[Handout for L8P2]
Testing our defenses from inside and outside

Test case design approaches

Exploratory testing vs scripted testing

Previously, we learned about two alternative approaches to test case design: exploratory vs scripted. That distinction was based on when test cases are designed. In scripted approach, they are designed in advance. In the exploratory approach, they are designed on the fly (to some extent).

Given next are alternative approaches based on some other aspects of testing.

Black-box vs white-box

This categorization is based on how much of SUT (software under test) internal details are considered when designing test cases. In the *black-box* approach, also known as *specification-based or responsibility-based testing*, we design test cases exclusively based on SUT's specified external behaviour. In the *white-box* approach, also known as *glass-box or structured or implementation-based testing*, we design test cases based on what we know about the SUT's implementation i.e. the code.

Knowing *some* important information about the implementation can actually help in black-box testing. This kind of testing is sometimes called *gray-box* testing. For example, if the implementation of a sort operation uses an algorithm to sort lists shorter than 1000 items and another to sort lists longer than 1000 items, we can add more meaningful test cases to verify the correctness of both algorithms.

Defensive testing vs design-by-contract testing

Test case design depends on if a defensive approach or a design-by-contract approach is used in coding. If a design-by-contract approach is used, one need not check an SUT for invalid input that violates pre-conditions specified in the contract. However, this is applicable only for testing programmable SUTs (i.e. API testing) as it is not wise to assume end users of a system (i.e. those using it via a UI) will not try to enter invalid inputs. If a defensive approach is used, we should check whether the SUT responds appropriately to both valid and invalid inputs.

Test case design techniques

If we want to test all possible variations of using the system, we may have to write an infinite number of test cases. For example, consider test case for adding a String item to a List

- Add an item when there are no other items in the collection
- Add an item when there is one item in the collection
- Add an item when there are two, three, ... items in the collection
- Add an item that has an English, a French, a Spanish, … word
- Add an item that is same as an existing item
- Add an item immediately after adding another item
- Add an item immediately after system startup
- ...

Exhaustive testing of this operation can take many more test cases. As you can see from that example, except for trivial systems, exhaustive testing is not possible!

Program testing can be used to show the presence of bugs, but never to show their absence! --Edsger Dijkstra

Every test case adds to the cost of testing. In some systems, a single test case can cost thousands of dollars (e.g. on-field testing of flight-control software). Therefore, we have to design our test cases wisely such that we make the best use of our testing resources. In particular,

- Testing should be **effective**: i.e. it finds a high % of existing bugs. A set of test cases that finds 60 defects is more effective than a one that finds only 30 defects in the same system.
- Testing should be **efficient**: i.e. it has a high rate of success (bugs found/test cases). A set of 20 test cases that finds 8 defects is more efficient than another set of 40 test cases that finds the same 8 defects.

That is why we have to learn various tools and techniques to improve E&E (Effectiveness and Efficiency) of testing. Next, let us look at some techniques useful for that.

Equivalence partitioning

Let us consider testing of the following operation.

isValidMonth (int m): boolean

Description: checks if m is in the range [1..12]. returns true if m is in the range, false otherwise.

It is inefficient and impractical to test for all integer values [-MIN_INT to MAX_INT]. Fortunately, we need not test for all possible input values. For example, if the input value 233 failed to produce the correct result, 234 is likely to fail too. We need not test both. In general most operations do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways. That means a range of inputs are treated the same way inside the operation. Testing one of the inputs *should be* as good as exhaustively testing all of them.

Equivalence partitioning is a technique that uses the above observation to improve the efficiency and effectiveness of testing. If we divide possible inputs into groups which are likely to be processed similarly by the SUT, we do not have to test every possible input in a given group. Such groups of input are called *equivalence partitions* (or *equivalence classes*). Equivalence partitioning can minimize test cases that are unlikely to find new bugs.

Equivalence partitions are usually derived from the specification of the SUT. Preconditions and postconditions too can help in identifying partitions. For example, these could be the Eq. partitions for the isValidMonth example:

- [MIN_INT ... 0] (below the range that produces 'true')
- [1 ... 12] (the range that produces 'true')
- [13 ... MAX_INT] (above the range that produces 'true')

Note that EP technique does not tell us how many test cases to pick from each partition. It depends on how thoroughly we want to test.

