UECS2354 Software Testing
Lab 07: Designing Tests (Part 1)

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

We will survey the first four approaches, as they can be utilized directly as part of unit testing. The last approach is more suited for higher-level testing such as in system or user acceptance testing.

You should be acquainted with the following concepts in Java:

- throwing and handling `Exceptions`
- concept of inheritance and polymorphic methods
- casting between inherited classes, `instanceof` operator and `ClassCastException`

Equivalence partitioning testing

1. The entire range of possible inputs for the parameters in the method being tested can be divided into equivalence partitions. Each partition contains equivalent inputs, in the sense that the method will behave in the same manner or produce an identical output for all input values within that partition.

2. We need to utilise only **one input value for each unique partition** in a test. If this input value produces the desired output or functionality in the method being tested, we can assume that all other input values within that partition will work.
   Conversely, if one of the input values in a partition does not work, then we assume that none of the other input values in that partition will work.
   This approach reduces the number of tests that we need to write.

3. In addition to the range of valid values, we also need to verify that a method can handle unspecified or invalid values that may cause the method to terminate prematurely or behave in an unexpected manner.
   The range of invalid values can be partitioned in the same way as those for valid values. We will create tests using inputs from these invalid partitions. This is particularly important when the input to the method is coming directly from user input (e.g. data input into a web form or files uploaded by a user to a website).
   Invalid user input is one of the primary causes of failure or security issues in programs!

4. The problem is that requirements for a method (from a low level design document) may not always clearly specify what the method should do when an invalid value is passed to one of its parameters. The convention in that case is to ensure that the code checks for such invalid values and throws an Exception object that reflects the specific problem with that particular invalid value. The `IllegalArgumentException` is conventionally the most widely used one.

5. It is important not to confuse Exceptions that are thrown specifically to respond to invalid parameter values, and Exceptions that are thrown during the normal operation of a method. Consider the case of a method that accepts a String parameter that specifies the name of a file which the method will open. If the String parameter has an invalid value (e.g. `null`), then ideally

we should immediately throw an Exception (such as `IllegalArgumentException`) because there is no way the code can open a file without a name.
On the other hand, even if the String parameter is valid, the process of attempting to open the file may fail due to variety of reasons (e.g. incorrect location specified). In that case, a `FileNotFoundException` will be thrown as well: however here, this Exception reflects a problem with method operation, not with the parameter passed.

6. This is why it is important to check carefully the Exception type that is received back in a test method that expects an Exception. If we pass the tested method an invalid parameter value, then we would expect back the right type of Exception corresponding to that invalid parameter value. If the method accepts the invalid parameter without complaining but subsequently crashes later and throws a different type of Exception, then we would consider the test to have failed.

7. Below are some possible considerations for invalid values for different kinds of parameters that may be present in a method signature:

Types	**Possible invalid values**
numeric values (`int`, `double`, `float`, `long` and `short`	negative values, 0, excessively large values(in the context of that particular application)
Strings	`null`, empty String
`Collection` types (e.g. array, `ArrayList`)	`null`, array with length 0, array with elements that are `null` (for the case when the elements of the array are reference variables themselves)
reference variables of user defined classes	`null`, objects with uninitialized member variables

NOTE:
Methods with many parameters result in a large number of combinations of possible invalid input partitions and correspondingly large number of tests to write. The implemented method must also include code to check its parameters for all these invalid values and throw suitable Exceptions when they are found. This may result in an excessively large number of tests, as well as additional code in these methods that may make it more complicated and difficult to read and understand.

On the other hand, neglecting to check for any invalid values will almost guarantee an unexpected program crash at a later stage of development or testing. The key therefore is find the right balance between no tests at all, and testing for every single possible combination of invalid input values.

Regardless of what invalid input values are tested for, it must be borne in mind that ALL valid equivalence classes MUST be tested. This is the bare minimum for a satisfactory partitioning approach.

