Software Testing Tutorials

Tutorial 1: Category-Partition Testing
- Question Specification
- Solutions

Tutorial 2: Structural Testing
- Question Specification
- Solutions

Tutorial 3: Data Flow Testing
- Question Specification
- Solutions

Tutorial 4: Mutation Testing
- Question Specification
- Solutions

Tutorial 1: Category-Partition Testing

Question Specification

Task: *Derive test specifications using the category partition method for the following Airport connection check function.*

Airport connection check: the airport connection check is part of an (imaginary) travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. It is described here at a fairly abstract level, as it might be described in a preliminary design before concrete interfaces have been worked out.
Specification signature: Valid Connection (Arriving Flight: flight, Departing Flight: flight) returns validity code.
Validity code 0 (OK) is returned if Arriving Flight and Departing Flight make a valid connection (the arriving airport of the first is the departing airport of the second) and there is sufficient time between arrival and departure according to the information in the airport database described below.
Otherwise, a validity code other than 0 is returned, indicating why the connection is not valid.

Data types

- Flight: A “flight” is a structure consisting of
    - A unique identifying flight code, three alphabetic characters followed by up to four digits. (The flight code is not used by the Valid Connection function.)
    - The originating airport code (3 characters, alphabetic)
    - The scheduled departure time of the flight (in universal time)
    - The destination airport code (3 characters, alphabetic)
    - The scheduled arrival time at the destination airport.
- Validity Code: The validity code is one of a set of integer values with the following interpretations:
    - 0: The connection is valid.
    - 10: Invalid airport code (airport code not found in database)
    - 15: Invalid connection, too short: There is insufficient time between arrival of first flight and departure of second flight.
    - 16: Invalid connection, flights do not connect. The destination airport of Arriving Flight is not the same as the originating airport of Departing Flight.
    - 20: Another error has been recognized (e.g., the input arguments may be invalid, or an unanticipated error was encountered)
- Airport Database: The Valid Connection function uses an internal, in-memory table of airports which is read from a configuration file at system initialization. Each record in the table contains the following information:
    - Three-letter airport code. This is the key of the table and can be used for lookups.
    - Airport zone. In most cases the airport zone is a two-letter country code, e.g., ”US” for the United States. However, where passage from one country to another is possible without a passport, the airport zone represents the complete zone in which passport-free travel is allowed. For example, the code ”EU” represents the European countries which are treated as if they were a single country for purposes of travel.
    - Domestic connect time. This is an integer representing the minimum number of minutes that must be allowed for a domestic connection at the airport. A connection is “domestic” if the originating and destination airports of both flights are in the same airport zone.

- International connect time. This is an integer representing the minimum number of minutes that must be allowed for an international connection at the airport. The number -1 indicates that international connections are not permitted at the airport. A connection is “international” if any of the originating or destination airports are in different zones.

Activities
Having considered this specification you should carry out the following activities that will be facilitated by your tutor:

1. Individually, have a look again at the specification and write down a list of Independently Testable Functions (ITFs) in this specification. There might be more than one — however, carefully justify each ITF you have identified so you can argue for it. [take max 5 mins to do this]

2. Choose a partner you will work with in the tutorial and discuss your decision from activity 1 (if the tutorial has an odd number of students you will need to have one group of three). Agree on a final list of ITFs for this specification. [take max 5 mins for this activity]

3. Whole group discussion: read out your lists of ITFs: the tutor will write up distinct ITFs so you have a definitive list of ITFs — be selective, if the group does not find the justification convincing do not include a proposed ITF in the final list. Hint: there are probably only 2 or 3 ITFs depending on how you look at it. [take max 5 mins for this activity]

4. Select two of the ITFs and split the group to work on them separately. Each group should work in the following way:

5. Individually, each group member should work to identify the parameters and environment elements that are relevant to their ITF. Once you have identified these you should then work out characteristics of your parameters and environment elements. [allow around 10 mins for activities 4 and 5]

6. Together with your partner(s) agree your merged list, then agree the list with your group. [allow 5 mins for this activity].

7. Individually, identify value classes for each of your characteristics. Check these with your partner and agree a list. [allow 10 mins for this activity].