Here's an example from an OO system. What are the equivalence partitions for newGame() operation of Logic component? In general, we have to consider equivalence partitions of all *data participants*[5] that take part in the operation such as the target object of the operation call, input

[5] We use the term "data participants" to mean both objects and primitive values. Note that this is not a standard term

parameters of the operation call, and other data/objects accessed by the operation such as global variables. Here are the equivalence partitions for the push operation.

First, we should identify all "participants" of the operation.

- Parameters
- The object itself

Since newGame() does not have any parameters, the only participant is Logic itself. Note that if we are using a glass-box or a grey-box approach, we might even include other associated objects that are involved in the operation as participants. For example, Minefield object can be considered as another participant of the newGame() operation. Here, we assume a black-box approach.

Next, we identify eq. partitions for each participant. Will the newGame operation behave differently for different Logic objects? If yes, how will it differ? Yes, it might behave differently based on the game state. Therefore, out equivalence partitions are: PRE_GAME (i.e., before the game is started, minefield does not exist yet), READY (i.e., a new minefield has been created and waiting for player's first move), IN_PLAY, WON, LOST

Similarly, here are the equivalence partitions for the markCellAt(int x, int y) operation. The partitions in bold represent valid inputs:

- Logic: PRE_GAME, **READY, IN_PLAY**, WON, LOST
- x: [MIN_INT..-1] **[0..(W-1)]** [W..MAX_INT]
- y: [MIN_INT..-1] **[0..(H-1)]** [H..MAX_INT]
- Cell at (x,y): **HIDDEN**, MARKED, CLEARED

Should we test incorrect input stated above? It depends on whether we are using the defensive approach (should test incorrect input) or the design-by-contract approach (no need to test incorrect input).

Here's another example. Let us consider the push operation from a DataStack class.

Operation: push(Object o): boolean
Description: Throws MutabilityException if the global flag FREEZE==true
Else, throws InvalidValueException if o==null.
Else, returns false if the stack if full.
Else, put o at the top of the DataStack and returns true.

Here are the equivalence partitions for the push operation.

- DataStack object (ds): [full] [not full]
- o: [null] [not null]
- FREEZE: [true][false]

A test case for the push operation can be a combination of the equivalence partitions. Given below is such a test case.

id: DataStack_Push_001
description: checks whether pushing onto a full stack works correctly
input: ds is full, o!=null, FREEZE==false
expected output: returns false, ds remains unchanged

How do we combine these equivalence partitions and how many test cases should we create? This question is addressed later in this handout. Also notice how we mention the return value as well as the ds object under the "expected output". In general, the expected output should specify the return value as well as the state of all data participants of the operation that may be changed during the operation.

When deriving equivalence partitions for a given data participant, we use our knowledge of how the SUT behaves. Table below gives some examples. However, note that EP technique is a mere heuristic and not an exact science. The partitions we come up with depend on how we 'speculate' the SUT to behave internally. Applying EP under glass-box or gray-box approach can yield more precise partitions.

Specification	**Equivalence partitions**
isValidFlag(String s): boolean Returns true if s is one of ["F", "T", "D"]. The comparison is case-sensitive.	["F"] ["T"] ["D"] ["f", "t", "d"] [any other string][null] * ["f", "t", "d"] is optional
squareRoot(String s): int Pre-conditions: s represents a positive integer Returns the square root of s if it is an integer; returns 0 otherwise.	[s is not a valid number] [s is a negative integer] [s has an integer square root] [s does not have an integer square root]
isPrimeNumber(int i): boolean Returns true if i is a prime number	[prime numbers] [non-prime numbers] * there are too many prime numbers to consider each one as a separate equivalence partition

When a data participant of an SUT is expected to be a subtype of a given type, each subtype that has a bearing on the SUT's behavior should be treated as a separate equivalence partition. For example, consider the following operation.

Operation: compare(Expression first, Expression second): boolean
Description: returns true if both expressions evaluates to the same value

If the Expression is an abstract class which has two sub classes Sum and Product, we have to test the operation for both parameter types Sum and Product.

Boundary Value Analysis

Boundary value analysis is another heuristic that can enhance the E&E of test cases designed using equivalence partitioning. It is based on the observation that often bugs result from incorrect handling of boundaries of an equivalence partition. This is not surprising, as the end points of the boundary is often used in branching instructions etc. where the programmer can make mistakes.

