Answer all questions in the space provided here.

Question	Points	Score
Question 1	4	
Question 2	6	
Question 3	10	
Question 4	10	
Question 5	10	
Question 6	10	
Total	50	

Part A. Unit Testing

Question 1 ..... **4 marks**

There are different units of measurement for temperature. These include **Kelvin**, the absolute, thermodynamic temperature scale, **Celsius**, a commonly used measurement in the world, and **Fahrenheit**, which is used in only a small number of countries. The conversion between Celsius and the other two units are given in the table below.

Table 1: Conversion between the Celsius scale and other scales.

	From **Celsius**	To **Celsius**
Fahrenheit	$°\text{F} = °\text{C} \times \frac{9}{5} + 32$	$°\text{C} = (°\text{F} - 32) \times \frac{5}{9}$
Kelvin	$°\text{K} = °\text{C} + 273.15$	$°\text{C} = °\text{K} - 273.15$

In this question, develop test cases for a program that converts temperatures between different units, using **decision table**-based techniques.

The program takes as input three parameters: (1) a number, (2) a character that represents the input unit of measurement, and (3) a character representing the output unit (C for Celsius, F for Fahrenheit and K for Kelvin). The program's output is a number in the output unit, calculated according to the rules in Table 1 above.

Question 2 ..... **6 marks**

In Lecture 3 we introduced Boundary Value Testing (BVT) technique. In BVT, we test the values around the allowed range of each input variable. Depending on robustness requirements and fault occurrences assumptions (single or multiple), test cases take different sets of values for each variable.

For a specification with n number of variables over some numerical domains, let's denote these variables x_i, where $1 \leq i \leq n$. Remember that the notation for the cardinality of a set S (the number of elements in S) is $\#S$. Each variable x_i has a number of valid ranges that are bounded from both sides (in other words, none of the ranges goes down to $-\infty$ or up to ∞). The number of valid ranges is denoted r_i for variable x_i. Hence, there are gaps between the ranges.

As a result, the minimum number of test cases for BVT testing is a function of the number of valid ranges. For example: for the simplest case when $r_i = 1$ for any variable x_i, there are only 2 invalid value ranges outside of the valid range (one below the minimum, and one above the maximum). With single fault assumption, the minimum number of tests is then $4n + 1$.

Given the above notation, (1) work out a formula for the minimum number of test cases for **robust BVT** (but **not** worst-case), and (2) briefly explain why.

Question 3 ..... **10 marks**

The A* algorithm is a widely-used, heuristics-based graph search algorithm. It finds a path from a given start node to a given goal node. A* uses a distance-plus-cost heuristic function $f(x) = g(x) + h(x)$ to determine the order of traversal. For the current node x, $g(x)$ is the cost from the starting node to x, and $h(x)$ is an estimate of the distance to the goal from x.

```
   Input: start node
   Input: goal node
   Output: a path from start to goal
 1 closedset ← ∅
 2 openset ← {start}
 3 came_from ← ∅                                                    // the empty map
 4 g(start) ← 0                                  // Cost from start along best known path
 5 h(start) ← heuristic_cost_estimate(start, goal)
 6 f(start) ← g(start) + h(start)                // Estimated total cost from start to goal.
 7 while openset ≠ ∅ do
 8 |   current ← node, where node ∈ openset ∧ ∀x: openset • f(node) ≤ f(x)
 9 |   if current = goal then
10 |   |   return reconstruct_path(came_from, goal)
11 |   end
12 |   openset ← openset \ {current}
13 |   closedset ← closedset ∪ {current}
14 |   for neighbour ∈ neighbour_nodes(current) do
15 |   |   if neighbour ∈ closedset then
16 |   |   |   continue
17 |   |   end
18 |   |   tentative_g_score ← g(current) + dist_between(current, neighbour)
19 |   |   if neighbour ∉ openset then
20 |   |   |   openset ← openset ∪ {neighbour}
21 |   |   |   h(neighbour) ← heuristic_cost_estimate(neighbour, goal)
22 |   |   |   tentative_is_better ← true
23 |   |   else if tentative_g_score < g(neighbour) then
24 |   |   |   tentative_is_better ← true
25 |   |   else
26 |   |   |   tentative_is_better ← false
27 |   |   end
28 |   |   if tentative_is_better = true then
29 |   |   |   came_from ← came_from ∪ (neighbour, current)
30 |   |   |   g(neighbour) ← tentative_g_score
31 |   |   |   f(neighbour) ← g(neighbour) + h(neighbour)
32 |   |   end
33 |   end
34 end
35 return failure
```

Algorithm 1: The A* algorithm.

Note that *reconstruct_path*, *neighbour_nodes*, *dist_between* and *heuristic_cost_estimate* are functions defined elsewhere.

(Continued overleaf)

(a) (5 marks) Draw the program graph for the above function.

(b) (5 marks) Recall that McCabe's essential complexity measures how *unstructured* the logic of a program is by calculating the Cyclomatic complexity of the condensed program graph. In this part, (1) draw the final condensed graph for the program graph you came up with in part (a) above, and (2) calculate the Cyclomatic complexity of the condensed graph you draw.