8. Together with your partner(s) work out how many test case specifications you have assuming no constraints between the value classes. Also, pay some attention to the expected result for a test. Then work together to try to identify situations where you can use constraints to reduce the number of possible test case specifications. [take a further 10 mins for this activity]

9. Finally, as a group try to merge your efforts to create a set of test case specifications for your chosen ITF. [allow 5 mins for this]

Solutions

Activities 1-3: Individually Testable Features (ITFs)

ITFs (also known as Individually Testable Functions) refer to an aspect or part of the system under test which can be tested in isolation.

The system under test in the course is usually so small/simple that one might identify the entire system as having just one ITF. This is arguably the case here: the Valid Connection function is intended to do one thing and one thing only: check whether two flights form a valid connection -- whether someone trying to make a longer journey can practically get off the first flight and onto the second. The decisions necessary involve establishing that the second flight departs from the airport at which the first arrives, and that the time gap between arrival and departure is sufficient for the appropriate transfer time for that airport (international transfers being treated differently to domestic transfers).

Three suggested refinements of the function's operation that came out of tutorials were as follows:

1. The function correctly identifies valid connections in situations where valid connections are offered. We'll call this ITF **VALID**. This corresponds to return code 0.
2. The function correctly identifies invalid connections in situations where non-connecting flights (either due to mismatched airports or insufficient transfer time) are submitted. We'll call this ITF **INVALID**. This corresponds to return code 15 and 16.
3. The function correctly identifies and flags error conditions. We'll call this ITF **ERROR**. This corresponds to return codes 10 and 20.

Use of documented system behaviour in order to break down tests like this is often a good idea. Surely however we could continue this process to break it down into on eITF per return code? It's possible, but we're beginning to split hairs, as will become apparent as we go on to develop characteristics for partitioning in the later exercises. Let's stick with the above three ITFs for the moment.

Activities 4-6: Parameters, environment and characteristics

Next we identify environmental factors and input parameters relevant to the ITFs. The input parameters are clearly identified as the arriving and departing flight data structures.

We could break these out into their structural fields (flight code, originating airport, departure time, destination airport, arrival time), but let's keep it at a high level for the time being.

Environmental factors are less clear. Remembering that the environment describes external factors which you need to configure in particular ways in order to specify and execute tests which fully exercise the system, the major environmental factor here is definitely the database: any test will need to look the arriving and departing flights' origin and destination airports up in the database in order to decide whether the connection is a domestic connection or an international one, so we'll need to specify what data is in the airport database when testing.

Other possible environmental factors include the following:

- System memory: the specification notes that the Valid Connection function uses an *in-memory* table of airports. Could the system run out of memory? If we consider the number of airports in the world

(about 44,000 in 2009 according to the CIA World Factbook) and the amount of data stored per airport (looks like only four small fields - a few bytes), this shouldn't be a problem on a modern PC.
- Start-up time: the specification mentions loading the airport database at system initialisation. This might cause responsiveness problems if initialisation takes a long time. However since the specification doesn't make any statements requiring the system to respond in a timely manner we can ignore this issue.
- Locale locale: for this kind of application, variations in time zone can cause huge amounts of trouble. The specification however states that times are expressed in universal time, so locale probably isn't an issue.

You should bear in mind when designing tests is your stopping point. We've already established that there are a very large number of factors which could affect system behaviour. We also know that there are usually a near-infinite number of possible inputs and control paths within even fairly simple software systems. Consequently, you could continue testing most systems until the end of the universe without running out of thests. However you should always be on the lookout for ways to get these infinities under control. *Plausible* arguments to amalgamate or throw out classes of test are very important here, and the above three points give an example of the kind of thought process you might follow in order to justify ignoring certain issues.

So in short, the answers are:
- Environment: airport database
- Input parameters: arriving flight, departing flight

Remember too that in a larger system we might have different environmental factors and inputs for each ITF. here however all three of our ITFs are affected by all three of the above factors.

Characteristics
Many characteristics which will affect the outcome of your tests can be lifted directly from the system specification. Again these will vary from ITF to ITF.
- ITF 1 (VALID)
    - Arriving and Departing Flight's originating and destination airports must all have entries in the airport database
    - Arriving Flight's destination airport must match Departing Flight's originating airport
    - If flight is international (either flight's destination airport zone is different to its originating airport zone), then the connecting airport must allow international transfers (airport database record does not have an international connect time of -1)
    - After the Arriving Flight's arrival time, there must be a gap no smaller than the appropriate (domestic or international)connect time before the Departing Flight's departure time
- ITF 2 (INVALID)
    - Arriving and Departing Flight's originating and destination airports must all have entries in the airport database
    - At least one of the following must be true: (can test jointly in ITF 4: CORRECT with ITF 1)
        - Arriving Flight's destination airport doesn't match Departing Flight's originating airport
        - If flight is international (either flight's destination airport zone is different to its originating airport zone), then the connecting airport doesn't allow international transfers (airport database record does not have an international connect time of -1)
        - The Departing Flight must leave before the Arriving Flight lands, or so soon afterwards that there's not enough (domestic or international) connect time