E.g. markCellAt(int x, int y) operation could contain code such as
if (x > 0 && x <= (W-1)) which involves boundaries of x's equivalence classes.

When using boundary value analysis, we test the values around the boundary of an equivalence partition. Typically, we choose one value from the boundary, one value just below the boundary, and one value just above the boundary. Table below gives some examples of boundary values.

Equivalence partition	**Some possible boundary values**
[1-12]	0,1,2, 11,12,13
[MIN_INT, 0] *MIN_INT is the minimum possible integer value allowed by the environment.	MIN_INT, MIN_INT+1, -1, 0 , 1
[any non-null String]	Empty String, a String of maximum possible length
[prime numbers], ["F"], ["A", "B", "C"]	No specific boundary
[non-empty Stack]	Stack with: one element, two elements, no empty spaces, only one empty space

Combining multiple inputs

Often, an SUT can take multiple data participants. Once we have selected values to test for each data participant (using equivalence partitioning, boundary value analysis, or some other technique), how do we combine these values to create test cases? Consider the following scenario.

Operation to test:

calculateGrade(participation, projectScore, isAbsent, examScore)

Values to test (invalid values are underlined)
participation: 0, 1, 19, 20, <u>21</u>, <u>22</u>
projectScore: A, B, C, D, F
isAbsent: true, false
examScore: 0, 1, 69, 70, <u>71</u>, <u>72</u>
Given next are some techniques we can use here.

All combinations - This technique has a higher chance of discovering bugs. However, the number of combinations can be too high to test. In the above example, we will have to test 6x5x2x6=360 cases.

At least once – This technique, illustrated in the table below, aims to include every value at least once.

Table 1. Test cases for calculateGrade (V1.0)

Case No	participation	projectScore	isAbsent	examScore	Expected
1	0	A	True	0	...
2	1	B	False	1	...
3	19	C	AVV	69	...
4	20	D	AVV	70	...
5	21	F	AVV	71	Err Msg
6	22	AVV	AVV	72	Err Msg

AVV = Any Valid Value, Err Msg = Error Message

This technique uses one test case to verify multiple input values. For example, test case 1 verifies SUT for participation==0, projectScore==A, isAbsent==Ture, and examScore==0. However, the expected result for test case 5 could be an error message, because of the invalid input data. This means we may never know whether the SUT works correctly for the projectScore==F input as it is not being used by the other 4 test cases that do not produce an error message. Furthermore, if the error message was due to participation==21 then we cannot be sure whether examScore==71 too will return the correct error message. This is why we should test invalid input one at a time, and not combine them with the testing of valid input values. This results in 9 test cases, as shown in the table below.

Table 2. Test cases for calculateGrade (V2.0)

Case No	participation	projectScore	isAbsent	examScore	expected
1	0	A	True	0	...
2	1	B	False	1	...
3	19	C	AVV	69	...
4	20	D	AVV	70	...
5	AVV	F	AVV	AVV	...
6	21	AVV	AVV	AVV	Err Msg
7	22	AVV	AVV	AVV	Err Msg
8	AVV	AVV	AVV	71	Err Msg
9	AVV	AVV	AVV	72	Err Msg

A further necessary improvement to this approach is to ensure testing of all combinations between interlinked parameters. For example, let us assume a link between isAbsent and examScore that says an absent student can only have examScore==0. To cater for the hidden invalid case arising from this, we can add a 10th test case for which isAbsent==True and examScore!=0. In addition, test cases 3-9 should have isAbsent==false so that the input remains valid.

Table 0.3. Test cases for calculateGrade (V3.0)

Case No	participation	projectScore	isAbsent	examScore	Expected
1	0	A	True	0	…
2	1	B	False	1	…
3	19	C	False	69	…
4	20	D	False	70	…
5	AVV	F	False	AVV	…
6	21	AVV	False	AVV	Err Msg
7	22	AVV	False	AVV	Err Msg
8	AVV	AVV	False	71	Err Msg
9	AVV	AVV	False	72	Err Msg
10	AVV	AVV	True	!=0	Err Msg

All-pairs – This technique creates test cases so that for any given pair of parameters, all combinations between them are tested. It is based on the observations that a bug is rarely a result of more than two interacting factors. The resulting number of test cases is lower than the "all combinations" approach, but higher than the "at least once" approach. The technique for creating such a set of test cases is beyond the scope of this handout.