8. Refer to the class `PartitionExample`, the `getGrade()` method accepts a single `int` parameter representing a mark for a subject and returns a String reflecting the grade for that corresponding mark. The grades and their corresponding mark ranges are shown in the table below:

Mark range	Grade
0 – 50	F
51 – 60	D
61 – 70	C
71 – 80	B
81 – 100	A

```java
package my.edu.utar;

public class PartitionExample {

   public String getGrade(int mark) {

      if (mark < 0 || mark > 100)
         throw new IllegalArgumentException("Mark out of range");
      if (mark < 51)
         return "F";
      else if (mark < 61)
         return "D";
      else if (mark < 71)
         return "C";
      else if (mark < 81)
         return "B";
      else
         return "A";
   }

   public double calculateTax(int salary) {

      if (salary < 0)
         throw new IllegalArgumentException("Salary cannot be negative");

      if (salary <= 20000)
         return 0.1*salary;
      else if (salary <= 40000)
         return 2000 + 0.15*(salary-20000);
      else if (salary <= 80000)
         return 5000 + 0.20*(salary-40000);
      else
         return 13000 + 0.30*(salary-80000);
   }
}
```

There are 5 distinct valid equivalence partitions; each of which result in a different value returned from the method. Although it is not explicitly stated here, we can safely assume that marks below 0 and above 100 are not valid.

There is therefore a total of 7 equivalence partitions to test: the original **5 valid** ones, and **2 invalid** ones (all negative numbers, and all numbers above 100).

We only need one test for each equivalence partition; since all values within that partition produce identical behaviour in the method being tested. Although we can randomly choose any value in that partition, by convention we normally choose a value that is roughly in the middle of the partition range.

Thus, for the 0-50 partition, we select 25; for the 51-60 partition, we select 55 and so on. For partitions with large ranges, such as the 2 invalid ones, we can randomly choose any value in that range.

9. Refer to PartitionExampleTest, the parameterised test methods `testGetGradeValidValues()` and `testGetGradeInvalidValues()` demonstrate the 7 tests that need to be conducted for all 7 partitions. Notice that for `testGetGradeInvalidValues()`, we do not include any `assert` statement as we expect an exception to be thrown once the method is invoked; so **no value will be returned**. Run `PartitionExampleTest` to verify all tests pass.

```
package my.edu.utar;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
import my.edu.utar.PartitionExample;

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

import static org.junit.Assert.assertEquals;

@RunWith(JUnitParamsRunner.class)
public class PartitionExampleTest {

   PartitionExample pe = new PartitionExample();

   @Test
   @Parameters({"25,F", "55,D", "65,C", "75,B", "90,A" })
   public void testGetGradeValidValues(int mark, String expectedGrade) {
      String grade = pe.getGrade(mark);
      assertEquals(expectedGrade, grade);
   }

   @Test(expected=IllegalArgumentException.class)
   @Parameters({"-5", "120"})
   public void testGetGradeInvalidValues(int mark) {
      pe.getGrade(mark);
   }
```

Exercise

Look at the `calculateTax()` method in the class `PartitionExample`. It calculates the tax to be paid, given the salary of an employee. The following table summarises the manner in which the tax is calculated. Write a test method that tests all partitions for this method.

Salary range	**Tax to be paid**
20,000 and below	10% on the amount
20,001 – 40,000	2000 + 15% on the amount above 20,000
40,001 – 80,000	5000 + 20% on the amount above 40,000
80,001 and above	13000 + 30% on the amount above 80,000

10. The `combineStrings()` method in the `AnotherExample` class does not involve division of output based on valid partitions as in the case of the previous two methods.

    Here, we will create partitions based on this principle:

    - one partition to represent an **unsuccessful** search (empty string returned)
    - another to represent a **successful** search (one or more smaller substrings returned).

