DIT636 / DAT560 - Practice Examination

There are a total of 14 questions on the practice exam (**there will be fewer on the real exam - we gave you some extra questions to study with**). On all essay questions, you will receive points based on the quality of the answer, not the quantity.

Question 1 (Warm Up) - 10 Points

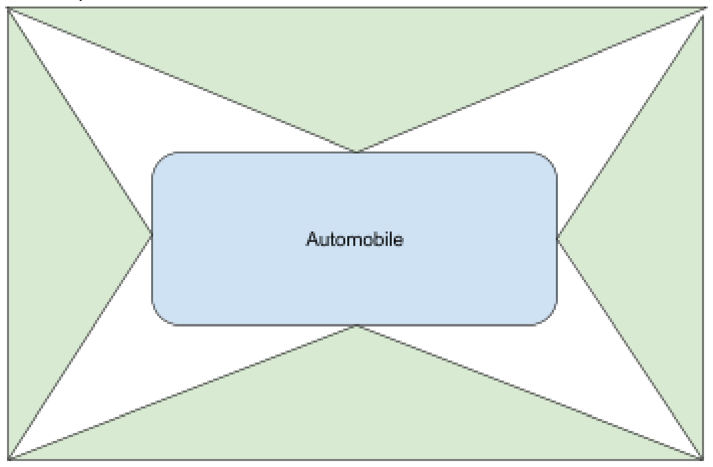
Multiple solutions may apply. Select all that are applicable. True/False worth 1 point each, multiple choice worth 2 points each.

1. For the expression `(a || !c) || (a && b)`, the test suite **(a, b, c) = {(T, F, T), (F, T, T), (T, T, T), (F, F, F)}** provides:
    a. Path Coverage
    b. Decision Coverage
    c. Basic Condition Coverage
    d. Compound Condition Coverage
2. During exploratory testing, the couch potato tour recommends running the system for a long period of time (e.g., overnight) to see if there are failures that emerge over time.
    a. True
    b. False
3. A liveness property is a property that should hold over a path of unknown length.
    a. True
    b. False
4. Mutation is considered useful because a sequence of small code changes can approximately model a larger fault in the program.
    a. True
    b. False
5. You have designed the software for an ATM to meet the following requirement: "If the amount of remaining money in the machine cannot be calculated, then display an error screen and prevent users from performing any additional interactions until the functionality is restored." Which type of property is this?
    a. Correctness
    b. Reliability
    c. Robustness
6. An invalid mutant is one that is detected by many test cases.
    a. True
    b. False
7. You are designing a pedestrian detection system for a vehicle. Which **one** of the following quality attributes would be of most importance to you?
    a. Reliability
    b. Performance
    c. Scalability

    Briefly (1-2 sentences), explain why this is the most important.

Question 2 (Quality Scenarios) - 10 Points

Consider a camera-based object detection system in an automobile. This system uses a set of four cameras to capture the area around a vehicle (the green shaded areas in the image below):



This system offers the following functionality:

- Detects lanes on the road, warning the driver if the car crosses any lines that should not be crossed.
- Detects objects in the roadway (e.g., pedestrians or obstacles) that should not be collided with, and warns the driver if they are in the path of the vehicle.
- Warns the driver if there are conditions where the system cannot make accurate predictions. For example, if weather conditions cause a sharp reduction in image quality or if the camera has been covered.

Create one **reliability** and one **availability** scenario for this system, with a Description, System State, Environment State, External Stimulus, Required System Response, and Response Measure for each.

Question 3 (Testing Concepts) - 8 Points

- Define system (integration) testing and exploratory testing.
- Explain how systems are tested at each stage.
- Explain how the two stages differ.
- Explain types of faults that may be more likely to be exposed by each of the two stages.

Question 4 (Test Design) - 12 Points

Recall the object discussion system discussed in Question 2.

Internally, the following method is invoked each time that the sensor data is refreshed:

```
public String[] analyzeObstacles (List<Float> cameraData)
```

The input parameter `cameraData` represents a list of zero or more potential obstacles detected by that camera. Each Float represents the distance that a detected obstacle is from the camera.

For example, if:
`System.out.println(cameraData.get(0));`
prints “1.45” to the screen, this indicates that the camera has detected an object (`cameraData.get(0)`) that is 1.45 meters from that camera.

This method should analyze the camera data and return a String array, where each item is a warning for the driver. The warnings should include:

- Any detected obstacle that is within 2 meters of the camera.
- Any situation where an obstruction may be blocking the camera (indicated by a detected object that is listed as 0.00 meters from the camera).

Perform functional test design for this method.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative input values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

You do not need to create test specifications or concrete test cases. For invalid input, **do not** just write “invalid” - be specific. If you wish to make any additional assumptions about the functionality of this method, state them in your answer.

Question 5 (Test Design)

Consider a personnel management program that offers an API where, among other functions, a user can apply for vacation time:

```
public boolean applyForVacation (String userID, String startingDate,
String endingDate)
```

A user ID is a string in the format “firstname.lastname”, e.g., “gregory.gay”.
The two dates are strings in the format “YYYY-DD-MM”.

