Total Marks: 50 Marks

Answer all questions in the space provided here.

Part A. Unit Testing

Question 1. In Lecture 3 we introduced the Equivalence Class Testing (ECT) technique. In ECT, for each input variable, its input domain is *partitioned* into a number of subsets (equivalence classes) according to an equivalence relation. These subsets are mutually disjoint and their union is the input domain. We then select one test case from each equivalence class to reduce testing redundancy.

For a specification with n number of variables over some numerical domains, let's denote these variables x_i, where $1 \leq i \leq n$. Let C_{x_i} denote the valid equivalence classes for variable x_i (*i.e.,* C_{x_i} is a set of equivalence classes). Remember that the notation for the cardinality of a set S (the number of elements in S) is $\#S$. For simplicity, we will assume that there are no gaps between equivalence classes for each of the variables. In other words, there are only 2 invalid value intervals outside of the equivalence classes (one below the minimum, and one above the maximum).

Example: the minimum number of test cases for Weak Normal ECT is $max(\#C_{x_i})$, for x_i ranging over all variables. That is, the maximum number of equivalence classes for all variables.

Given the above notation and example, do the following: **(Total: 5 marks)**

(1) Work out a formula for the minimum number of test cases for **Strong Robust** ECT.

(2) Briefly explain why.

Question 2. In Lecture 3 we gave an example of the **NextDate** function. Given a day, a month & a year, the **NextDate** function determine the date of the following day. Suppose we need to develop a similar function, **YesterDate**, that, given a day, a month and a year, returns the date of the day before. **(Total: 10 marks)**

(a) Come up with equivalence classes for the three variables `day`, `month` and `year`. **(5 marks)**

(b) Based on your equivalence classes, develop a decision table for the **YesterDate** function. **(5 marks)**

Question 3. Merge sort is a fast and stable sorting algorithm with a time complexity $O(n \log n)$. A key part of Merge sort is a `merge` function, which takes two sorted lists and merges them into a single sorted list. One way of implementing the `merge` function can be seen in the pseudocode below. **(Total: 10 marks)**

Algorithm 1 Merge function for the Merge sort algorithm

Input: *left*: *List* {a sorted list of items}
Input: *right*: *List* {a sorted list of items}
Output: *result*: *List* {a sorted list as a result of merging the inputs *left* and *right*}

```
1:  let result: List
2:  while length(left) > 0 ∨ length(right) > 0 do
3:    if length(left) > 0 ∧ length(right) > 0 then
4:      if first(left) ≤ first(right) then
5:        result = append(first(left), result)
6:        left = rest(left)
7:      else
8:        result = append(first(right), result)
9:        right = rest(right)
10:     end if
11:   else if length(left) > 0 then
12:     result = append(first(left), result)
13:     left = rest(left)
14:   else if length(right) > 0 then
15:     result = append(first(right), result)
16:     right = rest(right)
17:   end if
18: end while
19: return result
```

Note that *length*, *first*, *rest* and *append* are functions defined elsewhere.

(Continued overleaf)

(a) Draw the program graph for the above function. **(5 marks)**
Hint: Statement 7 (the **else** statement) need not be part of the graph.

(b) Recall that McCabe's essential complexity measures how *unstructured* the logic of a program is, using the program graph as a representation. It does so by (1) iteratively condensing the program graph by replacing each *structured programming constructs* with an individual node, and (2) calculating the Cyclomatic complexity of the final condensed graph. **(5 marks)**

(1) Draw the final condensed graph for the program graph you came up with in part (a) above.

(2) Calculate the Cyclomatic complexity of the graph you drew.

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 `Person`s, `Event`s and `Invitation`s. 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 interactions of some components of the system that handles event creation 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, **CreateEventResource** is responsible for processing requests to create events. It gets user inputs, uses them to create the event object by invoking **EntityFactory**, and then saves the event by invoking **EntitySaver**. EntitySaver performs some validation and finally invokes **EventDAO** to actually save the event object in the database.

Note that the graphs within each component represent grossly simplified program graphs, with 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 indicates the main functionality of that node. Edges still represent control flow. Thick edges represent transfer of control (message passing and return) between components.