- ITF 3 (ERROR)
    - At least one of the Arriving and Departing Flight’s originating and destination airports doesn’t have an entry in the airport database
    - Various other input data errors, such as invaid arrival or departure times or badly formed flight codes or airport codes (overlaps a bit with the first point), or missing (null?) Arriving or Departing Flight parameters
    - Various database content errors, such as badly formed airport zone codes or invalid connect times
    - More fundamental database problems such as the database not being available

Some of the above might be difficult to test without more information: without knowing what programming language we're working with, we don't know how to provide a null (or equivalent) Arriving Flight for example.

Again, it's worth noting that these characteristics are almost all just bad values for the same information we're using to partition ITF 4; if we include invalid data of various kinds in our partitions for ITF 4, then we'll actually cover ITF 3 pretty well. Let's do that: by including invalid inputs in ITF 4 we can forget about ITF 3, and just have a single ITF.

Activity 7: Partitions/value classes
Shorthand notations:

Acronym	**Meaning**
AF	Arriving Flight
DF	Departing Flight
IFC	Identifying Flight Code
OAC	Originating Airport Code
SDT	Scheduled Departure Time
DAC	Destination Airport Code
SAT	Scheduled Arrival Time
AZ	Airport Zone
DCT	Domestic Connect Time
ICT	International Connect Time
CT	Connect Time

So now we write “AF -> DAC -> DCT” instead of “Arriving Flight’s Destination Airport’s Domestic Connect Time”.

Here are the partitions we derive from the characteristics we identified above. Note that we're now splitting the inputs out into some of their subfields so we can see when we're choosing between a set of mutually exclusive alternatives for a particular characteristic.

Characteristic	Partition	Value Classes	Comment
Database ok?	**P1**	Database ok	
	P2	Database not ok	Connect failure or similar
AF→OAC in DB?	**P3**	AF→OAC ∈ database	Arriving flight's originating airport is in database
	P4	AF→OAC ∉ database	Arriving flight's originating airport not in database
AF→DAC in DB?	**P5**	AF→DAC ∈ database	Arriving flight's destination airport is in database
	P6	AF→DAC ∉ database	Arriving flight's destination airport not in database
DF→OAC in DB?	**P7**	DF→OAC ∈ database	Departing flight's originating airport is in database
	P8	DF→OAC ∉ database	Departing flight's originating airport not in database
DF→DAC in DB?	**P9**	DF→DAC ∈ database	Departing flight's destination airport is in database
	P10	DF→DAC ∉ database	Departing flight's destination airport not in database
Flights meet?	**P11**	AF→DAC = DF→OAC	Flights meet at the same airport
	P12	AF→DAC ≠ DF→OAC	Flights don't meet at the same airport
AF international?	**P13**	AF→OAC→AZ = AF→DAC→AZ	Arriving flight is domestic (airport zones match)
	P14	AF→OAC→AZ ≠ AF→DAC→AZ	Arriving flight is international (airport zones don't match)
DF international?	**P15**	DF→OAC→AZ = DF→DAC→AZ	Departing flight is domestic (airport zones match)
	P16	DF→OAC→AZ ≠ DF→DAC→AZ	Departing flight is international (airport zones don't match)
Connect time ok?	**P17**	CT = -1	(International) transfers disallowed
	P18	DF→SDT - AF→SAT < CT	Departing flight leaves too early
	P19	DF→SDT - AF→SAT = CT	Departing flight leaves as early as permitted
	P20	DF→SDT - AF→SAT > CT	Departing flight leaves enough time to transfer

Activity 8: Constraints

On the face of it this leaves us with 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 × 4 = 1,024 combinations (two possibilities for each characteristic except connect time, for which we have four). Many partition combinations are impossible, and remember that all error values need to be tested on their own with all other inputs and environment factors set to valid values. Let's annotate our partitions accordingly:

Characteristic	**Partition**	**Value Classes**	**Conditions/properties**
Database ok?	**P1**	Database ok	
	P2	Database not ok	[error]
AF→OAC in DB?	**P3**	AF→OAC ∈ database	
	P4	AF→OAC ∉ database	[error]
AF→DAC in DB?	**P5**	AF→DAC ∈ database	
	P6	AF→DAC ∉ database	[error]
DF→OAC in DB?	**P7**	DF→OAC ∈ database	
	P8	DF→OAC ∉ database	[error]
DF→DAC in DB?	**P9**	DF→DAC ∈ database	
	P10	DF→DAC ∉ database	[error]
Flights meet?	**P11**	AF→DAC = DF→OAC	
	P12	AF→DAC ≠ DF→OAC	[error]
AF international?	**P13**	AF→OAC→AZ = AF→DAC→AZ	
	P14	AF→OAC→AZ ≠ AF→DAC→AZ	[property International]
DF international?	**P15**	DF→OAC→AZ = DF→DAC→AZ	
	P16	DF→OAC→AZ ≠ DF→DAC→AZ	[property International]
Connect time ok?	**P17**	CT = -1	[if International]
	P18	DF→SDT - AF→SAT < CT	
	P19	DF→SDT - AF→SAT = CT	[single]
	P20	DF→SDT - AF→SAT > CT	

Note we've annotated P19 as “single” just to test the boundary between too little connect time and enough connect time; the specification isn't clear on whether *exactly* the right connect time is safe or not, but let's assume that it will be. We use “single” because it's a special case that we think only needs investigating once; arguably we could do the same for P18 and P20 — there's a bit of value judgement/intuition here — but the idea is that not making these two single ensures that CT is tested thoroughly (too little CT and enough CT) for both domestic and international flights.

Now: if every “single” or “error” condition needs to be tested on its own with otherwise valid inputs, then we need seven such tests. Then we're left with non-error, non-single tests: our first six characteristics only have one remaining valid value and our connect time partition has three, so we now have 1 × 1 × 1 × 1 × 1 × 1 × 2 × 2 × 3 = 12 combinations left to test, for a total of 7 + 12 = 19 tests. In fact it's even less than that if we look at the “International” condition: for combination (P13, P15) (a domestic flight), International is false, so we don't test P17. We do test P17 for the three remaining International combinations (P13, P16), (P14, P15) and (P14, P16). This modifies our product from (2 × 2) × 3 to (1) × 2 + (3) × 3 = 11 (the (1) and (3) are a breakdown of (2 × 2) according to which combinations produce domestic and international flights). This saves us one test, so we're now down to 18.

Activity 9: Test case specifications
In this table we enumerate all 18 test case specifications; any characteristic for which we don't specify a partition must be chosen from one of its valid (non-error) partitions (but we don't care which). I make this explicit for test 7, the first non-error test, so we can see that all 20 partitions are covered.

Test	**Specification**	**Partitions covered**
1	Database not ok	P2
2	AF→OAC ∉ database	P4
3	AF→DAC ∉ database	P6
4	DF→OAC ∉ database	P8
5	DF→DAC ∉ database	P10
6	AF→DAC ≠ DF→OAC	P12
7	Domestic; shortest possible connect	P1, P3, P5, P7, P9, P11, P13, P15, P19
8	Domestic; connect too short	P13, P15, P18
9	Domestic; connect long enough	P13, P15, P20
10	Int'l DF; int'l disallowed	P13, P16, P17
11	Int'l DF; connect too short	P13, P16, P18
12	Int'l DF; connect long enough	P13, P16, P20
13	Int'l AF; int'l disallowed	P14, P15, P17

14	Int'l AF; connect too short	P14, P15, P18
15	Int'l AF; connect long enough	P14, P15, P20
16	Both int'l; int'l disallowed	P14, P16, P17
17	Both int'l; connect too short	P14, P16, P18
18	Both int'l; connect long enough	P14, P16, P20

Test cases

Environment: database contents			
Airport code	**Airport Zone**	**Domestic connect time**	**International connect time**
GLA	EU	30	90
EDI	EU	35	95
SKL	EU	25	-1
NRT	JP	29	85
JFK	US	40	120

The test cases now follow; since we don't use AF→IFC, DF→IFC, AF→SDT or DF→SAT (and didn't define tests based on them), I've not included them in the table.