The function returns TRUE if the user was able to successfully apply for the vacation time. It returns FALSE if not. An exception can also be thrown if there is an error.

This function connects to a user database. Each user has the following relevant items stored in their database entry:

- User ID
- Quantity of remaining vacation days for the user
- An array containing already-scheduled vacation dates (as starting and ending date pairs)
- An array containing dates where vacation cannot be applied for (e.g., important meetings).

Perform functional test design for this function.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative input values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

You do not need to create test specifications or concrete test cases. For invalid input, **do not** just write “invalid” - be specific. If you wish to make any additional assumptions about the functionality of this method, state them in your answer.

Question 6 (Exploratory Testing) - 8 Points

Exploratory testing typically is guided by “tours”. Each tour describes a different way of thinking about the system-under-test and prescribes how the tester should act when they explore the functionality of the system.

1. Describe one of the tours that we discussed in class **other than the supermodel tour**.
2. Consider a web-based discussion forum. This forum offers the following functionality:
   - Users can register for an account, using their e-mail address. They can choose a display name (must be unique) and a password for their account.
   - Once registered, users can log into their account.
   - Some users, called “administrators”, have additional privileges.
   - Administrators can create, edit the name and description of, and delete “boards”.
   - A “board” is an area where users can post topics related to the subject of that board.
   - Users can also reply to topics.
   - Users can edit their own posts in topics.
   - Administrators can also edit or delete posts in topics.

   Describe three distinct sequences of interactions with one or more functions of this system you would explore during exploratory testing of this system, based on the tour you described above. Explain the interactions you would take when executing that sequence, and why those actions fulfill the goals of that tour. These sequences should be different from each other (i.e., limited overlap).

Question 7 (Unit Testing) - 9 Points

Consider the obstacle detection method that you developed test specifications for in Question 4:

```
public String[] analyzeObstacles (List<Float> cameraData)
```

Based on your test specifications, write three JUnit-format test cases:

1. Create one test case that checks a normal usage of the method, with at least one detected obstacle.
2. Create one test case that checks a normal usage of the method, with a potential obstruction.
3. Create one test case reflecting an error-handling scenario (an exception is thrown).

Question 8 (Structural Testing)

For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage. For each test, note which lines are covered.
c. Develop test input that will provide branch coverage. For each test, note which branches are covered. You may reuse input from the previous problem.
d. Develop test input that will provide path coverage. For each test, note which paths are covered. You may reuse input from the previous problem.
e. Modify the program to introduce a fault so that you can demonstrate that even achieving path coverage will not guarantee that we will reveal all faults. Please explain how this fault is missed by your test cases.

```
1.   int findMax(int a, int b, int c) {
2.        int temp;
3.        if (a > b)
4.           temp=a;
5.        else
6.           temp=b;
7.        if (c > temp)
8.           temp = c;
9.        return temp;
10.   }
```

(Just including test input is sufficient - you do not need to write full JUnit cases)

Question 9 (Structural Testing) - 16 Points

This function takes an array and string. It will then remove the letters in the string from the array, one at a time, and return the array. If a letter is in the array multiple times, it will only be removed the number of times that it appears in the string (e.g., if “b” is in the array twice, and the string is “big”, then “b” will only be removed one time).

```
1.  public String[] removeLetters(String[] letters, String word) {
2.      int lettersRemoved = 0;
3.      while(word.length() != 0){
4.          for(int i = 0; i < letters.length; i++){
5.              if(letters[i].equals(word.substring(0, 1))){
6.                  letters[i] = "";
7.                  lettersRemoved++;
8.                  break;
9.              }
10.         }
11.         word = word.substring(1);
12.     }
13.     int idx = 0;
14.     String[] output = new String[letters.length - lettersRemoved];
15.     for(int i = 0; i < letters.length; i++){
16.         if(!letters[i].equals("")){
17.             output[idx] = letters[i];
18.             idx++;
19.         }
20.     }
21.     return output;
22. }
```

For example:

- removeLetters(["s", "t", "r", "i", "n", "g", "w"], "string") ➞ ["w"]
- removeLetters(["b", "b", "l", "l", "g", "n", "o", "a", "w"], "balloon") ➞ ["b", "g", "w"]
- removeLetters(["d", "b", "t", "e", "a", "i"], "edabit") ➞ [ ]

1. Draw the control-flow graph for this function. You may refer to line numbers instead of writing the full code.
2. Identify test input that will provide statement and branch coverage. You do not need to create full unit tests, just supply input for the function.

   For each input, list the line numbers of the statements covered as well as the specific branches covered (use the line number and T/F, i.e., “3-T” for the true branch of line 3). Do not use the test input from the examples above. Come up with your own input.

Question 10 (Data Flow Testing) - 12 Points

Using the same code from Question 9:

1. Identify the def-use pairs for all variables.
2. Identify test input that achieves all def-use pairs coverage.

Note: You may treat arrays as a single variable for purposes of defining DU pairs. This means that a definition to **`letters[i]`** or to array **`letters`** are both definitions of the same variable, and references to **`letters[i]`** or **`letters.length`** are both uses of the same variable.