Part B. Integration Testing & System Testing

During the semester we've been working on the *Who's coming?* project. In this project, the goal is to create a simple Web application for event management, dealing with `Persons`, `Events` and `Invitations`. In the code base of the last part of the project, individual components have been integrated together to make the Web application functional.

With some simplification, the loading of the `Event` page is shown below in Figure 1. The component **Router** is responsible for routing HTTP requests to the appropriate *resource* that handles them. As the name suggests, **EventResource** is responsible for creating pages to present event details. Once authentication and authorisation are determined to be required, the resource consults the `LoginManager` to grant/refuse access based on user input. The `LoginManager` in turn invokes `UserDAO` to load user details given the user name. If access is granted, the resource then invokes `EventDAO` to obtain event details. If access control is not required, or that the user is denied access, it constructs a page correspondingly. Finally, `EntityResource` returns the page back to the `Router`.

Note that the graphs within each component represent greatly simplified program graphs, with only some important control structures preserved. Nodes in these graphs don't represent individual statements, but rather blocks of statements that can be grouped together logically. The *italic label* beside each node explains the main functionality of that node. Edges still represent control flow. Thick edges represent transfer of control (message passing and return) between components.

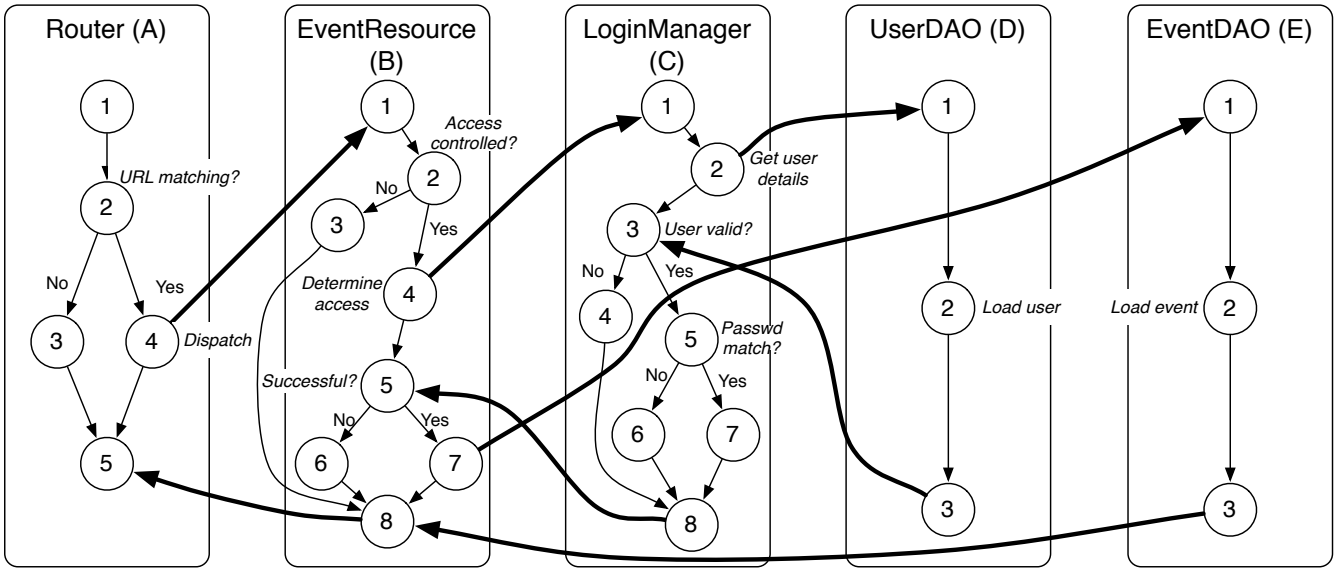

Figure 1: The interactions of some components of the *Who's Coming?* system that handle event presentation.

(Continued overleaf)

Question 4 ..... **10 marks**

In Lecture 6 we introduced the concepts of MM-paths and MM-path graphs. They can help us identify test cases in integration testing. We will develop MM-paths and the MM-path graph for the components depicted in Figure 1 in this question.

(a) (6 marks) List **all** MM-paths for **all** the 5 components above. You can use letters A, B, C, D and E to represent the 5 components in Figure 1. Recall that an MM-path starts from a *source* node and ends at a *sink* node, with no intervening sink nodes.

For example, the MM-path for component D (`UserDAO`) is $MEP(C, 1) = (1, 2, 3)$. The MM-path for component E (`EventDAO`) is $MEP(E, 1) = (1, 2, 3)$.

(b) (4 marks) Recall that an MM-path graph is a directed graph whose nodes are MM-paths and edges are transfer of control (message passing & return) between components. Based on the above MM-paths you developed in part (a), do the following:

(1) Draw the MM-path graph for the components.

(2) Calculate the MM-path complexity for the graph you just drew.

Question 5 ..... **10 marks**

Finite state machines can be very useful in testing by capturing system functionality in terms of states, events, transitions, actions and guards.

