Exam 12 June 2019, questions

Object Oriented Software Development (University of Melbourne)

THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

FINAL EXAM

Semester 1, 2019

SWEN20003 Object Oriented Software Development

Exam Duration: 2 hours

Total marks for this paper: 120

This paper has 8 pages

Authorised materials:
Students may NOT bring any written material into the room.
Students may NOT bring calculators into the room.

Instructions to invigilators:
Each student should initially receive a script book.
Students may NOT keep the exam paper after the examination.

Instructions to students:

- The exam has 5 questions across 3 sections, and all questions must be attempted. Questions should all be answered in the script books provided, **not** the exam paper. Start the answer to each question on a new page in the script book.
- Answer all questions on the right-hand lined pages of the script book. The left-hand unlined pages of the script book are for draft working and notes and will **not** be marked.
- Ensure your student number is written on all script books during writing time.
- The marks for each question are listed along with the question. Please use the marks as a guide to the detail required in your answers while keeping your answers concise and relevant. Point form is acceptable in answering descriptive questions. Any unreadable answers will be considered wrong.
- The section titled "Appendix" gives the documentation for several Java classes that you **can** use in your questions. You are not required to use all the listed classes and methods.
- Worded questions must all be answered in English, and code questions must all be answered in Java.

This paper will be reproduced and lodged with Baillieu Library.

Continued overleaf . . .

1 Short Answer (24 marks)

Question 1. **(24 marks)**

Answer the following questions with **brief**, **worded** responses Your answers should be written as **dot points**, **not essays**.

a) Explain the difference between a **primitive data type** and a **class**, and give examples to support your explanation. (4 marks)

b) How do `==` and `.equals` check equality? Give **examples** of when each is appropriate. (4 marks)

c) Define the terms **unit** and **unit test**, and describe the **purpose** of **unit testing**. (4 marks)

d) Describe the general type of problem solved by the **Strategy** and **Template** design patterns; in your answer, describe the components of the patterns and how they work together, and the similarities/differences between the two. (6 marks)

e) Describe the **purpose** and **behaviour** of the following stream pipeline. Give a **real-world example** where you might use this code. Be sure to address each line of code in your answer. (6 marks)

```
List<String> powerfulHeroes = allHeroes.stream()
                                .filter(h -> h.getPower() >= 9000)
                                .sorted((h1, h2) -> (h1.getPower() - h2.getPower()))
                                .map(Hero::getName)
                                .collect(Collectors.toList());
```

2 System Design (30 marks)

Question 2. **(30 marks)**

You have been recruited by the Forces of Injustice to develop and maintain EViL, the system they use to coordinate missions against their enemies.

The system uses locations to track people. A location consists of a latitude and longitude, and a name. A location can also calculate the distance from itself to another location.

A person consists of their name, their current location, and zero or more special powers. A person can also calculate the distance from themself to another person. A person can either be an ally or an enemy of the Forces of Injustice.

Our allies know all the locations where they may rest and recover, and will be targetting an enemy during their missions. Allies can also track their target's location. Enemies can be either heroes or sidekicks. Heroes contain no more information, but sidekicks are usually given a mentor, probably so we don't kidnap them...

Every power has a range, and may be offensive or defensive. Powers can also be activated, and when used will generally target another person.

For the questions below, you must rely **only** on the specification provided; you may make design decisions about method arguments, but do **not** make assumptions about behaviours that haven't been specified.

a) Using **only** the description given above, draw a UML class diagram for *EViL*. In your class diagram show the attributes (including type) and methods that are implied from the problem description. You must show privacy, class relationships, association directions and multiplicities. You do **not** need to show getters and setters, or constructors. (24 marks)

b) Describe **two** test cases you might write to test your implementation, stating specifically what *behaviour/component* you are testing, what an *input* might be, and the expected *output/result*. Do **not** write any Java code for this question.

Invalid example: *Test whether X object is created correctly/test whether Y variable is given a value.*
These tests are not related to your design or implementation, they are testing your ability to write code that works.

Invalid example: *Test whether a target is knocked out when hit by a power.*
This is not specified, or even vaguely suggested by the description provided, nor is it a logical assumption; not all powers will knock out a target.

Valid example: *Ensure powers can't be instantiated with a negative range.*
This is also not specified, but is a **logical** and **sensible** assumption; negative range doesn't make sense, so it is suitable to protect against. (6 marks)

 Continued overleaf ...

3 Java Development (66 marks)

Question 3. **(22 marks)**

For this question you will implement classes for a simple alarm system, using a *very simplified* version of the Observer pattern.

Note: do **not** use the `Observer` class or `Observable` interface to solve this problem.

a) Implement an `Alarm` class with the following: (12 marks)

i. Two attributes: `threshold` (the value where the alarm is activated) and `isActive` (whether the alarm has been activated; alarms are initially inactive).

ii. Appropriate initialization, accessor, and mutator methods.

iii. An `activate` method to *activate* the alarm, which also prints out the message `'Intruder alert!'`.

iv. A *notify* method which gets the value from a *sensor* and activates the alarm if the sensor's value is over the threshold.

b) Implement an abstract `Sensor` class with the following: (10 marks)

i. Two attributes: a `value` (the measured value), and `alarm` (an `Alarm`).

ii. Appropriate initialization methods.

iii. A `measure` method to *measure* a value from the world; *how* this might happen is not specified.

iv. An `updateMeasurement` method that *gets the measured value* and *notifies the alarm* when the measured value has changed by more than 1%.

v. A `toString` method that returns the sensor's `value`.