Test	**Arriving flight**			**Departing flight**			**Return code**	**Notes**
	OAC	**DAC**	**SAT**	**OAC**	**SDT**	**DAC**		
1	GLA	NRT	13:00	NRT	15:00	JFK	20	Database broken
2	XXX	NRT	13:00	NRT	15:00	JFK	10	Bad AF→OAC
3	GLA	XXX	13:00	NRT	15:00	JFK	10	Bad AF→DAC
4	GLA	NRT	13:00	XXX	15:00	JFK	10	Bad DF→OAC
5	GLA	NRT	13:00	NRT	15:00	XXX	10	Bad DF→DAC
6	GLA	NRT	13:00	JFK	15:00	NRT	16	Flights don't meet
7	GLA	SKL	13:00	SKL	13:25	EDI	0	Shortest possible connect

8	GLA	SKL	13:00	SKL	13:24	EDI	15	Domestic; too short
9	GLA	SKL	13:00	SKL	13:26	EDI	0	Domestic; long enough
10	GLA	SKL	13:00	SKL	15:00	JFK	20	International DF; disallowed
11	GLA	EDI	13:00	EDI	14:34	JFK	15	International DF; too short
12	GLA	EDI	13:00	EDI	14:36	JFK	0	International DF; long enough
13	NRT	SKL	13:00	SKL	15:00	GLA	20	International AF; disallowed
14	NRT	EDI	13:00	EDI	14:34	GLA	15	International AF; too short
15	NRT	EDI	13:00	EDI	14:36	GLA	0	International AF; long enough
16	NRT	SKL	13:00	SKL	15:00	JFK	20	Both international; disallowed
17	NRT	EDI	13:00	EDI	14:34	JFK	15	Both international; too short
18	NRT	EDI	13:00	EDI	14:36	JFK	0	Both international; long enough

There are lots of other things we could think about here — other error conditions, what happens if arrival and departure times are on different days, etc. — but I don't want to overcomplicate this example and think this is a reasonable set of test cases.

Tutorial 2: Structural Testing

Question Specification

The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the i th point appearing in p, similarly for (p.get(i)).y. This exercise is concerned with structural testing of code and so the focus is on creating test sets that satisfy some particular coverage criterion.

```
Vector doGraham(Vector p) {
    int i, j, min, M;

    Point t;
    min = 0;

    // search for minimum
    for (i = 1; i < p.size(); ++i) {
        if ( ((Point) p.get(i)).y < ((Point) p.get(min)).y ) {
            min = i;
        }
    }

    // continue along the values with same y component
    for (i = 0; i < p.size(); ++i) {
        if ( ( ((Point) p.get(i)).y == ((Point) p.get(min)).y ) &&
                ( ((Point) p.get(i)).x > ((Point) p.get(min)).x ) ) {
                    min = i;
                }
    }
}
```

- Prerequisites: before the tutorial you should review the reading on structural testing. In particular, you should read Pezz`e and Young's chapter 12 which will provide adequate background for this tutorial.
- Preparation: Review the code fragment drawn from the doGraham method above. If need be, check the documentation on the Vector class and any other Java documentation you might require. •

Activities

1. First Activity (around 10 minutes): Individually, convert the Java code comprising the beginning of the doGraham method into a flow graph.
2. Second Activity (around 5 minutes): Team up with one other member of your tutorial group, swap flow graphs and check them to see they agree, resolve any differences and decide on an agreed flowgraph for the rest of the activities.

3. Third Activity (around 15 minutes): Split the tutorial group up into three subgroups and construct test sets for your flowgraph that are adequate for the following criteria:
    a. Statement Coverage
    b. Branch Coverage
    c. Basic Condition Coverage.
4. Fourth Activity (around 10 minutes): Rotate your solutions around the groups (statement coverage tests go to basic condition coverage group; basic condition coverage goes to branch coverage; branch coverage goes to statement coverage group). Check if the other group's solution is correct. If you find any errors check with them and agree a resolution to the problem.
5. Fifth Activity (around 10 minutes): For the test set you have just checked can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set.
6. Sixth Activity (if there is any time left): As a group can you create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

Solutions

Test suites for various coverage criteria

Test suites for various coverage criteria		
Coverage criterion	**Test vectors: { [x, y]* }+**	**Expected results**
Statement	{ [0, 1], [1, 0], [2, 0] }	min = 2
Branch	{ [1, 1], [-1, -1] }	min = 1
	{ [-1, -1], [2, -1] }	min = 1
Basic condition	{ [6, 5], [4, 4], [5, 4] }	min = 2

