Australian National University

2016 Semester 1 - Final Examination

Software Construction

(COMP2500/COMP6442)

Writing Period: 3 hour duration
Study Period: 15 minutes duration
Permitted Materials: One A4 page with notes on both sides.
Note also the standard lab tools are available including: Java, eclipse, gedit, vim, emacs, git, umbrello, dia, gcc, man, the calculator on the computer, ...
The Java API is available at file:///usr/share/doc/openjdk-7-jre-headless/api/index.html

NO calculator permitted (physical electronic device).

Please Read The Following Instructions Carefully.

This exam will be marked out of 100 and consists of 4 questions. Questions are of unequal value. The value of each question is shown in square brackets. Questions that are partitioned into parts show the number of marks given to each part within square brackets.

Students should attempt all questions. Answers must be saved into the question's directory (Q1, Q2, Q3, Q4) using the file(s) described in the question statement. Marks may be lost for giving information that is irrelevant.

Network traffic may be monitored for inappropriate communications between students, or attempts to gain access to the Internet.

The marking scheme will put a high value on clarity so, as a general guide, it is better to give fewer answers in a clear manner than to outline a greater number of less clear answers.

Question 1 [40 marks]

Highest marks are gained by providing clear, concise, and short answers. Save your answers in the text file 'Q1answers.txt' in the directory Q1 (this file is already created with places to add your answers, so edit the file 'Q1answers.txt' and save your answers directly into it). Use your favourite editor for e.g. vim, gedit etc. Please make sure that this file is saved both as you progress through the exam and before the exam ends.

i. [4 marks] Two different authentication methods used by SSH are a) using a password and, b) using a public key. Describe in detail the public key authentication method. Include descriptions for each phase involved in both the client and server.

ii. [4 marks] What is the advantage of using a factory design pattern (such as the factory method or abstract factory) over just using “new” in the code using the products of this factory?

iii. [4 marks] Explain the difference between coupling and cohesion in terms of software design. Why is it advantageous to have low coupling and high cohesion in a design?

iv. [4 marks] Explain the differences between a distributed revision control system such as git and client-server system such as svn.

v. [4 marks]

   a) Using git, describe the steps involved in setting up a local repository, adding files to it and pushing them to a remote repository such as a gitlab server.

   b) Describe the three main components found within the .git folder present in a git repository.

vi. [4 marks] Explain the Android Security Sandbox. What advantages does it provide?

vii. [4 marks] Describe the four Android App Components. What are features common to all components? How can components communicate with each other?

viii. [4 marks] What are the advantages of storing persistent data using XML or JSON files rather than using Serializable Java Objects?

ix. [8 marks]

Only do one of the two sections below.

[COMP 2500 Students]

a) Explain the concept of a git rebase operation.

b) Describe the advantages and disadvantages of performing a git rebase operation.

c) When would it be appropriate to perform a rebase?

d) List the differences between a git rebase and a git merge operation

[COMP 6442 Students]

Software Design Patterns can be categorized into three broad groups: Creational, Structural and Behavioral Design Patterns.

a) Describe the features of each category

b) Give an example of a Design Pattern from each category and explain its Intent

Question 2 [30 marks]

The aim of this question is to complete a mini software development project. This involves: i) Understanding the requirements for a simple application, ii) Creating a software design in UML using an appropriate Design Pattern, iii) Implementing the application using Java, iv) Demonstrating the use of some software engineering tools such as git, Makefiles v) Writing JUnit test cases and testing the implementation, vi) Writing appropriate documentation. Your code and answers must all be included in the Q2 directory.

Design and implement a program to model a university car park building. The building must have the following requirements:

- The building can park Vehicles of 3 types: Cars, Motorcycles and Minivans
- Each parking spot may be Small, Compact or Large in size
- A Car may occupy a Compact or Large spot
- A Motorcycle may occupy only a Small spot
- A Minivan may occupy only a Large spot

The program to model the university car park building must have the following requirements:

- The program must be named Carpark
- The program must accept available parking spot numbers using 3 input arguments as given below
    - The first argument represents the number of Small parking spots
    - The second argument represents the number of Compact parking spots
    - The third argument represents the number of Large parking spots
- For e.g. the program may be invoked in the following way: `$ java Carpark 20 10 20`
- After invocation, the program must wait for further input from System.in, then
    - The program must emulate the entrance of the car park with an automatic ticketing system which issues an appropriate entry ticket to a Vehicle based on its type, size and the amount of free parking spots available for that Vehicle type
    - The program must prompt the user for what type of Car the user wants to park. For e.g. "What vehicle would you like to park?"
    - The user may enter either Car, Motorcycle or Minivan
    - The program must keep track of the number of free parking spots

- Depending on the availability of parking spots, the program must either accept the user's vehicle saying "Welcome, please continue to park" or decline saying "Sorry, there are no spots available for your vehicle."
- If a Car wants to park, the program must tell the car whether to park in a Compact spot or a Large spot. Priority must be to fill up the Compact spots first. For e.g., "Welcome, please continue to park in a Compact spot"
- If the user enters, "Bye", the program must exit.

An example session with the Carpark program would look like this:

```
$ java Carpark 2 0 1
> What vehicle would you like to park?
Car
> Welcome, please continue to park in a Large spot
> What vehicle would you like to park?
Minivan
> Sorry, there are no spots available for your vehicle.
> What vehicle would you like to park?
Bye
```

Your tasks are:

a) [10 marks] Write a basic working version of the Carpark program using Java. Sample code is provided in Carpark.java in the Q2 directory to get you started with argument parsing and accepting user input from the command line.

b) [3 marks] Creating a software design for the Carpark in UML. Export the diagram to a file called UML.png in the Q2 directory. Provide a short summary explaining your design in the file named DesignSummary.txt provided in the Q2 directory

c) [2 marks] Setup and use git within the Q2 directory for revision control and make at least 3 commits.

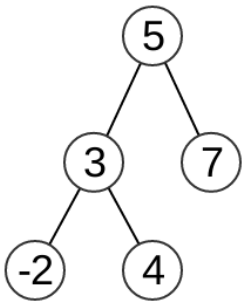
d) [3 marks] Write JUnit unit tests to test the Carpark program. Your tests must account for statement, branch and path coverage.

e) [2 marks] Write a Makefile for building and testing the Carpark program with 3 targets: make, make clean, make test

f) [10 marks] Overall quality of submission. This includes aspects such as properly formatted code and UML diagram, enough detailed comments within code, appropriately named variables/field/methods/classes, meaningful comments for git commits and use of appropriate design patterns.

Question 3 [20 marks]

The aim of this question is to evaluate your understanding of Tree data structures. Include all code written for this question in the Q3 directory. The diagram below represents a Binary Search Tree.



An abstract class BST is provided in the file BST.java in the Q3 directory.

a) [8 marks] Extend the BST abstract class to create concrete classes which hold integer values, in its nodes along with its left and right children. Implement all the methods specified. Save the newly created Java files in the Q3 directory.

b) [8 marks] Add a recursive function,

`int findCountSum(int targetSum)`, to the BST abstract class. The purpose of this function is to find and count the number of paths from any BST node down to a leaf node which sum to the given value targetSum. This sum includes both the starting node and the leaf node's values. Note that a single leaf node also counts as a path. Implement this function in your concrete classes.

For the BST given in the diagram and the following targetSum values, findCountSum would return as follows:

- When targetSum = 12, findCountSum = 2
- When targetSum = 7, findCountSum = 2
- When targetSum = -1, findCountSum = 0
- When targetSum = 1, findCountSum = 1
- When targetSum = 0, findCountSum = 0
- When targetSum = 6, findCountSum = 1

Write a demo program, BSTDemo in the file BSTDemo.java provided in the Q3 directory with a main() function to construct the BST given in the diagram by calling the insert function similar to the code snippet shown below:

```
BST bt = new MyBST(5);

bt = bt.insert(3);
bt = bt.insert(7);
bt = bt.insert(-2);
```

```
bt = bt.insert(4);
bt = bt.insert(10);
bt = bt.remove(10);

System.out.println("size: " + bt.size());
System.out.println("height: " + bt.height());
System.out.println("findCountSum: " +
                   bt.findCountSum(targetSum));
```

BSTDemo must accept the targetSum as an input parameter and output the size, height and findCountSum.