Using use cases

Use case testing is straightforward in principle: we base our test cases on the use cases. It is used for testing the system as a whole. For example, the main success scenario can be one test case while each variation (due to extensions) can form another test case. However, note that use cases do not specify the exact data entered to the system. Instead, it might say something like "user enters his personal data to the system". Therefore, the tester has to choose data by considering equivalence partitions and boundary values. The combinations of these could result in one use case producing many test cases. To increase E&E of testing, we have to focus more on high-priority use cases. For example, we can use scripted approach to test high priority test cases and exploratory approach to test other areas of concern that could emerge during testing.

Coverage-based testing

In the context of testing, *coverage* is a metric used to measure the extent to which testing exercises the code. Here are some examples of different coverage criteria:

- **Function/method coverage** measures the coverage in terms of functions executed e.g. testing executed 90 out of 100 functions.
- **Statement coverage** measures coverage in terms of executed lines of the source code e.g. testing executed 23k out of 25k LOC.
- **Decision/branch coverage** measures coverage in terms of decision points e.g. an if statement evaluated to both true and false.

- **Condition coverage** measures coverage in terms of boolean sub-expressions evaluated both to true and false. Condition coverage is not the same as the decision coverage; e.g. if(x>2 && x<44) is considered one decision point but two conditions. For 100% branch or decision coverage, we will need 2 test cases:

  (x>2 && x<44) == true [e.g. x = 4]

  (x>2 && x<44) == false [e.g. x = 100]

  For 100% condition coverage, we will need 3 test cases

  (x>2) == true , (x<44) == true [e.g. x = 4]

  (x<44) == false [e.g. x = 100]

  (x>2) == false [e.g. x = 0]
- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% path coverage means all possible paths have been executed. A commonly used notation for path analysis is called *Control Flow Graphs (CFG)*. If you are not familiar with CFGs, the side-note below introduces you to CFGs.
- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits from* the operations in the SUT.

Measuring coverage is often done using *coverage analysis tools*. We can use coverage measurements to improve E&E of our testing. For example, if a set of test cases (designed using other techniques) does not achieve 100% branch coverage, we need to add more test cases to cover missed branches.

[Side-Note] Control Flow Graphs (CFG)

CFG is a graphical representation of the execution paths of a code fragment. A CFG consists of:

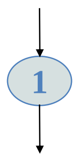
Nodes: Each node represents one or more sequential statements with no branches

Directed Edges: Each edge represents a branch, an possible path in execution

Given below is the CFG notation :

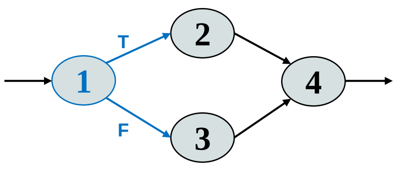
A set of sequential statements (without any branches) is represented as a single node. E.g.

```
x=2;        //node 1
y=3;        //node 1
z=x+y;      //node 1
print (z);  //node 1
```



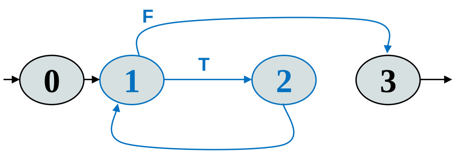
Conditional statements: E.g.

```
if x < 10 then //node 1
   z = x + y;//node 2
else   z = x – y;   //node 3
z = z + 2;          //node 4
```



Loops: E.g.

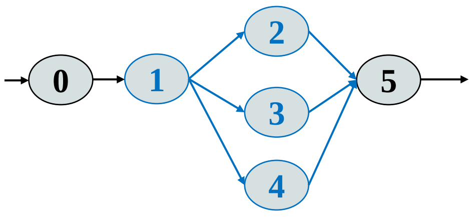
```
x++; //node 0
while (x < 10) { //node 1
   z = x+ y; //node 2
   x++;     //node 2
```



```
}
resetX(); //node 3
```

Multi-way branching: E.g.

