UECS2354 Software Testing
Lab 07: Designing Tests (Part 2)

There are 5 main approaches to performing black box testing:

1. Equivalence partitioning
2. Boundary value analysis
3. Decision table testing
4. **State transition testing**
5. Use case testing

State transition testing

1. Some systems have certain components or subsystems that can be modelled as a Finite State Machine (FSM). This means that the system can be in a finite (or limited) number of different states, and the transitions from one state to another are determined by a set of rules. A FSM for a system has four basic parts:

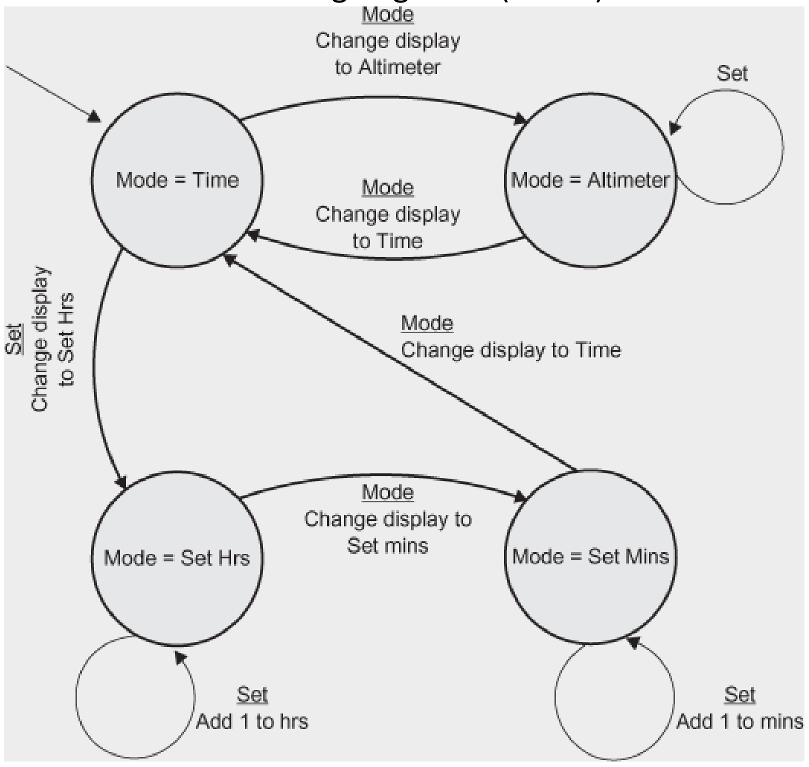
   i. The states possible for that system to be in. A state is a static, stable situation in the system.
   ii. The possible transitions from one state to another (but not all transitions are allowed)
   iii. The events that cause a transition. An event could be an input to the system, or something inside the system that changes
   iv. The actions or outputs that result from a transition. An event can also change the system's internal state without generating an output, or cause both to happen simultaneously.

2. A FSM is often shown as a state transition diagram. The diagram depicts the states that a component or system can assume, and shows the events that cause a transition from one state to another; as well as any actions that may result from this transition.

   A distinguishing characteristic of a FSM is that the output from the machine for a given input event is dependent on the current state, so it is possible for the same input into the system to produce different outputs or results.

   In any given state, one event can cause only one action, but that same event - from a different state - may cause a different action and a different end state.

3. An example of a state transition diagram that represents the operation of a simple digital watch is shown below. The 4 circles represent states which correspond to the mode the watch is in: `time`, `altimeter`, `set mins` and `set hrs`.



The lines with arrows between the circles are transitions between the states, and the labels for the arrows specify the event that causes the transition as well as the output or action of that transition. There are only 2 possible events in this system: `mode` and `set`.

This correspond to the action of depressing the mode and set buttons on the watch. The description below the **mode** or **set** event describes the output or action when the event triggers the transition. For example, change display to altimeter or add 1 to hrs.

As an example, if we are currently in the `time` state, the `mode` event causes transition to the `altimeter` state; and at the same time, the display on the watch changes to altimeter. Alternatively, the `set` event causes a transition to the `set hrs` state; and at the same time, the display on the watch changes to set Hrs.

4. A **state table** is a useful companion to the state transition diagram. It shows the total number of combinations of states and transitions possible in the FSM; and helps in identifying invalid transitions. The state table lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa).

   For the simple digital watch example, the 4 states (`time, altimeter, set Hrs` and `set Mins`) are listed on the left, and the two events (`mode` and `set`) are listed at the top. Each cell then represents a state-event pair. The content of each cell indicates which state the system will move to, when the corresponding event occurs while in the associated state.