Question 4. **(21 marks)**

In this question you will implement the method

```
public String encode(String message, HashMap<Character,Character> map)
```

that takes a message and *encodes* its characters, where:

- `message` - the message to be encoded
- `map` - a dictionary that maps a character to its *encoded* character

***Algorithm*:**
Iterate through the characters in `message`, retrieve that character's *encoding* from `map`, and add the encoded character to the encoded message. The output of this method is the encoded message, where all characters from the original message have been encoded.

Note 1: All spaces must be removed from the input before encoding.

Note 2: If any character in the dictionary maps to itself, your method should create and throw an `InvalidDictionaryException`; you may assume this class exists.

Note 3: If any character is not present in the dictionary, you should create and throw an `InvalidCharacterException` exception; you may assume this class already exists.

Example 1 (Invalid Input):

Input:
```
dictionary.put('a',  'a');
encode("Java is great!", dictionary)
```

Output: `InvalidDictionaryException: 'a' can't map to itself`
since `'a'` is encoded to `'a'`.

Example 2 (Invalid Input):

Input:
```
dictionary.put('a',  'A');
encode("Java is great!", dictionary)
```

Output: `InvalidCharacterException: 'J' not present in dictionary`
since there are characters in the message that are not able to be encoded with the dictionary.

Example 3 (Valid Input):

Input:
```
dictionary.put('a', 'B');
dictionary.put('b', 'A');
dictionary.put('A', 'b');
dictionary.put('B', 'a');
encode("a b AB", dictionary);
```

Output: `"BAba"`. Note that all spaces have been removed, and all characters have been replaced with their corresponding encoding.

 Continued overleaf . . .

Question 5. **(23 marks)**

Hard Question! In this question you will implement a small object oriented system using generics. You may assume the `DoubleEnded<T>` interface exists. The `DoubleEnded<T>` interface represents a data structure that has a "front" and a "back". Any class that implements this interface can therefore return elements from either end.

a) Implement the `DoubleEndedString` class. This class should have an instance variable `string` of type `String`, and two integers `front` and `back` which *initially* store the index of the first and last characters of `string`. You should also implement an appropriate constructor for this class.

Your class definition should begin with:

```
public class DoubleEndedString implements DoubleEnded<String>
```

(6 marks)

b) Implement the following methods from the `DoubleEnded<T>` interface for the `DoubleEndedString` class:

`int elementsLeft()`	Returns the number of elements between `front` and `back`.
`boolean hasNext()`	Returns `true` if the object has more elements to return.
`void reset()`	Resets the `front` and `back` counters of the object.

(4 marks)

c) Implement the remaining functionality from the `DoubleEnded<T>` interface for the `DoubleEndedString`. You may assume the methods from the previous question exist, even if you haven't implemented them.

`T getFromFront()`	Returns the element given by the `front` index, and then increments `front` by one; returns `null` if no elements left.
`T getFromBack()`	Returns the element given by the `back` index, and then decrements `back` by one; returns `null` if no elements left.
`ArrayList<T> getFrontToBack()`	Returns all elements between indexes `front` and `back` as an `ArrayList`; returns `null` if no elements left.

Example: The code below demonstrates how the `DoubleEndedString` class could be used.

```
DoubleEndedString d = new DoubleEndedString("Hello");

System.out.println(d.getFromFront());       -> returns H
System.out.println(d.getFromBack());        -> returns o
System.out.println(d.elementsLeft());       -> returns 3
System.out.println(d.getFrontToBack());     -> returns [e, l, l]
System.out.println(d.getFromFront());       -> returns null
System.out.println(d.hasNext());            -> returns false
```

When `d` is created `H` is the current `front` character, and `o` the `back`. After retrieving these elements the indexes are moved, meaning there are only 3 characters left. The next method then retrieves all remaining elements (`'ell'`) as a list. At this point there are no more elements to retrieve, so trying to get an element returns `null`, and `hasNext()` returns `false`. (13 marks)

Continued overleaf . . .

4 Appendix

Hashmap

```
public class HashMap<K,V>
        extends AbstractMap<K,V>
        implements Map<K,V>, Cloneable, Serializable
```

The `HashMap` class, in the `java.util` package, implements the Map interface, which maps keys to values. Any non-null object can be used as a key or as a value.

`HashMap()`	Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
`boolean containsKey (Object key)`	Returns `true` if this map contains a mapping for the specified key.
`boolean containsValue (Object value)`	Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K, V>> entrySet()`	Returns a `Set` view of the mappings in the map.
`V get(Object key)`	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`Set<K> keySet()`	Returns a `Set` view of the keys contained in this map.
`V put(K key, V value)`	Associates the specified value with the specified key in this map.
`void putAll(Map<? extends K, ? extends V> m)`	Copies all of the mappings from the specified map to this map.
`boolean remove(Object key)`	Removes the mapping for the specified key from this map if present.
`int size()`	Returns the number of key-value mappings in this map.

Continued overleaf . . .

ArrayList

```
public class ArrayList<E>
        extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, Serializable
```

The `ArrayList` class, in the `java.util` package, a resizable-array implementation of the `List` interface.

`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`boolean add(E e)`	Appends the specified element to the end of this list.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`boolean equals(E element)`	Compares the specified object with this list for equality.
`E get(int index)`	Returns the element at the specified position in this list.
`int lastIndexOf(Object o)`	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E remove(int index)`	Removes the element at the specified position in this list.
`boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present.
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element.
`int size()`	Returns the number of elements in this list.

— End of Exam —