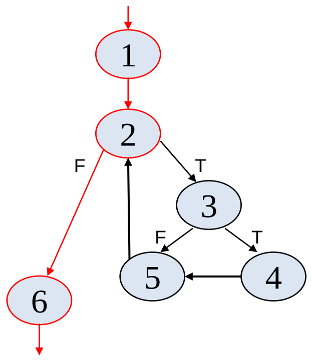
```
x++; //node 0
switch (x){ //node 1
 case 0:
   z = x; break;//node 2
 case 1:
 case 2:
   z = y; break;//node 3
 default:
   z = x-y; //node 4
}
z = x+y ; //node 5
```



Note how the same edge represents both case 1 and case 2.

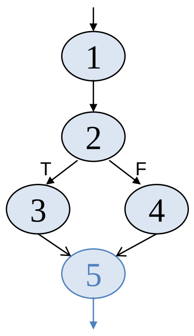
The figure below shows the complete CFG for the `min` function given below.

```
void min(int[] A){
  int min = A[0];          //node 1
  int i = 1;               //node 1
  int n = A.length;        //node 1
  while (i < n){   //node 2
   if (A[i] < min) //node 3
      min = A[i]; //node 4
         i++;              //node 5
  }
 print(min);               //node 6
}
```



It is recommended to have exactly one entry edge and exactly one exit edge for each CFG. Sometimes we have to add a *logical node* (i.e. a node that does not represent an actual program statement) to enforce "exactly one exit edge" rule. Node 5 in the Figure below is a logical node.

```
void foo(){
  int min = A[0];          //node 1
  if (A[i] < min)          //node 2
     min = A[i];           //node 3
  else
     i++;                  //node 4
  }
```



```
            }
```

A *path* is a collection of nodes that can be traversed from the entry edge to the exit edge in the direction of edges that link them. For example, 1-2-4-5 in the above CFG is a path

Other QA techniques

There are many QA techniques that do not involve executing the SUT. Given next are some such techniques that can complement the type of testing we have discussed so far in this chapter.

Inspections & reviews

Inspections involve a group of people systematically examining a project artifact to discover defects. Members of the inspection team play various roles during the process, such as the *author* - the creator of the artifact, the *moderator*- the planner and executer of the inspection meeting, the *secretary* - the recorder of the findings of the inspection, and *inspector/reviewer*- the one who inspects/reviews the artifact. All participants are expected to have adequate prior knowledge of the artifact inspected. An inspection often requires more than one meeting. For example, the first meeting is called to brief participants about the item to be inspected. The second meeting is called once the participants have studied the artifact. This is when the actual inspection is carried out. A third meeting could be called to re-inspect the artifact after the defects discovered as an outcome of inspection are fixed. An advantage of inspections is that it can detect functionality defects as well as other problems such as coding standard violations. Furthermore, inspections can verify non-code artifacts and incomplete code, and do not require test drivers or stubs. A disadvantage is that an inspection is a manual process and therefore, error prone.

Formal verification

Formal verification uses mathematical techniques to prove the correctness of a program. An advantage of this technique over code testing is that it can prove the absence of errors. However, one of the disadvantages is that it only proves the compliance with the specification, and not the actual utility of the software. Another main disadvantage is that it requires highly specialized notations and knowledge which makes it an expensive technique to administer. Therefore, formal verifications are usually used in safety-critical software such as flight control systems.

Static analyzers

These are tools that automatically analyze the code with an aim to find anomalies such as unused variables and unhandled exceptions. Detection of anomalies helps in improving the code quality. Most modern day IDEs come with some inbuilt static analysis capabilities. For example, an IDE will highlight unused variables as you type the code into the editor. Higher-end static analyzers can check for more complex (and sometimes user-defined) anomalies, such as overwriting a variable value before its current value is used.

Worked examples

[Q1]

a) Explain the concept of *exploratory testing* using Minesweeper as an example.

b) Explain why exhaustive testing is not possible using the newGame operation (from Logic class in the Minesweeper case study) as an example.

[A1]
(a) If we test the Minesweeper by simply playing it in various ways, especially, trying out those likely to be buggy, that would be exploratory testing.

b) Consider this sequence of test cases:

Test case 1. Start minesweeper. Activate newGame() and see if it works.

Test case 2. Start Minesweeper. Activate newGame(). Activate newGame() again and see if it works.