The branch coverage test suite comprises two successive calls to doGraham with different test vectors. It's also worth noting that all three of the above test suites actually satisfy all three criteria. This is a strong argument in favour of developing your tests incrementally:

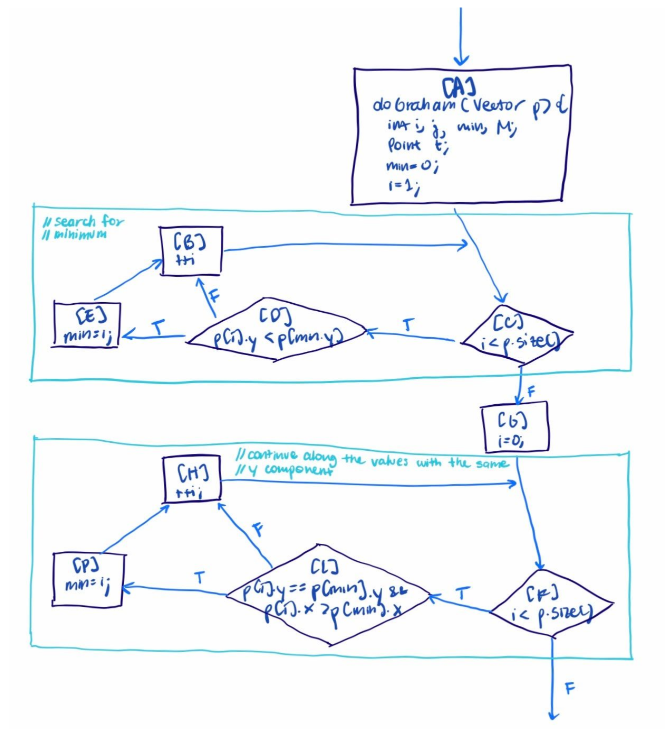
- Start with something simple
- See if it satisfies the adequacy criterion that you're aiming for
- If it doesn't, then observe what statements/branches/conditions were missed and target them next

Activity five suggests that you take these test suites and generate three mutants of the original code (one for each coverage criterion) which are not killed by their respective suite. You need to be very careful here that you create mutants which aren't equivalent to the original program. It's best in this situation to also try to come up with a test that shows that your mutant does break the code (i.e. a test that "kills" the mutant, to use mutation testing terminology — more about that in Lecture 9…).

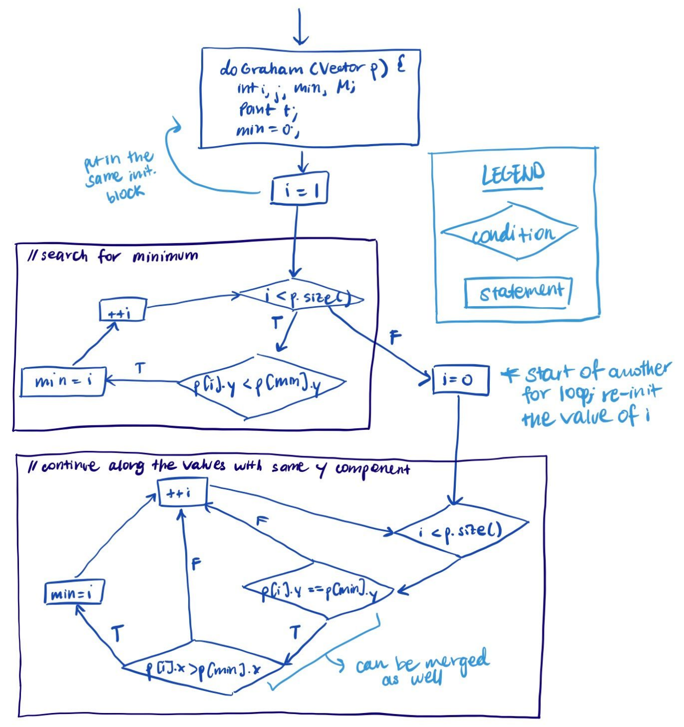
The thing to draw from this activity is that a "perfect" test suite should reject (i.e. fail) all programs which aren't correct. In practice though, they don't. Clearly here, your tests can obtain 100% branch coverage (and other measures) and still not detect many broken versions of the correct program.