   For example, the top middle cell represents the state (with corresponding action) that the system will transition into, if the previous state was mode=time and the event that occurs is mode.

State	**Mode (event)**	**Set (event)**
Mode = Time	Mode = Altimeter / Change Display to Altimeter	Mode = Set Hrs / Change Display to Set Hrs
Mode = Altimeter	Mode = Time / Change Display to Time	Null
Set Hrs	Mode = Set Mins / Change Display to Set Mins	Set Hrs / Add 1 to Hrs
Set Mins	Mode = Time / Change Display to Time	Set Mins / Add 1 to Mins

Some cells may represent erroneous events; events that are not expected to happen in certain states, although they can.

For example, while in the `altimeter` state, the `set` event should not happen. The `null` indicates that no action will take place if this event should occur; and from the original state transition diagram, we can see that the system will still continue to stay in the `altimeter` state.

Go through the state table to verify that it represents correctly all the states and transitions shown in the state transition diagram.

UECS2354 Software Testing
Lab 07: Designing Tests (Part 2)

5. Refer to the class `FSMWatch`, although we can represent the states and events with Strings, the String data type can also represent a vast number of possible values in addition to this. To reflect more accurately the concept that the number of different states and events are limited to 4 and 2 respectively, it would be better to define **special data types** which have only 4 and 2 possible values respectively.

   The **`enum`** data type serves this purpose. We declare two `enum` types (`WatchStates`, `WatchEvents`) that are used to represent the states and events possible in this FSM. We can declare variables from the `enum` data types or use them as method parameters.

```
package my.edu.utar;

enum WatchStates {setHrs, setMins, Altimeter, Time }

enum WatchEvents { pressMode, pressSet }

public class FSMWatch {

   private WatchStates currentWatchState = WatchStates.Time;
   private String watchDisplay = "Showing Time";
   private int currentHrs = 0;
   private int currentMins = 0;

   public FSMWatch(WatchStates currentWatchState, String watchDisplay,
      int currentHrs,int currentMins) {

      this.currentWatchState = currentWatchState;
      this.watchDisplay = watchDisplay;
      this.currentHrs = currentHrs;
      this.currentMins = currentMins;
    }

   public WatchStates getCurrentWatchState() {
      return currentWatchState;
   }

   public void setCurrentWatchState(WatchStates currentWatchState) {
      this.currentWatchState = currentWatchState;
   }

   public String getWatchDisplay() {
      return watchDisplay;
   }

   public void setWatchDisplay(String watchDisplay) {
      this.watchDisplay = watchDisplay;
   }

   public int getCurrentHrs() {
      return currentHrs;
   }

   public void setCurrentHrs(int currentHrs) {
      this.currentHrs = currentHrs;
   }
```

```
  public int getCurrentMins() {
     return currentMins;
  }

  public void setCurrentMins(int currentMins) {
     this.currentMins = currentMins;
  }

  public void processEvent(WatchEvents eventToDo) {
     if (eventToDo == WatchEvents.pressMode)
     {
        switch (currentWatchState) {
           case Time:
                  watchDisplay = "Showing Altimeter";
                  currentWatchState = WatchStates.Altimeter;
                  break;
           case Altimeter:
                watchDisplay = "Showing Time";
                currentWatchState = WatchStates.Time;
                break;
           case setHrs:
                watchDisplay = "Showing Set Mins";
                currentWatchState = WatchStates.setMins;
                break;
           case setMins:
                watchDisplay = "Showing Time";
                currentWatchState = WatchStates.Time;
                break;
        }
     }
     else if (eventToDo == WatchEvents.pressSet){
        switch (currentWatchState) {
           case Time:
                  watchDisplay = "Showing Set Hrs";
                currentWatchState = WatchStates.setHrs;
                break;
           case Altimeter:
                  break;
           case setHrs:
                  if (++currentHrs > 23) currentHrs = 0;
                  break;
           case setMins:
                  if (++currentMins > 59) currentMins = 0;
                  break;
        }
     }
  }
}
```

6. Notice that the display on the watch is represented as a String variable, although it can be better represented as an `enum` data type since it has only 4 possible values: (Showing Altimeter, Showing Time, Showing Set Mins, Showing Set Hrs). By doing this, we can see the difference between working with String and `enum` type parameters when writing parameterised tests.