Suppose we want to implement a Web service that allows users to create invitations programmatically. A user will specify the user name, password and the details of the invitation to be created. The system creates the invitation upon successfully validating the provided information. A *timeout* function will be implemented. It puts a limit on how long the service can wait to receive user inputs. When the time is up, the system will go back to the state where it is ready to accept another request.

(a) (5 marks) Draw a system-level finite state machine depicting this scenario. If you are thinking of input validation, *do not* break it down into validation of different inputs now. Since the system is a long-running system, we will assume that there is no final state. That is, after an invitation is successfully created, the system goes back to the state where it is ready to create another invitation.

(b) (5 marks) When creating an invitation, two inputs are expected: an **event** and a **recipient**. Draw a more detailed finite state machine for this Web service that takes into account input validation and creating/saving the invitation. Assume that the Web service processes invitation creation in the following steps.

1. Verify authentication through user name and password.
   (a) If authentication fails, return a `Not authenticated` error message.
2. Verify that the event exists and that the user can create invitations for the given event (is the creator).
   (a) If the conditions do not hold, return a `Bad request` error message, or
   (b) If the time-out has been reached, return to the previous state.
3. Verify that the recipient is valid.
   (a) If the recipient does not exist, is the owner of the event or has already been invited, return a `Bad request` error message.
4. Create the invitation, send back an acknowledgement and return to the initial state.
   (a) If an error occurs, return an `Internal server error` error message and go back to the initial state.

Part C. Software Metrics

Question 6 ..... **10 marks**

At the end of Lecture 7 we introduced Weyuker's 9 properties to evaluate software metrics. Some of the properties (for example, properties 1, 3, 4 and 8) are quite simple and intuitive. However, some other properties are a bit more involving and need further analysis.

Weyuker's property 5 states that the complexity of a program fragment should be less than that of the whole program. More formally, $\forall P, Q : Program \bullet M(P) \leq M(P+Q) \wedge M(Q) \neq M(P+Q)$, where M represents a given metric.

Suppose that we want to define a new graph-based OO metric for classes based on the calling relationship between methods in classes. We call this metric *Graph Entropy*, or *Ety*. This metric is calculated indirectly from the call graph of the software being measured. Specifically, *Ety* is defined over a directed multigraph $G = (V, E)$ (A multigraph is just a graph where there can be more than one edge between a pair of nodes). In G, the set of nodes V represents the classes in the system; and for any two nodes V_1 and V_2 (that represent classes `C1` and `C2`, respectively), an edge e is created from V_1 to V_2 if a method in C_1 calls a method in C_2.

For example, class `C1` defines methods `foo1()` and `foo2()`, and class `C2` defines methods `bar1()` and `bar2()`. Suppose that the calling relation between the 2 classes is `C1.foo1()` invokes `C2.bar1()`, and `C1.foo2()` invokes `C2.bar2()`. Then, two edges will be created, both of which go from V_1 to V_2.

The *degree* of a node v in G is the sum of its in-degree and out-degree. Denote with $D_G = \{d_1, d_2, \ldots\}$ the set of *unique* degree values of G. We can calculate the probability of each $d_i \in D_G$ by taking the ratio of its number of occurrences and the total occurrences of degrees of all nodes. We denote the probability of d_i by $p(d_i)$. Given the above definitions, Graph Entropy is defined as follows (assuming that all nodes are connected to some other node):

$$Ety(G) = \sum_{i}^{\#D_G} p(d_i) * \log_2 \frac{1}{p(d_i)} \tag{1}$$

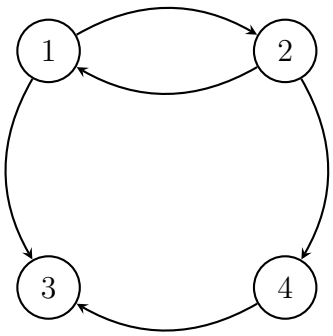

Figure 2: An example graph aggregating the calling relationship between classes.

For example, for the graph shown above, the degrees of the nodes are given in Table 2. From the table we can see that $D_G = \{2, 3\}$, and each of the unique degrees occurs 2 times. The degree probabilities are therefore: $p(2) = p(3) = \frac{2}{4} = 0.5$.

Table 2: Degrees of the four nodes in Figure 2.

Node	**In-degree**	**Out-degree**	**Degree**	**No. occurrences**	**Degree probability**
1	1	2	3	2	0.5
2	1	2	3		
3	2	0	2	2	0.5
4	1	1	2		

Thus, Graph Entropy of the above graph is then $Ety(G) = 0.5 * \log_2 \frac{1}{0.5} + 0.5 * \log_2 \frac{1}{0.5} = 1$.
For Weyuker's property 5 and the metric Ety, do the following:

- State whether the property holds or not.
- Prove your claim (informally).

For clarity reasons, we will make the following assumptions.

- Programs P and Q are both classes in the object-oriented sense.
- Composition is defined by method invocation from one class to another class.

End of the paper