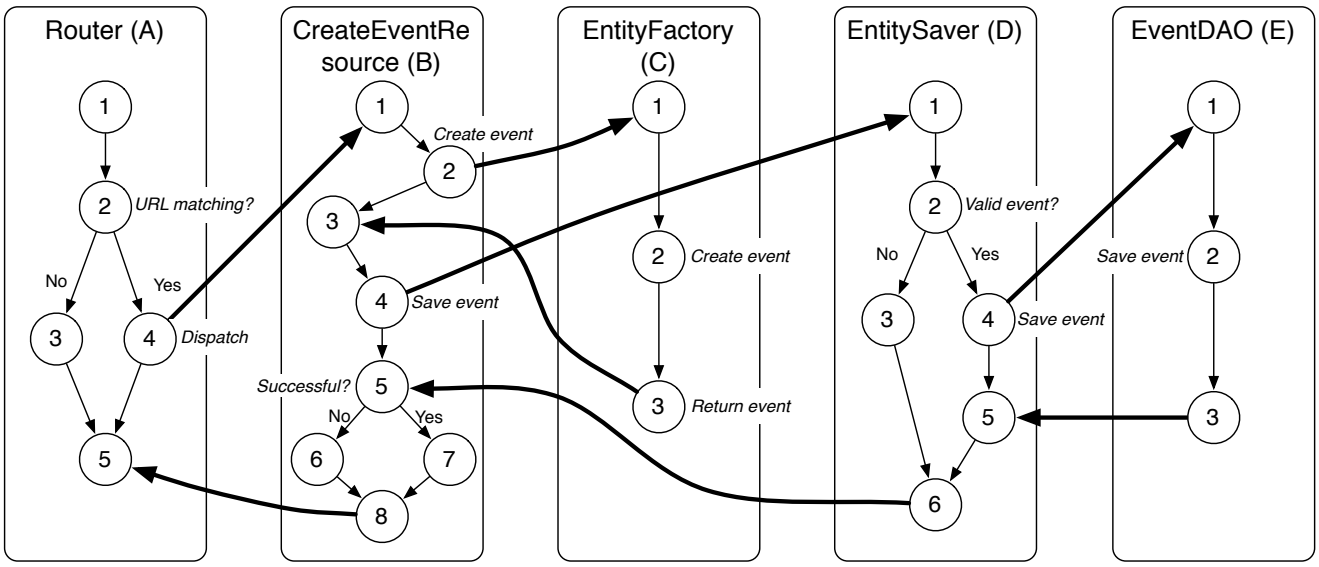

Figure 1: The interactions of some components of the *Who's Coming?* system that handle event creation.

(Continued overleaf)

Question 1. 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. **(Total: 10 marks)**

(a) 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. **Note:** for simplicity reasons, do not treat node 4 in component CreateEventResource as a source node. **(5 marks)**

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

(b) 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: **(5 marks)**

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

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

Question 2. Finite state machines can be very useful in system testing by capturing system functionality in terms of states, events, transitions, actions and guards. Suppose we want to extend the functionality for creating events depicted in Figure 1 to add the ability for the user to cancel the operation. A simple system-level finite state machine specifying this extended function can be seen below in Figure 2. **(Total: 10 marks)**

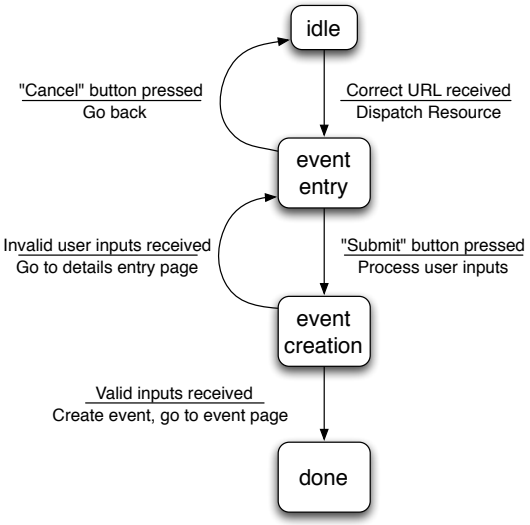

Figure 2: The finite state machine that specifies the extended event creation function.

For simplicity reasons, suppose that for an event to be created, it needs a *name*, a *location* and a *time* attribute (let's ignore the *description* attribute in the course project specification). Based on the above finite state machine, do the following.

(1) In Lecture 8 we introduced the 5 kinds of fundamental constructs of requirements specifications: **data**, **actions**, **devices** (ports), **events** and **threads**. For **data**, **actions**, **devices** and **events** (but not for **threads**), identify one example of each of them. **(4 marks)**

(2) Recall that an atomic system function is an observable action at the system level in terms of port input events and output events. Identify **3** atomic system functions together with their port input and output events. An example is given below. **(6 marks)**

Table 1: Event creation port input events

Event	**Description**
ie0	Correct URL received by the router

Table 2: Event creation port output events

Event	**Description**
oe0	Event details input screen displayed

Table 3: Atomic system functions (ASFs) for the event creation functionality.

ASF	**Input events**	**Output events**
ASF0: Routing of a HTTP request	ie0	oe0

Part C. Software Metrics

Question 1. 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 6 states that the complexity of the composition of two programs P and R may not be the same as the composition of programs Q and R, even though P and Q have the same complexity. Formally, $\exists\, P, Q, R\colon \mathit{Program} \bullet M(P) = M(Q) \land M(P + R) \neq M(Q + R)$, where M represents a given metric.

Weighted Methods per Class (WMC) is an object-oriented metric defined for *classes*. It calculates the complexity of a given class by summing over the complexity values of all methods defined in that class. *I.e.,* for any given class C, $WMC(C) = \sum_{i=1}^{n} C_{m_i}$, where i ranges over all methods m_1 to m_n, and C_{m_i} represents the complexity value of the method m_i.

For Weyuker's property 6 and the metric WMC, do the following: **(Total: 5 marks)**

(1) State whether the property holds or not.

(2) Prove your claim (informally).

For clarity reasons, we will make the following assumptions.

- Programs P, Q and R are all classes in the object-oriented sense.
- Program composition here (the + operator) represents class inheritance. Hence, $P+R$ represents a class obtained from R by extending it with P as a super class (*i.e.,* as the result of the composition, R becomes a subclass of P).
- Programs (classes) P, Q and R are not connected in any way before being composed.
- For any given method, its complexity is 1. In other words, WMC represents the total number of defined methods in the class.
- Finally, inherited methods that are not overridden are **not** counted in WMC.