7. The class `FSMWatch` contains instance variables that model the internal state of the watch. In addition to `currentWatchState` and `watchDisplay` which represent the current state and the display shown on the watch, we also have `currentHrs` and `currentMins` to store the value for the hours and minutes in the watch. These need to be included since there are some events in some states which cause the value of the hours and minutes to be incremented by 1.

8. The **`processEvent()`** method is the **key method** that implements the state transition diagram functionality. It accepts a parameter that represents an event; and changes the internal state, watch display as well as hours or minutes as necessary.

   Notice that there are 2 main **`if`** statements corresponding to the 2 possible events. Within the blocks for these respective `if` statements, there is a `switch` structure which checks the current state of the watch in order to determine the next state to transition to.

   Notice also that in the processing of the `setHrs` and `setMins` event, we not only increment either `currentHrs` or `currentMins` by 1, we also reset them to 0 if they exceed a certain value.

   For example, if incrementing `currentHrs` or `currentMins` causes them to exceed 23 or 59 respectively, they will be reset back to 0. Although this is not stated explicitly in the state transition diagram, it is the normal expectation for a 24 hour time keeping system, and so it is implemented here.

9. Testing a system modelled as a FSM basically involves ensuring as comprehensive a testing of the state transition diagram as possible. The tests are derived from the state transition diagram or state table and may involve either:
   - A **typical** use scenario of the system, which may cover **some states and transitions**; but not all.
   - A **more** comprehensive test that ensures **all states** in the FSM are covered. Some transitions may not be tested; this will depend on the state transition diagram being modelled.
   - The **most** comprehensive test ensures **all transitions** are exercised. This helps to check whether erroneous events (events that are not expected in certain states) do not result in invalid transitions. Trying out all transitions will also ensure that all states are covered.

10. In the class `FSMWatchTest`, there are 4 different parameterised test methods: `testTimeState()`, `testAltimeterState()`, `testSetHrsState()` and `testSetMinsState()`.
    Each representing the 4 different states (`time`, `altimeter`, `set hrs` and `set mins`). Notice that all 4 tests start by instantiating an object from the class `FSMWatch` with its internal state variables set to the correct values for that particular state.

    Both `testTimeState()` and `testAltimeterState()` have two input parameter combinations to test for the two possible transitions out of these states caused by either the `pressMode` or `pressSet` events.
    We initialize the instance variables of `currentHrs` and `currentMins` to 0 in both tests, since the transitions in these two states do not involve any change to these instance variables and thus we do not need to check them with any `assertsXXX` method calls. However, there is a change to these instance variables in the states `set hrs` and `set mins`, and so the test methods `testSetHrsState()` and `testSetMinsState()` will accept different values that it will pass to these variables when it instantiates the `FSMWatch` object for testing.

```
package my.edu.utar;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
import my.edu.utar.FSMWatch;
import my.edu.utar.WatchEvents;
import my.edu.utar.WatchStates;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;

@RunWith(JUnitParamsRunner.class)
public class FSMWatchTest {

   @Test
   @Parameters({"pressSet,setHrs,Showing Set Hrs",
   "pressMode,Altimeter,Showing Altimeter"})
   public void testTimeState(WatchEvents eventToDo, WatchStates
   expectedState, String expectedDisplay) {

      FSMWatch fw = new FSMWatch(WatchStates.Time, "Showing Time", 0, 0);
      fw.processEvent(eventToDo);
      assertEquals(expectedState, fw.getCurrentWatchState());
      assertEquals(expectedDisplay, fw.getWatchDisplay());
   }

   @Test
   @Parameters({"pressSet,Altimeter,Showing Altimeter",
   "pressMode,Time,Showing Time"})
   public void testAltimeterState(WatchEvents eventToDo, WatchStates
   expectedState, String expectedDisplay) {

      FSMWatch fw = new FSMWatch(WatchStates.Altimeter, "Showing
   Altimeter", 0, 0);
      fw.processEvent(eventToDo);
      assertEquals(expectedState, fw.getCurrentWatchState());
      assertEquals(expectedDisplay, fw.getWatchDisplay());
   }

   @Test
   @Parameters({"pressSet,setHrs,Showing Set Hrs,5,10,6,10",
      "pressSet,setHrs,Showing Set Hrs,23,10,0,10",
      "pressMode,setMins,Showing Set Mins,0,5,0,5"})
   public void testSetHrsState(WatchEvents eventToDo, WatchStates
   expectedState, String expectedDisplay,
         int currentHour, int currentMinute, int expectedHour, int
   expectedMinute) {

      FSMWatch fw = new FSMWatch(WatchStates.setHrs, "Showing Set Hrs",
   currentHour, currentMinute);
      fw.processEvent(eventToDo);
      assertEquals(expectedState, fw.getCurrentWatchState());
      assertEquals(expectedDisplay, fw.getWatchDisplay());
      assertEquals(expectedHour, fw.getCurrentHrs());
      assertEquals(expectedMinute, fw.getCurrentMins());
   }
```

```
    @Test
    @Parameters({"pressSet,setMins,Showing Set Mins,5,10,5,11",
       "pressSet,setMins,Showing Set Mins,5,59,5,00",
       "pressMode,Time,Showing Time,0,5,0,5"})

    public void testSetMinsState(WatchEvents eventToDo, WatchStates
    expectedState, String expectedDisplay,
          int currentHour, int currentMinute, int expectedHour, int
    expectedMinute) {
       FSMWatch fw = new FSMWatch(WatchStates.setMins, "Showing Set Mins",
    currentHour, currentMinute);
       fw.processEvent(eventToDo);
       assertEquals(expectedState, fw.getCurrentWatchState());
       assertEquals(expectedDisplay, fw.getWatchDisplay());
       assertEquals(expectedHour, fw.getCurrentHrs());
       assertEquals(expectedMinute, fw.getCurrentMins());
    }
}
```

