THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

FINAL EXAM

Semester 2, 2018

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 contain no more than **four** dot points, **not** essays.

a) Explain the difference between abstraction using inheritance, and abstraction using interfaces. In your answer, describe **one** application for each type that demonstrates your explanation. (4 marks)

b) List and explain any **two** of the symptoms of poor software design. (4 marks)

c) Describe the general type of problem solved by the *Observer* design pattern; in your answer, describe the components of its design and how they work together. (4 marks)

d) Explain the terms private and protected, and why we *generally* prefer private attributes over protected attributes. (4 marks)

e) Explain why we use the `equals` method when checking equality of *objects* and not `==`. (4 marks)

f) 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. (4 marks)

```
Recipe recipe = new Recipe("Chicken Caesar Salad");

List<Item> ingredientsToBuy = catalogue.stream()
                        .filter(item -> recipe.contains(item))
                        .filter(item -> item.isAvailable())
                        .sorted((m1, m2) -> m1.getName().compareTo(m2.getName()))
                        .collect(Collectors.toList());
```


Continued overleaf . . .

2 System Design (30 marks)

Question 2. **(30 marks)**

You have joined budding games company Eleanorus as lead designer for their new game *EarthEdge*.

EarthEdge has two main types of game assets: items, and players. All game assets are defined by their position in the world, and the image (or 2D model) that represents them.

The two main item types are chests and weapons. All items may be *active* or *destroyed*. Weapons also have damage and range, and may be held by *at most* one player. Chests may be *open* or *closed*, and can also hold any number of weapons.

All players keep track of their health and experience level, and all players can move through the world, and can use their weapon to attack. Players also share a count of how many other players are active in the game. Players can be either *human* or *AI*, and human players all have a username. AI players have no further characteristics.

Finally, items and human players can be *interacted with* by the person playing the game.

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 *EarthEdge*. In your class diagram show the attributes (including type) and methods that are implied from the problem description. You must show class relationships, association directions and multiplicities. You do **not** need to show getters and setters, or constructors.

Note: You may assume that `Position`, `Image`, and `Input` are provided to you as libraries; you do not need to include them as separate classes in your UML diagram. (24 marks)

b) Describe **two** test cases you might write to test your design, 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. (6 marks)

Continued overleaf . . .

3 Java Development (66 marks)

Question 3. **(22 marks)**

For this question you will implement classes for a basic music simulator. You may assume that the enum `Type` exists with values `Brass`, `Woodwind`, `String`, `Percussion`.
The `Type` class also has the following method:

`int getHighestNote()`	Returns the frequency of the average highest note possible by that type.

a) Implement an **immutable** `Instrument` class with the following: (7 marks)

i. Attributes to represent the instrument's `type`, whether it is *first-* or *secondhand* (new or used), and a numeric `id`.

ii. Appropriate initialization, accessor, and mutator methods.

iii. A `compareTo` method that results in `Instrument` objects being arranged in *increasing* order of the highest note given by the `Type`; specifically, if `i1`'s highest note is higher than `i2`'s, `i2.compareTo(i1)` should be negative.

iv. A `toString` method that returns the `type` and `id` of the `Instrument`; for example: `"BRASS: id=1"`.

v. An `equals(Instrument other)` method that returns true when two instruments are the same type, are both either new or used, and share the same ID.

b) Implement a `Band` class with the following: (15 marks)

i. Two attributes: a `name`, and a list of `Instrument` objects (the band)

ii. Appropriate initialization, accessor, and mutator methods.

iii. A method to add an `Instrument` to the band.

iv. A method to remove an `Instrument` from the band.

v. A method to sort the instruments in the band by highest note.

vi. A method to select an `Instrument` for a solo; the soloist is the `Instrument` with the highest note. This `Instrument` should be returned as the output of the method.

vii. A `toString` method that returns the `type` and `id` of all the `Instrument`s in the band; for example: `"[BRASS: id=1, BRASS: id=2, WOODWIND: id=3]"`.