    For each partition, think of two combinations of different input parameters to produce the desired result. For example, an empty string can be returned if the length of the words specified by x is larger than all the substrings in words; or if words itself is an empty string.

```java
package my.edu.utar;

import java.util.StringTokenizer;

public class AnotherExample {

   // Given a String words containing a sequence of smaller strings and
   // an integer x, return a String which contains the smaller strings
   // from words whose length is greater than x.

   public String combineStrings(String words, int x) {
      StringTokenizer st = new StringTokenizer(words);
      String returnStr = "";

      while (st.hasMoreElements()) {
         String currentWord = (String) st.nextElement();
         if (currentWord.length() > x)
               returnStr = returnStr + currentWord + " ";
      }
      return returnStr.trim();
   }
}
```

    Notice that there is no existing code to check for invalid parameters (such as `null` for the parameter words).

    Also, while we may initially consider negative values of x as invalid, if we read the specification carefully, we should realise that the method still works fine with negative values. In such a situation, it will simply return all the substrings in the parameter word, the length of each substring is at least 1.

11. Refer to the class `AnotherExampleTest`, the first two combinations of input parameters for `testCombineStringsValidValues()` test for the first partition (empty string returned); while the next three combinations test for the second partition (string with one or more substrings returned). Notice that the last combination tests with a negative value for x.

    In the second test method, `testCombineStringsInvalidValuesV1` it calls `combineStrings`, passing it `null` for the parameter words.

```
package my.edu.utar;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
import my.edu.utar.AnotherExample;

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

import static org.junit.Assert.assertEquals;

@RunWith(JUnitParamsRunner.class)
public class AnotherExampleTest {

   AnotherExample ae = new AnotherExample();

   @Test
   @Parameters(method="CombineStringParam")
   public void testCombineStringsValidValues(String words, int x, String
   expectedResult) {
      String result = ae.combineStrings(words, x);
      assertEquals(expectedResult, result);
   }

   private Object[] CombineStringParam(){
      return new Object[]{
         new Object[]{"cat mouse horse",10,""}, //invalid partition test
         new Object[]{"",2,""},  //invalid partition test
         new Object[]{"cat dog horse",3,"horse"},
         new Object[]{"mouse house boat",4,"mouse house"},
         new Object[]{"cat dog",-5,"cat dog"}
      };
   }

   @Test
   public void testCombineStringsInvalidValuesV1() {
      ae.combineStrings(null, 5);
   }
}
```

As we can see from the result of running `AnotherExampleTest`, 5 test runs for `testCombineStringsValidValues` pass, while error (marked with red x) occur for `testCombineStringsInvalidValuesV1`.

A test **failure** occurs when an `assertXXXX` method is called and fails.
A test **error** occurs when an **unexpected Exception** occurs in a test method. The keyword here is **unexpected**.

To illustrate the difference between test failure and test error more clearly, make a modification to the expected result in one of the parameter combinations for `testCombineStringsValidValues`. Run `AnotherExampleTest` again. The test run that fails is marked as failure with a blue x in the JUnit view, as contrasted with the red x for the error in `testCombineStringsInvalidValuesV1`.

12. If we study the failure trace for `testCombineStringsInvalidValuesV1`, it will show the sequence of statements and method calls that led up to the `NullPointerException` being thrown.
    The first 2 lines of the trace indicate that the Exception was thrown within the constructor of the `StringTokenizer` class. This occurs because in the `combineStrings` methods, an attempt is made to instantiate a `StringTokenizer` object by passing the `words` parameter (which has the value `null`) to the constructor of `StringTokenizer`.

    ```
    StringTokenizer st = new StringTokenizer(words);
    ```

    `combineStrings` in turn is called by `testCombineStringsInvalidValuesV1` in `AnotherExampleTest`. This in turn is called by the `runChild` method that is part of the `JUnitParams` framework.