11. For the `testSetHrsState()`, there are 2 possible conditions to check when the `setHrs` event is triggered. First, if the `currentHrs` instance variable is less than 23, it is incremented by one; and secondly, if the `currentHrs` instance variable is 23, than it is reset to 0. This corresponds to the last 4 values in the first two input parameter combinations (`5,10,6,10` and `23,10,0,10`), which are associated with the occurrence of the `setHrs` event.

    Similar comments apply as well to `testSetMinsState()`. Running all 4 tests will ensure that all states and all transitions are exercised; so this is the most comprehensive test available.

12. Study the 4 parameterised tests carefully, we will realise that they only **differ in terms of the values they pass** to the `FSMWatch` object that they instantiate. The additional 2 assert statements in `testSetHrsState()` and `testSetMinsState()`:

    ```
    assertEquals(expectedHour, fw.getCurrentHrs());
    assertEquals(expectedMinute, fw.getCurrentMins());
    ```

    can also be included in `testAltimeterState()` and `testTimeState()` since they are redundant here and do not make a difference. Based on this observation, we can collapse all 4 parameterised tests into a single parameterised test; with the values that are used to instantiate the `FSMWatch` object being passed in as parameters as well.

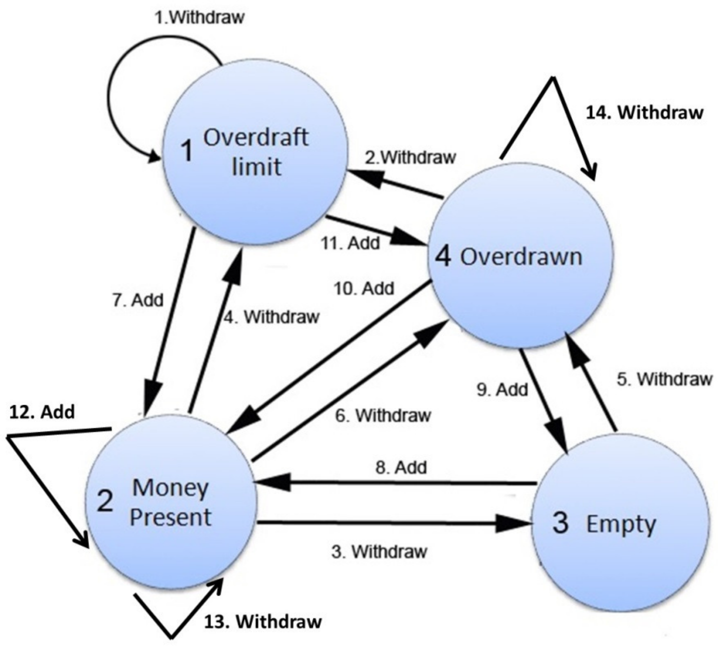
    Exercise
    Create a new test consisting of this single parameterised test.

13. The state transition diagram below represents 4 different states of a customer's bank account that correspond to the current balance of the account.

    There are 2 events: **`add`** and **`withdraw`**, which cause transitions between these states. Each of these events is specified with an associated amount of money, for example, withdraw 100 or add 200, which will change the current balance accordingly.