Continued overleaf . . .

Question 4. **(24 marks)**

In this question you will implement the method

```
public String filterAndConcat(String s1, String s2, String illegalChars, int minFreq)
```

that takes two strings and concatenates them together, where:

- `s1` - the first `String` to be concatenated
- `s2` - the second `String` to be concatenated
- `illegalChars` - a `String` that contains characters that **should not** appear in the final output
- `minFreq` - the minimum frequency required for a character to be included in the output

***Algorithm*:**
Concatenate ("add") the two inputs `s1` and `s2` into a single `String`, then count the occurrence of each character. The output should then contain **only** the characters that appear in the concatenated `String` with **at least** `minFrequency`.

Note 1: All spaces are removed from the inputs before concatenation.

Note 2: If any illegal characters are found in the inputs, your method should create and throw an `IllegalCharacterException` exception; you may assume this class already exists.

Example 1 (Invalid Input):

Input: `filterAndConcat("Hello", "Hell", "Hll", 2)`

Output: `IllegalCharacterException: 'H' found in input string.`, since one of the illegal characters (`'H'`) was found in one of the inputs.

Example 2 (Valid Input):

Input: `filterAndConcat("Hello", "Hell", "", 2)`

Output: `"HellHell"`, since every character but `'o'` appears at least as many times as `minFreq`.

Example 3 (Valid Input):

Input: `filterAndConcat("You know nothing", "John Snow", "", 9)`

Output: `""`, since none of the characters meet the frequency threshold.

Example 4 (Valid Input):

Input: `filterAndConcat("Android good", "Apple bad", "", 3)`

Output: `"dodoodd"`, noting that spaces have been removed from the inputs, and that neither `'a'` or `'A'` appears in the output as they are *different* characters.

Continued overleaf . . .

Question 5. **(20 marks)**

Hard Question! In this question you will implement a small object oriented system using generics. Assume the `Transformer<R>` interface exists, which declares one abstract method:

`R transform(R input)`	Performs some transforming operation on the `input` object of type `R`.

a) Implement the class `StringTransformer`, including the `transform` method. This class should also have an instance variable `n`. The `transform` method transforms `String` objects by returning the first `n` characters of the input (as a `String`).
Your class definition should begin with:

```
public class StringTransformer implements Transformer<String>
```

(5 marks)

b) Implement the `TransformingList<R, T extends Transformer<R>>` class where `R` is any arbitrary type, and `T` is any class that can *transform* objects of type `R`.
This class should have two instance variables: an `ArrayList` of type `R`, and a `transformer` of type `T`. Include an appropriate constructor.
Your class definition should begin with:

```
public class TransformingList<R, T extends Transformer<R>>
```

For example, `TransformingList<String, StringTransformer>` stores a list of `String` objects, and transforms them using the `StringTransformer` object. (4 marks)

c) Implement the following methods for the `TransformingList` class:

`void add(R element)`	Inserts the argument `element` at the end of the list.
`R transformIndex(int index)`	Transforms the `index`'th object in the list and returns it.
`ArrayList<R> transformList()`	Returns a new list containing the transformed values of each element in the list.

`String toString()`	Returns a `String` representing the contents of the `ArrayList`. The `String` should be in the form
	`Element1 -> Transformation1`
	`Element2 -> Transformation2`

Example:

```
StringTransformer transformer = new StringTransformer(5);
TransformingList<String,StringTransformer> list = new TransformingList(transformer);

list.add("Hello World");
list.add("Java");
list.add("I am Iron Man");

System.out.println(list.transformIndex(2));
System.out.println(list);
```

The output of this code would be `"Java"`, followed by `"Hello World -> Hello"`, `"Java -> Java"`, and `"I am Iron Man -> I am "`, on separate lines. Since the transformer is constructed with `n = 5`, transforming each element gives a substring of (at most) length 5.

Note: you need to correctly handle the case where `n > input.length()`. (11 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 —