Control Flow Graphs

Here are two control flow graphs for the doGraham program:



This one is useful for developing statement/branch coverage



This one breaks the compound condition in the second loop apart, and will be helpful for developing condition-based coverage since it makes Java's short-circuiting behaviour clearer (node L becomes two nodes, L and M)

Note how for loops are handled in both graphs: the initialiser (i = 1 or i = 0) is executed before the loop; then the loop condition is evaluated; then the loop body is executed; and finally the counting expression (++i) is executed before returning to the top of the loop. Also note how in nodes D, L, and M, a shorthand was used for the rather long-winded Java expressions in the source. The annotations on the graph are there to make it clear to you what's happening, so you don't need a literal copy of the entire source code - but something which makes it clear what's happening will be helpful.

Statement coverage

Statement-adequate test suite	
Test vector	Expected result
{ [0,1], [1,0], [2,0] }	min = 2

A good place to start with this kind of problem is loops - modify the initialisation, termination condition or increment in order to iterate too many or too few times. So changing i =1 to i = 0 on line 8 looks like a good start:

```
// MUTANT SC1
    // search for minimum
    for (i = 0; i < p.size(); ++i) {
        if ( ((Point) p.get(i)).y < ((Point) p.get(min)).y ) {
            min = i;
        }
    }
```

This mutant is not killed by any of the tests, so it looks promising. Unfortunately however, **it can't be killed by any test at all**, or at least not any test which just pays attention to the program's output based on valid input vectors: min starts off at 0, so Mutant SC1 just compares the first vector element with itself. Unnecessary, but harmless. You couldn't even detect this mutant by (say) supplying a vector with a single null element, as while Mutant SC1 would throw an exception at line 9, Original would also throw an exception — just this time at line 18. To identify this mutant, your test harness would have to be counting the number of accesses to p.get(), or something similar. There are circumstances in testing a system where you might do this, but all we're doing is looking at the final value of min, so for our purposes **Mutant SC1 is equivalent to original**!

All of the above tests result in min >= 1, so changin i = 0 to i = 1 on line 17 looks like a good next try:

```
// MUTANT SC2
    // continue along the values with same y component
    for (i = 1; i < p.size(); ++i) {
        if ( ( ((Point) p.get(i)).y == ((Point) p.get(min)).y ) &&
                ( ((Point) p.get(i)).x > ((Point) p.get(min)).x ) ) {
                    min = i;
                }
    }
```

However this is also an unfortunate choice! The first loop finds the first point in the vector with minimum y value, and the second loop is there in order to search through all points with this minimum y and find the one which has the maximum x value. So the first comparison made in the loop (with i = 0) is actually always unnecessary - in fact the second loop could start with i = min+1 since min will always be the first point with the minimum y, and the second loop should only be looking at subsequent points min+1, min+2, … so **Mutant SC2 is equivalent to original** too!

One of the unfortunate aspects of doing a course in Software Testing is that we spend a depressing amount of time looking at really awful code: in this case for example, it's trivial to achieve what these two loops do in a single loop. It's not really awful, but it could be simultaneously clearer, more concise and more efficient.

This is one of the big problems with mutation testing: a huge number of mutants are actually equivalent to the original code. The change they involve has no effect on program execution. This is a waste, and in many cases unfortunately unavoidable: think how smart a mutant generator would have had to have been in order to recognise equivalence of the above three programs.

Let's get us a proper distinct mutant. Let's try i = 2.

```
// MUTANT SC3
    // continue along the values with same y component
    for (i = 2; i < p.size(); ++i) {
        if ( ( ((Point) p.get(i)).y == ((Point) p.get(min)).y ) &&
                ( ((Point) p.get(i)).x > ((Point) p.get(min)).x ) ) {
                    min = i;
                }
    }
```

The mutant still gives min = 2 on our original statement coverage test suite. In order to kill it now we need to add a test whose min is less than 2 and is found by the second loop. This, for example, would do:

Mutant SC3-killing test		
Test vector	**Expected result**	**SC3 result**
{ [0,1], [1,1] }	min = 1	min = 0

Note that the new test on its own doesn't achieve statement coverage (it never executes line 12), so it's worth keeping the old test around:

Mutant SC3-killing statement-adequate test suite		
Test vector	**Expected result**	**SC3 result**
{ [0,1], [1,0], [2,0] }	min = 2	
{ [0,1], [1,1] }	min = 1	min = 0

Branch coverage

Branch-adequate test suite	
Test vector	**Expected result**
{ [1,1], [-1,-1] }	min = 1
{ [-1,-1], [2,-1] }	min = 1