Test case 3. Start Minesweeper. Activate newGame() three times consecutively and see if it works.

…

Test case 267. Start Minesweeper. Activate newGame() 267 times consecutively and see if it works.

Well, you get the idea. Exhaustive testing of newGame() is not possible.

[Q2]
In the University of Nowere, students are given matriculation number according to the following format:

[Faculty Alphabet] [Gender Alphabet] [Serial Number] [Check Alphabet]

E.g.. CF1234X

The valid value(s) for each part of the matriculation number is given below:

Faculty Alphabet:
- Single Capital Alphabet
- Only 'C' to 'G' are valid

Gender Alphabet:
- Single Capital Alphabet
- Either 'F' or 'M' only

Serial Number
- 4 digits number
- From 1000 to 9999 only

Check Alphabet
- Single Capital Alphabet
- Only 'K', 'H', 'S', 'X' and 'Z' are valid

Assume you are testing the isValidMatric(String matric). Identify equivalence partitions and boundary values for the matriculation number.

[A2]
String length: (less than 7 characters), (7 characters), (more than 7 characters)

For those with 7 characters,

[Faculty Alphabet]: (‘C’, ‘G’), (‘c’, ‘g’), (any other character)

[Gender Alphabet]: (‘F’, ‘M’), (‘f’, ‘m’), (any other character)

[Serial Number]: (1000-9999), (0000-0999), (any other 4- characters string)

[Check Alphabet]: ('K', 'H', 'S', 'X', 'Z'), ('k', 'h','s', 'x', 'z'), (any other character)

[Q3]
Given below is the overview of the method dispatch(Resource,Task), from an emergency management system (e.g. a system used by those who handle emergency calls from the public about incidents such as fires, possible burglaries, domestic disturbances, etc.). A task might need multiple resources of multiple types. E.g. the task 'fire at Clementi MRT' might need two fire engines and one ambulance.

dispatch(Resource r, Task t):void

Overview: This method dispatches the Resource *r* to the Task *t*. Since this can dispatch only one resource, it needs to be used multiple times should the task need multiple resources.

Imagine you are designing test cases to test the method dispatch(Resource,Task). Taking into account equivalence partitions and boundary values, which different inputs will you combine to test the method?

[A3]

Test input for t	Test input for r
• A fully dispatched task • A task requiring one more resource • A task with no resource dispatched Considering the task types required • A task requiring only one type of resources • A task requiring multiple types of resource • Null	• A resource required by the task • A resource not required by the task • A resource already dispatched for anther task • null

[Q4]
Given below is an operation contract taken from a restaurant booking system. Use equivalence partitions and boundary values to design a set of test cases for it.

boolean transferTable (Booking b, Table t)
Description: Used to transfer a booking *b* to Table *t*, if *t* has enough room.
Preconditions: *t* has room for *b* , b.getTable() != t
Postconditions: b.getTable() == t

[A4]
Equivalence partitions

Booking:

Invalid: null, not null and b.getTable==t, Valid: not null and b.getTable != t

Table:

Invalid: null, not vacant, vacant but doesn't have enough room,

Valid: vacant and has enough room.

Boundary values:

Booking:

Invalid: null, not null and b.getTable==t,

Valid:not null and b.getTable != t

Table:

Invalid: null, not vacant, (booking size == table size + 1)

Valid: (booking size == table size), (booking size == table size-1)

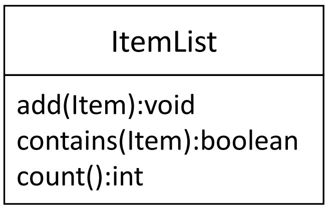
Test cases:

Test case	**Booking**	**Table**
1	null	Any valid
2	not null and b.getTable==t	Any valid
3	Any valid	null
4	Any valid	not vacant
5	Any valid	(booking size == table size + 1)
7	not null and b.getTable != t	(booking size == table size)
8	Any valid	(booking size == table size-1)

Note: We can use Bookings of different sizes for different test cases so that we increase the chance of finding bugs. If there is a minimum and maximum booking size, we should include them in those test cases.

[Q5]

Assume you are testing the add(Item) method specified below.