    If an Exception occurs and it is not caught and handled within an enclosing `try-catch` block, the exception is propagated upwards to the higher level method that called it. In this example, the `runChild` method calls the `testCombineStringsInvalidValuesV1` method, which in turn calls the `combineStrings` method, which in turn calls the constructor of `StringTokenizer`.
    As there is no `try-catch` block anywhere in any of these methods, when the `NullPointerException` is thrown in the `StringTokenizer` constructor, it is propagated all the way up to the JVM, which causes premature termination of the unit test. This is signalled in the JUnit view as an error. When a test failure or test error occurs, this indicates there is fault or bug in either the test or the method being tested; which necessitates appropriate debugging.

13. The following is the version of `combineStrings` that is correctly implemented to check for invalid parameter values.

    ```
    public String combineStrings(String words, int x) {

       if (words == null) throw new IllegalArgumentException();
       StringTokenizer st = new StringTokenizer(words);
       String returnStr = "";

       while (st.hasMoreElements()) {
          String currentWord = (String) st.nextElement();
          if (currentWord.length() > x)
             returnStr = returnStr + currentWord + " ";
       }
       return returnStr.trim();
    }
    ```

    In `AnotherExampleTest`, the `testCombineStringsInvalidValuesV2` is correctly implemented to test that an invalid parameter value is properly handled in `combineStrings`. Make the necessary changes in both source code files, and run `AnotherExampleTest`. All tests should pass successfully.

    ```
    @Test(expected=IllegalArgumentException.class)
    public void testCombineStringsInvalidValuesV2() {
       ae.combineStrings(null, 5);
    }
    ```

UECS2354 Software Testing
Lab 07: Designing Tests (Part 1)

Exercise

The methods in `OtherMethods` class do not provide checking for invalid parameters. Rewrite these methods so that they include code to check for invalid parameters, and create parameterised tests for the rewritten methods that include values from valid and non-valid equivalence partitions.

Note:

There is no fixed approach for determining the different valid partitions available for the parameters of these methods. The partitions could reflect differences in the values of the parameters (e.g. empty array vs array with elements), or the results returned by the method (e.g. an empty string vs. a string with contents). The tester determines this based on the functional specification of the method, and analysing the implemented code; if this is available.

UECS2354 Software Testing

Lab 07: Designing Tests (Part 1)

Boundary Value Analysis Testing

Boundary value analysis (BVA) is based on testing at the boundaries between equivalence partitions. A boundary value can be formally defined as an input value or output value which is on the edge of an equivalence partition (for example the minimum or maximum value of a range), or at the smallest incremental distance on either side of an edge. In BVA, we consider boundaries in both **valid** and **invalid** input partitions.

The rationale for BVA is based on the principle of error clustering. Defects tend to be found in clusters, with 20% (or less) of the modules accounting for 80% (or more) of the defects. One of the places where errors tend to occur more often is at the boundary between partitions; and BVA is very effective at picking out these bugs.

Consider the grades / mark ranges example.

Mark range	Grade
0 – 50	F
51 – 60	D
61 – 70	C
71 – 80	B
81 – 100	A

With boundary value analysis, we can view the boundary as a dividing line between two partitions. Hence we have a value on each side of the boundary (but the boundary itself is not a value).

Consider the mark range 51-60 and its corresponding grade D. The boundary between grade D and F is between 50 and 51; and the boundary between grades D and C is between 60 and 61. To apply boundary value analysis, we will take the boundary values of each distinct partition to use in our tests.

Thus, for the 5 distinct valid equivalence partitions that we have here, we will test for the values 50, 51, 60, 61, 70, 71, 80 and 81. In addition, we also test for values at the boundary between the valid partitions and invalid partitions. Here, the invalid partitions are all negative numbers, and numbers above 100. So, we will test for -1, 0, 100 and 101.

Note that all the values that we use for BVA also meet the criteria of equivalence partitioning; that is, we should test at least one input value for each distinct partition. In BVA, we will end up testing 2 input values for each distinct partition: one at the lowest end of the partition, and the other at the highest end.

Therefore, by performing BVA, you will also automatically perform equivalence partitioning testing. BVA can be considered as a more comprehensive form of testing that increases the chances of detecting certain bugs that might not be located using equivalence partitioning alone. Of course, it also results in considerably more tests as well; so we may need to consider other criteria such as risk, budget and time constraints in deciding whether to use this approach.

