) {
        Set<String> revs = reviews.get(bookTitle);
        return (revs == null) ? Collections.emptySet() : revs;
    }
}
```

We want to use such `BookReviews` repositories, and your task is to add an appropriate method to the **`ShopList`** class. It has the following specification:

```
    /*@ requires repository != null
     * All ReviewedBooks in this ShopList will be updated, so as to include
     * all reviews found in "repository" for that ReviewedBook.
     */
 public void addBookReviews(BookReviews repository)
```

An example of usage, where we use again our "`inventory`" `ShopList`, would be:

```
BookReviews someReviews = new BookReviews();
someReviews.addNewReview("Hamlet", "Greatest␣play␣of␣all␣times!");
someReviews.addNewReview("Hamlet", "Supposedly␣great␣literature,␣but␣I␣fell␣asleep");
someReviews.addNewReview("Hamlet", "Greatest␣play␣of␣all␣times!");
someReviews.addNewReview("Don␣Quixote", "Funny␣tale");
inventory.addBookReviews(someReviews);
```

b. *(10 points)* **Implement this `addBookReviews` method,** as one of the methods of the `ShopList` class. You only have to provide the code for the method itself.

Question 4 (15 points)

a. *(3 points)* Declare a new Exception class "`BookReviewException`", which will be used to indicate that the same review would occurs twice in a `ReviewedBook`. (See Question 3 for `ReviewedBook`)

Reuse the existing Java construction mechanism to obtain Exceptions *with dedicated messages*.

b. *(7 points)* Change `ReviewedBook.addReview` as programmed in Question 3 so that it throws:

- a `BookReviewException` if the review text that is (attempted to be) added already occurs in the list of reviews for that book;
- an `IllegalArgumentException` (which is a standard Java subclass of `RuntimeException`) if the parameter `reviewText` 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 `addReview` declaration.**

c. *(5 points)* Write a code fragment that constructs a `ReviewedBook` "Ulysses" by "James Joyce", and add a review for that book. Include code to handle just those Exception(s) that *must* be handled, as required by the Java language, in order for the code to compile correctly. When an exception is thrown, its message should be printed on the standard output. Explain your choice of error handling.

Question 5 (15 points)

Consider the following classes "`Bookshop`" and "`BookBuyer`":

```java
public class Bookshop {
    private int bookCount;
    private double money = 0.0;

    public Bookshop(int stock) {
        bookCount = stock;
    }

    public void buy(int count, double price) {
        if (count <= bookCount) {
            bookCount = bookCount - count;
            money = money + price;
        }
    }

    public double getMoney() {
        return money;
    }
}
```

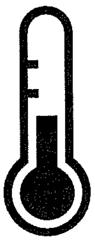
```
 1  public class BookBuyer extends Thread {
 2      private int bookWish;
 3      private Bookshop favoriteShop;
 4      private double moneyLeft;
 5
 6      public BookBuyer(int bookWish, double moneyLeft, Bookshop favoriteShop) {
 7          this.bookWish = bookWish;
 8          this.moneyLeft = moneyLeft;
 9          this.favoriteShop = favoriteShop;
10      }
11
12      @Override
13      public void run() {
14          favoriteShop.buy(bookWish, moneyLeft);
15      }
16
17      public static void main(String[] args) {
18          Bookshop popularShop = new Bookshop(7);
19          BookBuyer bert = new BookBuyer(3, 35.00, popularShop );
20          BookBuyer ernie = new BookBuyer(5, 45.00, popularShop );
21          bert.start();
22          ernie.start();
23          try {
24              bert.join();
25              ernie.join();
26          } catch (InterruptedException e) {
27          }
28          System.out.println("Profit:␣" + popularShop.getMoney() );
29      }
30  }
```

a. (*6 points*) What are the possible outcomes printed by the `main` method of `BookBuyer`? Which of them are erroneous? **Explain your answer,** in particular explain how each of the possible outcomes can happen, referring to the execution steps of the program.

b. (*4 points*) If we remove the *entire* **`try`**-block from `BookBuyer` (lines 23–27), 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 `Bookshop` such that the erroneous outcomes can no longer occur? **Explain why your solution works.**

Question 6 (15 points)



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. *(3 points)* Which of the following methods best used to protect the *integrity* of the messages? Explain your answer.

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

b. *(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.

c. *(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 10-12 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"}.
- this is followed by four digits,

d. *(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 rumors 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 of the method `String.split` says 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.

e. *(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.