Assume i to be the Item being added.

preconditions:

i != null [throws invalidItemException if i == null]

contains(i) == false [throws duplicateItemException if contains(i) == true]

count() < 10 [throws listFullException if size() == 10] postconditions:

contains(i) == true;

new count() == old count()+1

Invariants: (an "invariant" is true before and after the method invocation).

0 <= count() <= 10

(a) What are the equivalence partitions relevant to testing the add(Item) method?

(b) What are the boundary, and non-boundary, values you will use to test the add(Item) method?

(c) Design a set of test cases to test the add(Item) method by considering the equivalence partitions and boundary values from your answers to (a) and (b) above. Test case 0 is for illustration only.

	Test input : i	Test input: ItemList object	Expected output
0	Any non-null Item	5 Items in the list, contains(i)==false	contains(i)==true count()==6
1			

[A5]

(a)

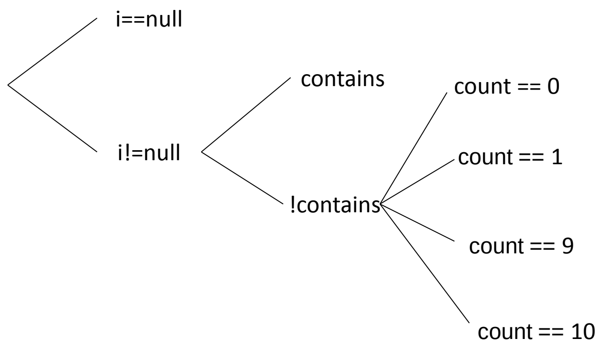
i: i != null, i == null

list: contains(i)==true, contains(i)==false, count() < 10, count() == 10

list == null should NOT be considered.

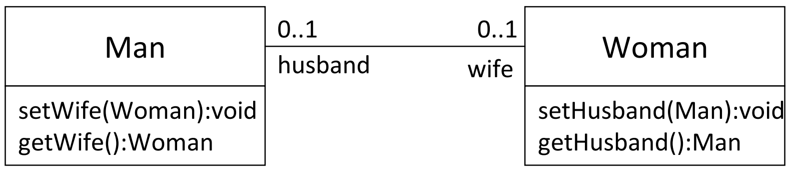
(b) list: count()==0, size==9, count()==10, count()== [1|2|3|4|6|7|8] (1 preferred)

(c)



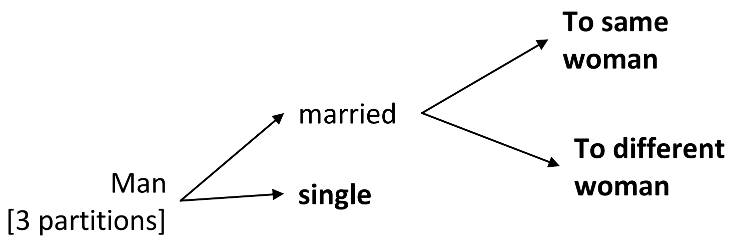
[Q6]

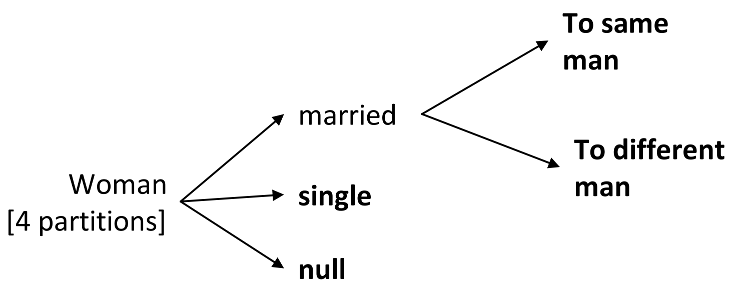
a) Use equivalence partitions and boundary values to choose test inputs for testing the setWife operation in the Man class.



b) Identify a set of equivalence partitions for testing isValidDate(String date) method. The method checks if the parameter date is a day that falls in the period 1880/01/01 to 2030/12/31 (both inclusive), given in the format yyyy/mm/dd.

[A6]
(a)





Partitioning 'married' as 'to same woman' and 'to different woman' seems redundant at first. Arguments for having it:

- The behavior (e.g. the error message shown) may be different in those two different situations.
- The 'to same woman' partition has a risk of misunderstanding between developer and user. For example, developer might thing it is ok to ignore the request while the users might expect to see an error message.