Question 11 (Data Flow Testing)

The following function returns true if you can partition an array into one element and the rest, such that this element is equal to the product of all other elements excluding itself.
For example:

- `canPartition([2, 8, 4, 1])` returns true (8 = 2 * 4 * 1)
- `canPartition([-1, -10, 1, -2, 20])` returns false.
- `canPartition([-1, -20, 5, -1, -2, 2])` returns true (-20 = -1 * 5 * -1 * -2 * 2)

```
1. public static boolean canPartition(int[] arr) {
2.       Arrays.sort(arr);
3.       int product = 1;
4.       if ((Math.abs(arr[0]) >= arr[arr.length-1]) || arr[0] == 0) {
5.           for (int i = 1; i < arr.length; i++){
6.               product *= arr[i];
7.           }
8.           return arr[0] == product;
9.       } else{
10.         for (int i = 0; i < arr.length-1; i++){
11.             product *= arr[i];
12.         }
13.         return arr[arr.length-1] == product;
14.      }
15.  }
```

1. Identify the def-use pairs for all variables.
2. Identify test input that achieves all def-use pairs coverage.

Note: You may treat arrays as a single variable for purposes of defining DU pairs. This means that a definition to `arr[0]` or to array `arr` are both definitions of the same variable, and references to `arr[0]` or `arr.length` are both uses of the same variable.

Question 12 (Mutation Testing) - 15 Points

This function takes in an integer, and inserts duplicate digits on both sides of all digits which appear in a group of one.

```
1. public static long lonelyNumbers(int n) {
2.     String s = " " + n + " ";
3.     long a = 0;
4.     StringBuilder sb = new StringBuilder();
5.     for(int i = 1; i < s.length() - 1; i++){
6.         if(s.charAt(i) == s.charAt(i - 1) || s.charAt(i) ==
               s.charAt(i + 1)){
7.             sb.append(s.charAt(i));
8.         } else
9.             sb.append("" + s.charAt(i) + s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13.   }
```

For example:

- `lonelyNumbers(4666) ➞ 444666`
- `lonelyNumbers(544) ➞ 55544`
- `lonelyNumbers(123) ➞ 111222333`
- `lonelyNumbers(33) ➞ 33`

Answer the following three questions for **each** of the following mutation operators:

- Relational operator replacement (ror)
- Arithmetic operator replacement (aor) (including short-cut operators)
- Constant for constant replacement (crp)

1. Identify all lines that can be mutated using that operator.
2. Choose **one** line that can be mutated by that operator and create **one** non-equivalent mutant for that line.
3. For that mutant, identify test input that would detect the mutant. Show how the output (return value of the method) differs from that of the original program.

Question 13 (Finite State Verification)

Temporal Operators: A quick reference list. p is a Boolean predicate or atomic variable.

- G p: p holds globally at every state on the path from now until the end
- F p: p holds at some future state on the path (but not all future states)
- X p: p holds at the next state on the path
- p U q: q holds at some state on the path and p holds at every state before the first state at which q holds.
- A: for all paths reaching out from a state, used in CTL as a modifier for the above properties (AG p)
- E: for one or more paths reaching out from a state (but not all), used in CTL as a modifier for the above properties (EF p)

An LTL example:

- G ((MESSAGE_STATUS = SENT) -> F (MESSAGE_STATUS = RECEIVED))
- It is always true (G), that if the message is sent, then at some point after it is sent (F), the message will be received.
    - More simply: A sent message will always be received eventually.

A CTL example:

- EG ((WEATHER = WIND) -> AF (WEATHER = RAIN))
- There is a potential future where it is a certainty (EG) that - if there is wind - it will always be followed eventually (AF) by rain.
    - More simply: At a certain probability, wind will inevitably lead to eventual rain. (However, that probability is not 100%)

Consider a finite state model of a traffic-light controller similar to the one discussed in the homework, with a pedestrian crossing and a button to request right-of-way to cross the road.

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:
pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

1. Briefly describe a safety-property (nothing “bad” ever happens) for this model and formulate it in CTL.
2. Briefly describe a liveness-property (something “good” eventually happens) for this model and formulate it in LTL.
3. Write a trap-property that can be used to derive a test case using the model-checker to exercise the scenario “pedestrian obtains right-of-way to cross the road after pressing the button”.

   A trap property is when you write a normal property that is expected to hold, then you negate it (saying that the property will NOT be true). The verification framework will then produce a counter-example indicating that the property actually can be met - including a concrete set of input steps that will lead to the property being true.

Question 14 (Finite State Verification)

Consider a simple microwave controller modeled as a finite state machine using the following state variables:

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press (assumes at most one at a time)
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

Formulate the following informal requirements in CTL:

1. The microwave shall never cook when the door is open.
2. The microwave shall cook only as long as there is some remaining cook time.
3. If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.

Formulate the following informal requirements in LTL:

1. It shall never be the case that the microwave can continue cooking indefinitely.
2. The only way to initiate cooking shall be pressing the start button when the door is closed and the remaining cook time is not zero.
3. The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.