For cases where the specification of the method does not involve division of output based on valid partitions (such as in `AnotherExample` and `OtherMethods`), we would normally create partitions to represent events such as an unsuccessful search (e.g. empty string returned), and another to represent a successful search (e.g. one or more smaller substrings in the string returned). The boundary values would then be those that result in an empty string returned, a string with one substring returned and an invalid input value.

Lab 07: Designing Tests (Part 1)

Exercise

Write test methods to test the 2 methods in the `PartitionExample` class using BVA.

```java
package my.edu.utar;

public class PartitionExample {

   public String getGrade(int mark) {

      if (mark < 0 || mark > 100)
         throw new IllegalArgumentException("Mark out of range");
      if (mark < 51)
         return "F";
      else if (mark < 61)
         return "D";
      else if (mark < 71)
         return "C";
      else if (mark < 81)
         return "B";
      else
         return "A";
   }

   public double calculateTax(int salary) {

      if (salary < 0)
         throw new IllegalArgumentException("Salary cannot be negative");

      if (salary <= 20000)
         return 0.1*salary;
      else if (salary <= 40000)
         return 2000 + 0.15*(salary-20000);
      else if (salary <= 80000)
         return 5000 + 0.20*(salary-40000);
      else
         return 13000 + 0.30*(salary-80000);
   }
}
```

UECS2354 Software Testing

Lab 07: Designing Tests (Part 1)

Decision table testing

1. Testing input combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical if not impossible. **Decision tables aid the systematic selection of effective combinations** and can have the beneficial side-effect of finding problems and ambiguities in the specification. It works well in conjunction with equivalence partitioning. The combination of conditions explored may be combinations of equivalence partitions.

2. A decision table shows combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects). Decision table testing is focused on designing test cases to execute the combination of inputs and/or stimuli (causes) shown in a decision table.

3. Example of decision table that models a specific scenario:

	Rule 1	**Rule 2**	**Rule 3**
Condition 1	T	F	T
Condition 2	T	T	T
Condition 3	T	-	F
Action 1	Y	N	Y
Action 2	N	Y	Y

   Each column in this table represents a business rule. Each column consists of a combination of inputs with certain values, and the associated actions that will be performed when those combination of values occur.

   - Business rule 1 requires all conditions to be true to generate action 1.
   - Business rule 2 results in action 2 if condition 1 is false and condition 2 is true but does not depend on condition 3.
   - Business rule 3 requires conditions 1 and 2 to be true and condition 3 to be false.

Exercise

Create a decision table corresponding to the scenario below.

A supermarket has a loyalty scheme that is offered to all customers. Upon purchasing an item:

- Loyalty cardholders can either choose to obtain an additional 10% discount on all future purchases OR accumulate loyalty points.
- The number of loyalty points accumulated is equal to the value of the current purchase.
- Customers without a loyalty card obtain an additional 10% discount on all future purchases ONLY if they spend more than RM 100 on the current purchase
- Otherwise they are not entitled to anything

Note:

Based on the decision table, we should notice that each rule has a dash for one of the particular conditions, which indicates that it does not matter whether that particular condition is true or false. For example, a customer has no loyalty card and has spent 100 or less. In this situation, it does not matter whether the customer desires an extra discount, as he or she does not quality for it.

In a table with 3 Boolean conditions, there should be, by right, a total of $2^3 = 8$ possible condition combinations. However, we only have 4 rules in this scenario because each rule effectively covers 2 possible condition combinations.