(b)

Initial partitions: [null] **[10-characters long]**[shorter than 10 characters][longer than 10 characters]

For 10-character strings:

- **c1-c4:** [not an integer] [less than 1880] **[1880-2030 excluding leap years][leap years within 1880-2030 period]**[2030-9999]
- **c5:** **["/"]**[not "/"]

- **c6-c7:** [not an integer][less than 1]**[2][31 day months: 1,3,5, 7,8, 10,12][30-day months: 4,6,9,11]** [13-99]
- **c8:** **["/"]**[not "/"]
- **c9-c10:** [not an integer][less than 1]**[1-28][29][30][31]**[more than 31]

In practice, we often use 'trusted' (e.g. those that come with the Java JDK or.NET framework) library functions to convert strings into dates. In such cases, our testing need not be as through as suggested by the above analysis.

[Q6]
Consider the following operation that activates an alarm when the landing gear (i.e. wheels structure) of an airplane should be deployed but has not been deployed.

```
isAlarmToBeSounded(TimeSinceTakeOff,
                   TimeToLand,
                   Altitude,
                   Visibility,
                   isGearAlreadyDeployed):boolean
```

Here is the logic for the operation. Landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet. The operation should return true if the landing gear is not deployed (i.e. isGearAlreadyDeployed ==false) on meeting a deployment condition. Assume that the smallest unit of measurement for time is 1s and for distance it is 1 feet. Also, assume that invalid input such as negative values will not be produced by the sensors. takeoff_max, landing_max, altitude_max, visibility_max are maximum values allowed by the instruments.

(a) List the equivalence partitions of the conditions that can lead to a decision about whether the alarm should be sounded.

(b) List boundary and non-boundary values you will test, using no more than one non-boundary value per equivalence partition.

(c) Ordinarily, for a critical system such as a "landing gear alarm system", all combination of equivalence partitions must be checked. Ignoring that requirement for a moment, design a minimal set of test cases that includes each eq. partition at least once. Use only boundary values.

(d) Now, you have been told that the correct functioning of the alarm is critically important if the plane is within 2 minutes of landing no matter what the other conditions are. How would you modify the test cases? Use only boundary values.

[A6]
(a) Equivalence partitions:

TimeSinceTakeOff:	[0s .. 2m] [2m1s .. takeoff_max]
Time to land :	[land_max.. 3m1s][3m .. 2m1s] [2m .. 0s]
Altitude:	[0 .. 2000], [2001 .. 2500], [2501 .. altitude_max]
Visibility:	[0 .. 1000], [1001 .. visibility_max]
isGearAlreadyDeployed:	[True] [False]

(b) take all values mentioned in (a), and one non-boundary value for each partition. E.g.

TimeSinceTakeOff: 0s, 1m, 2m, 2m1s, 3m, takeoff_max

(c) To test every partition at least once, we need only three test cases.

TimeSinceTakeOff	**TimeToLand**	**Altitude**	**Visibility**	**isGearAlreadyDeployed**
0s	land_max	0	0	[True]
2m1s	3m	2001	1001	[False]
-any value-	2m	2501	-any value-	-any value-

[Not required by the question] To test every boundary value at least once, we need six test cases.

TimeSinceTakeOff	**TimeToLand**	**Altitude**	**Visibility**	**isGearAlreadyDeployed**
0s	land_max	0	0	[True]
2m0s	3m1s	2000	1000	[False]
2m1s	3m	2001	1001	-any value-
takeoff_max	2m1s	2500	visibility_max	-any value-
-any value-	2m	2501	-any value-	-any value-
-any value-	0s	altitude_max	-any value-	-any value-

(d) Values 2m and 0s for the "TimeToLand" should be combined with every possible combination of the other four inputs.

For TimeToLand ==2m -> 4x6x4x2 -> 192 test cases

For TimeToLand == 0s -> 4x6x4x2 -> 192 test cases

For TimeToLand == land_max, 3m1s, 3m, 2m1s -> 4 test cases

In total we need 192+192+4=388 test cases.

We can also test a non-boundary value from the critical equivalence partition (e.g. TimeToLand ==1m), in which case, we will have an additional 192 test cases.

---End of Document---