For e.g.,

```
$ java BSTDemo 7

size: 5

height: 3

findCountSum: 2
```

c) [4 marks] Write a Bash test harness to test the demo program using the targetSum inputs and expected findCountSum outputs as given above.

Question 4 [10 marks]

Code for representing and showing expressions has be included into the Q4 directory. There are four types of expressions: literal expressions (LitExp) which are just integer values like 5, -4, 8..., variables (VarExp) which are just strings like "x", "y", "count"..., summation expressions (SumExp) which are a list of expressions that are added together like (1+4), (1 + x + y)..., and product expression (MultExp) which are a list of expressions that are multiplied together.

This is a recursive data structure where the expression is stored as a tree. Sometimes within a program when you write expressions you end up with repeated sub-expressions. Such code can be simplified by creating a new variable which first evaluates this sub-expression then assigns it to a variable. This variable can then be used to replace the sub-expression within the original expression.

For example, consider the expression "((x+1)*(x+1))" we could first evaluate "v=(x+1);" then the expression reduces to "(v*v)". This will generally make code more readable and also reduce the number of operations performed. Your task is to implement the function that will find an assignment that reduces the total operation count. This involves adding your solution within the "findReducingAssignment" method within Exp.java in the Q4 directory. This is the ONLY code you should modify. Do not add to or modify any other methods/classes.

Use the variable "v" as the name of the replacing variable. You may assume the original expression will not contain any variables "v".

Use the "countOperations" method to count the number of operations within expressions. The reduced operation count will be the operation count within the expression of the assignment plus the operation count of the expression which has had the sub-expression replaced with variable "v". If there is no assignment that reduces the total operation count then the "countOperations" method must return "null". There may be more than 1 assignment that minimizes the total number of operations. In such cases select the sub-expression within the assignment which first occurs in a breadth-first traversal of the expression tree.

Once you have completed your solution, when you run "DemoExpressions" you should see:

```
The expression (1 + 1) has 1 operation and there is no reducing
assignment.
The expression (x + 1) has 1 operation and there is no reducing
assignment.
The expression ((x + 1) * (x + 1)) has 3 operations if the
assignment v = (x + 1); was
made the expression would be reduced to (v * v) giving 1 + 1 =
2 operations.
The expression ((x + 1) * (1 + x)) has 3 operations and there
is no reducing assignment.
The expression ((2 * x) + (2 * x) + (2 * x)) has 5 operations
if the assignment v = (2 * x); was
```

```
made the expression would be reduced to (v + v + v) giving 1 +
2 = 3 operations.
The expression (((2 * x) * (2 * x)) + ((2 * x) + y) + (2 * x))
has 8 operations if the assignment v = (2 * x); was
made the expression would be reduced to ((v * v) + (v + y) + v)
giving 1 + 4 = 5 operations.
The expression ((2 * x) + (2 * x) + (3 * y) + (3 * y)) has 7
operations if the assignment v = (2 * x); was
made the expression would be reduced to (v + v + (3 * y) + (3 *
y)) giving 1 + 5 = 6 operations.
The expression (((2 * x) + (2 * x)) + (3 * y) + (3 * y)) has 7
operations if the assignment v = (3 * y); was
made the expression would be reduced to (((2 * x) + (2 * x)) +
v + v) giving 1 + 5 = 6 operations.
The expression (((2 * x) + (3 * y)) + (2 * x) + (3 * y)) has 7
operations if the assignment v = (2 * x); was
made the expression would be reduced to ((v + (3 * y)) + v + (3
* y)) giving 1 + 5 = 6 operations.
The expression (((2 * x) + (3 * y)) + (3 * y) + (2 * x)) has 7
operations if the assignment v = (3 * y); was
made the expression would be reduced to (((2 * x) + v) + v + (2
* x)) giving 1 + 5 = 6 operations.
The expression (((x + 1) * (x + 1)) + ((x + 1) * (x + 1))) has
7 operations if the assignment v = ((x + 1) * (x + 1)); was
made the expression would be reduced to (v + v) giving 3 + 1 =
4 operations.
```

Hints:

- The "replace" method returns a new expression which has the given variable taking the place of the given sub-expression.
- "LinkedList" is part of Java collections and provides a simple queue data structure.