    Thus, every transition between 2 different states (or within the same state) will be accompanied by a change in the current balance, with the exception of transition 1. The bank offers an overdraft facility, which allows the customer to withdraw more than what is currently in the balance up to the specified overdraft limit. For example, if the specified limit is -500, and the customer's current balance is 200, then the maximum amount in a single withdrawal is 700 (transition 4). The relationships between the account balance, states and transitions are as follows:

    - When the balance is exactly **0**, the account is in the **`Empty`** state.
    - When the balance is **positive** (+ve), the account is in the **`Present`** state.
    - When the balance is **negative** (-ve) but **less than the specified limit**, the account is in the **`Overdrawn`** state. This is the situation where the customer has withdrawn more than what she has in her current balance, but has not reached the specified limit yet.
    - When the balance is **negative** and **exactly at the specified limit**, the account is in the **`Overdraft`** state
    - Any attempt to withdraw an amount beyond the specified limit from either the `Overdraft`, `Present` or `Overdrawn` state will cause the balance to be set to the specified limit along with a transition to the `Overdraft` state (transitions 1, 2 and 4).
    - Any event that occurs with a given amount of money that can cause a transition that is not shown in the state transition diagram will result in an **Exception** being thrown. For example, there is no transition from the `Empty` state to the `Overdraft` state. Hence, a withdrawal event with an amount of money that might cause such a transition results in an `Exception` being thrown.



14. Refer to the class `FSMBankAccount`, we have 2 `enum` data types: `AccountStates` and `AccountEvents` to represent the finite values for the 4 different states of a customer's bank account and the 2 events. There are 3 instance variables which represent the overdraft limit, current balance and the current state of the bank account; all of which can be set via the constructor of the class.

    The key method that implements the functionality of the state transition diagram is **`processEvent()`** which accepts 2 parameters representing the event that occurs and the amount of money associated with it.

    Notice that there are 2 main **`if`** statements corresponding to the 2 possible events. Within the blocks for these respective `if` statements, there is a **`switch`** structure which checks the current state of the bank account and the balance of the account in order to determine the next state to transition to.

```
package my.edu.utar;

enum AccountStates { present, overdraft, overdrawn, empty}

enum AccountEvents { addMoney, withdrawMoney }

public class FSMBankAccount {

   private int overdraftLimit = -500;
   private int currentBalance = 0;
   private AccountStates currentState = AccountStates.empty;

   public FSMBankAccount(int overdraftLimit, int currentBalance,
   AccountStates currentState) {
      this.overdraftLimit = overdraftLimit;
      this.currentBalance = currentBalance;
      this.currentState = currentState;
   }

   public int getOverdraftLimit() {
      return overdraftLimit;
   }

   public int getCurrentBalance() {
      return currentBalance;
   }

   public AccountStates getCurrentState() {
      return currentState;
   }

   public void processEvent(AccountEvents eventToDo, int amount)
   {
      if (eventToDo == AccountEvents.addMoney) {
         currentBalance += amount;

         switch (currentState) {
            case empty:
            case present:
               currentState = AccountStates.present;
               break;
            case overdraft:
            case overdrawn:
               if (currentBalance > 0)
                  currentState = AccountStates.present;
               else if (currentBalance == 0 && currentState ==
                     AccountStates.overdrawn)
                  currentState = AccountStates.empty;
               else if (currentBalance < 0)
                  currentState = AccountStates.overdrawn;
               else
                  throw new IllegalArgumentException();
               break;
         }
      }
      else if (eventToDo == AccountEvents.withdrawMoney)
      {
         currentBalance -= amount;
         if (currentBalance < overdraftLimit)
            currentBalance = overdraftLimit;

         switch (currentState) {
            case present:
            case overdrawn:
               if (currentBalance == overdraftLimit)
                  currentState = AccountStates.overdraft;
               else if (currentBalance < 0)
                  currentState = AccountStates.overdrawn;
               else if (currentBalance == 0)
                  currentState = AccountStates.empty;
               else
                  currentState = AccountStates.present;
               break;
            case empty:
               if (currentBalance == overdraftLimit)
                  throw new IllegalArgumentException();
               currentState = AccountStates.overdrawn;
               break;
            case overdraft:
               break;
         }
      }
   }
}
```

Exercise

Create **4 different parameterised test methods** to represent the 4 different states available in the bank account. Ensure that the input parameter combinations for your test methods will exercise all states and transitions to achieve the most comprehensive test available. This includes checking that an exception is correctly thrown if the erroneous event mentioned previously occurs (for example, an event that causes transition from the `Empty` state to the `Overdraft` state).

When you are done, create another test class that collapses the 4 different parameterised tests into a single parameterised test method; in a similar way to that in the previous example.