4. The class `Customer` contains the instance variables required to model the above scenario; and the method `processPurchase()` contains the logic required to implement the decision table. Notice that the 3 parameters passed to the `processPurchase()` method reflects the 3 conditions in the table (whether the customer has a loyalty card, whether the customer chooses a discount and the amount spent on the purchase).

```
package my.edu.utar;

class Customer{

   private int loyaltyPoints, extraDiscount, defaultDiscount;

   public Customer(int loyaltyPoints, int extraDiscount, int
   defaultDiscount) {
      this.loyaltyPoints = loyaltyPoints;
      this.extraDiscount = extraDiscount;
      this.defaultDiscount = defaultDiscount;
    }

   public int getLoyaltyPoints() {
      return loyaltyPoints;
   }

   public int getExtraDiscount() {
      return extraDiscount;
   }

   public int getDefaultDiscount() {
      return defaultDiscount;
   }

   public void processPurchase(boolean haveCard, boolean chooseDiscount,
   int amountSpent)
   {
      if (amountSpent < 0)
         throw new IllegalArgumentException();

      if (haveCard)
      {
         if (chooseDiscount)
            extraDiscount = defaultDiscount;
         else
            loyaltyPoints += amountSpent;
      }
      else if (amountSpent > 100)
         extraDiscount = defaultDiscount;
   }
}
```

5. The `DemoDecisionTablesTest` class provides parameterised tests for all business rules in the decision table. Each business rule in fact covers 2 combinations of condition, since each rule has one irrelevant condition. We will still need to write tests for all 8 input combinations possible.

   Example, the first, where the customer has no loyalty card, has chosen a discount, and has spent 100 or less (false,true,0); and the second, where the customer has no loyalty card, has NOT chosen a discount, and has spent 100 or less (false,false,100).

   Notice that we use 2 boundary values for the amount spent in both tests (0 and 100). These 2 tests are necessary to confirm that the condition of extra discount selected (whether true or false) has no impact on the outcome.

   Study the class and compare it with the decision table.

```java
package my.edu.utar;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
import my.edu.utar.Customer;

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

import static org.junit.Assert.assertEquals;

@RunWith(JUnitParamsRunner.class)
public class DemoDecisionTablesTest {

   Customer  cus = new Customer(0,0,10);

   @Test
   @Parameters({"false,true,0,0,0", "false,false,100,0,0",
         "false,true,101,10,0", "false,false,200,10,0",
         "true,true,100,10,0", "true,true,101,10,0",
         "true,false,100,0,100", "true,false,101,0,101"})
   public void testProcessPurchaseValidValues
   (boolean haveCard, boolean chooseDiscount, int amountSpent,
         int expectedDiscount, int expectedLoyaltyPoints) {

      cus.processPurchase(haveCard, chooseDiscount, amountSpent);
      assertEquals(expectedDiscount, cus.getExtraDiscount());
      assertEquals(expectedLoyaltyPoints, cus.getLoyaltyPoints());
   }

   @Test(expected=IllegalArgumentException.class)
   @Parameters({"false,true,-1", "false,false,-2", // rule 1
      "true,false,-4", "true,true,-3"})
   public void testProcessPurchaseInvalidValues(boolean haveCard,
   boolean chooseDiscount,
         int amountSpent)      {
      cus.processPurchase(haveCard, chooseDiscount, amountSpent);
   }
}
```

UECS2354 Software Testing
Lab 07: Designing Tests (Part 1)

Exercise
Consider the description of another sample scenario below:

A supermarket has a scheme calculating the discount offered to customers on the following basis:

- If a customer opens a loyalty account with the supermarket, they obtain a 15% discount on all their purchases today.
- If a customer already has a loyalty account with the supermarket, they obtain a 10% discount on all their purchases today
- If a customer has a coupon they obtain 20% discount on all their purchases today.
- Discounts obtained can be accumulated with the condition that the coupon cannot be used in conjunction with the first option (i.e. a customer cannot choose to use the coupon and at the same time also open a loyalty account).
- We assume that given the choice between two possible discount options, the customer will choose the one that provides the maximum discount.

Create a decision table to help decide what discount a customer will receive based on the combination of conditions for the customer's state. From the table, implement a method that calculates the discount, and create a series of tests for this method.