

THE AUSTRALIAN NATIONAL UNIVERSITY

First Semester 2006 — Midsemester Examination

**COMP2100/2500
Software Construction**

Writing Period: 60 minutes

Study Period: 15 minutes

Permitted Materials: One A4 page with notes on both sides

Answer all questions

*Write your answers in the boxes provided in this booklet. Only those answers written in this booklet will be marked. There is additional space at the end of the booklet in case the boxes provided are insufficient. Label any answers you write at the end of the booklet to indicate which question they refer to. Do not remove this booklet from the examination room. **Do not write in red ink anywhere in the booklet.***

Name (family name first):

Student Number:

Official use only:

Q1 (10)	Q2 (8)	Q3 (7)	Q4 (10)	Total (35)

QUESTION 1 [10 marks]

- (a) Give the rationale for using a Version Control System in the software development process (why VCS are used?)

QUESTION 1(a)	[1 mark]

- (b) Name two *test selection strategies* used in the structural (aka white box) testing. Characterise them briefly. Which is more thorough? Which is more practical?

QUESTION 1(b)	[2 marks]

- (c) Name and characterise three kinds of *static code analysis*.

QUESTION 1(c)	[2 marks]

- (d) Explain why the Visitor pattern is not necessarily a good choice if the class hierarchy the visitor objects are meant to operate on is volatile, *i.e.* subject to frequent change.

QUESTION 1(d)	[2 marks]

- (e) What is a *widget* (as the notion from Graphical User Interface programming)?

QUESTION 1(e)	[1 mark]

- (f) Explain the meaning of the Java Swing *handler*. What is a standard way to implement handlers in Java Swing programming.

QUESTION 1(f)	[2 marks]

QUESTION 2 [8 marks]

This question is about recursive tree data structures for representing algebraic expressions. We use the 'Expression Version 4' design (with the Visitor pattern) studied in lectures and in Lab 2. Your task here is to add the notion of *variables* to the system.

A variable has (a) a name, i.e. a textual representation, which is a single letter, represented in Java using the type `char`; and (b) a value, which is an integer, represented in Java using the type `int`. The set of variables used in evaluating an expression is stored in a `Hashtable` which maps variable names to variable values.

You may assume that the main class has a field:

```
Hashtable variables = new Hashtable();
```

which is initialised thus:

```
variables.put('a', 1);
variables.put('b', 2);
variables.put('c', 3);
variables.put('d', 4);
```

So, for example, the expression

```
System.out.println((Integer) variables.get('c'))
```

will print the number 3. (This uses the automatic boxing and unboxing features of Java 1.5, but not generics. We could implement the lookup table with a `Hashtable<Character, Integer>` but since we haven't used this in labs or assignments, we'll stick to the old form for now.)

The `Expression` class looks like this.

```
public abstract class Expression {
    public abstract void accept(Visitor v);
}
```

Operations on expressions are performed by visitors, which must all implement the following interface.

```
public interface Visitor {
    public void visit(Constant c);
    public void visit(Addition a);
    public void visit(Variable v);
    public void visit(Multiplication m);
}
```

Class `Expression` will have four effective subclasses, representing constants, variables, sums and products of two expressions. Two of these, class `Constant` and class `Variable`, have been implemented:

```
public class Constant extends Expression {  
  
    int value;  
  
    public Constant(int v) { value = v; }  
  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```

and

```
public class Variable extends Expression {  
  
    char name;  
  
    public Variable(char c) { name = c; }  
  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```

In the class `Variable`, the field `name` represents the variable name (like `x` or `y`), the constructor is used to initialise it, and the `accept` method implements the `Visitor` call.

- (a) The `Variable` implementation is not perfect. How this implementation may be improved to better express the property that the name of the `Variable` object must stay the same during the object life time? Include only the changed parts.

QUESTION 2(a)	[1 mark]

QUESTION 3 [7 marks]

In this question you have to write a JUnit test class (*test code*) for a Java class (*production code*). The Java class `Point` defines two constructors for creating an instance of this class, and several methods which can be called on such instances.

```
public class Point {

    private double x;
    private double y;

    /** constructor for the (0,0) */
    public Point() {...}

    /** regular constructor */
    public Point(double x, double y) {...}

    /** point in coordinates relative to the point
     * (the centre of coordinate frame moved to p) */
    public Point relativeTo(Point p) {...}

    /** point moved on the vector v */
    public Point movedOn(PlanarVector v) {...}

    /** standard getter methods for point coordinates */
    public double getX() { return this.x; }
    public double getY() { return this.y; }

    /** equals test for this and Point r */
    public boolean equals(Object o) {...}

    ... ..

    /** distance from this Point to another Point p */
    public double distance(Point p) {...}

    public String toString() {

        return "Point (" + x + ", " + y + ")";
    }
}
```

- (a) The test code utilising the JUnit framework will contain several test methods and, possibly, *fixture* methods: `setUp()` and `tearDown()`. Explain the usefulness of the fixture methods, and how many times each of them (`setUp()` and `tearDown()`) are called during the test run of the class which contains them.

QUESTION 3(a)	[1 mark]

- (b) What assert method will you use to test the `distance()` method given that it returns the double value.

QUESTION 3(b)	[1 mark]

- (c) What factors determine how many assert calls you put into a test method.

QUESTION 3(c)	[1 mark]

QUESTION 4 [10 marks]

This question is about the PSP. A student has completed a PSP Project Plan Summary form for Program #2 and now starts working on Program #3.

Fill in the 'Plan' column of the PSP Project Plan Summary form based on a size estimate of 60 lines of code with maximum of 70 and minimum of 50. Take all other plan data from the completed form for Program #2.

The student takes 24 minutes to complete the Planning phase, 36 minutes to complete the Design phase, 36 minutes to complete the Code phase, 24 minutes to complete the Compile phase, 96 minutes to complete the Test phase, and 24 minutes to complete the Postmortem phase.

The student injects 7 defects in the Design phase, 9 in the Code phase and 4 in the Test phase.

The student removes 2 defects in the Code phase, 2 in the Compile phase and 16 in the Test phase.

The finished program is 40 lines of code.

Record all this information on the form.

Now complete the rest of the form: all totals in the 'Actual' column, plus everything in the 'To Date' and 'To Date %' columns.

You do not have to fill in the 'greyed-out' parts of the form.

(Note that the numbers have been carefully chosen so that all the results come out as simple round numbers. You do not need a calculator.)

Here is the student's PSP Project Plan Summary form for Program #2, with all details completed.

Here is a new PSP Project Plan Summary form for Program #3 that you have to complete using data taken from the form for Program #2 on page 12 and the description of events on page 11.