Here we observe that both of our tests give a result min = 1. So how about a mutant which gives a result of 1 a little more often than it should? Let's replace one of the min = i statements with min = 1. This kind of typo is particularly hard to spot, and a strong reason why variable names like i and l should be avoided:

```
// MUTANT BC1
    { min = 1; }
```

This mutant would be killed by any test whose result should be a min > 1, and where that point shares a y with an earlier point:

Mutant BC1-killing test		
Test vector	**Expected result**	**BC1 result**
{ [-1,-1], [2,-1], [3,-1] }	min = 2	min = 1

An alternate trick here would be to set min = 1 initially:

```
// MUTANT BC2
    Point t;
    min = 1;
```

This mutant will go wrong on vectors consisting only one point (with an out of bounds exception), and inputs where min should end up 0, but that point shares a y value with another - either of these tests will kill it:

Mutant BC2-killing tests		
Test vector	**Expected result**	**BC2 result**
{ [-1,-1] }	min = 0	Exception
{ [-1,-1], [2,-1] }	min = 0	min = 1

Finally, just to show that constants aren't the only good sources of mutants we could change the && on line 19 to ||:

```
// MUTANT BC3
        if ( ( ((Point) p.get(i)).y == ((Point) p.get(min)).y ) ||
                ( ((Point) p.get(i)).x > ((Point) p.get(min)).x ) ) {
                    min = i; }
```

This mutant will go wrong in all kinds of situations, but not our current test suite. Here's a killer:

Mutant BC3-killing test		
Test vector	**Expected result**	**BC3 result**
{ [1,3], [0,3] }	min = 0	min = 1

Don't forget to include enough of the original test suite that you still get branch adequacy:

Mutant BC1-killing branch-adequate test suite		
Test vector	**Expected result**	**BC1 result**
{ [1,1], [-1,-1] }	min = 1	
{ [-1,-1], [2,-1], [3,-1] }	min = 2	min = 1

Mutant BC2-killing branch-adequate test suite		
Test vector	**Expected result**	**BC2 result**
{ [1,1], [-1,-1] }	min = 1	
{ [1,1], [2,-1] }	min = 1	
{ [-1,-1] }	min = 0	min = 1

Mutant BC3-killing branch-adequate test suite		
Test vector	**Expected result**	**BC3 result**
{ [1,1], [-1,-1] }	min = 1	
{ [1,1], [2,-1] }	min = 1	
{ [1,3], [0,3] }	min = 0	min = 1

Basic condition coverage

Basic condition-adequate test suite	
Test vector	**Expected result**
{ [6,5], [4,4], [5,4] }	min = 2

This test suite also fails to kill Mutant SC3, so we can re-use Mutant SC3 here, and use the same test from the statement-adequate suite to kill it:

Mutant SC3-killing basic condition-adequate test suite		
Test vector	**Expected result**	**BC1 result**
{ [6,5], [4,4], [5,4] }	min = 2	
{ [0,1], [1,1] }	min = 1	min = 0

Coda

Comprehensive test suite to achieve basic condition coverage:

Test suites for a various coverage criteria		
Coverage criterion	**Test vectors: { [x,y]* }**	**Expected results**
Basic condition	{ }	min = 0
	{ [0,0], [1,2] }	min = 0
	{ [1,2], [1,1] }	min = 1
	{ [5,0], [2,1], [6,0] }	min = 2

This is good in that it exercises the code more thoroughly (notice that loops are covered for a number of different iteration counts, including zero). However it's pretty scary when it comes to generating a mutant that can sneak past it. My suggestion here is based on the observation that the resulting values for min here are all quite small: if we change the type of the index variables from int to byte (in the first line of the method), then their maximum value becomes 127.

```
// Evil Mutant
Vector doGraham(Vector p) {
    byte i, j, min, M;
    ...
}
```

Supplying a test vector with more than 128 entries will expose this fault if the lowest-y/highest-x point is beyond index 127 in the vector.

Tutorial 3: Data Flow Testing

Question Specification

Consider the following program:

```
static int find (int list [], int n, int key) {
    // binary search of ordered list
    int lo = 0, mid;
    int hi = n - 1;
    int result = -1;
    while ((hi < lo) && (result == -1)) {
        mid = (lo + hi) / 2;
        if (list[mid] == key)
            result  = mid;
        else if (list[mid] > key)
            hi = mid - 1;
        else    // list[mid] < key
            lo = mid + 1;
    }
    return result;
}
```

This is not a particularly good example of programming but it is useful for the purposes of this tutorial.

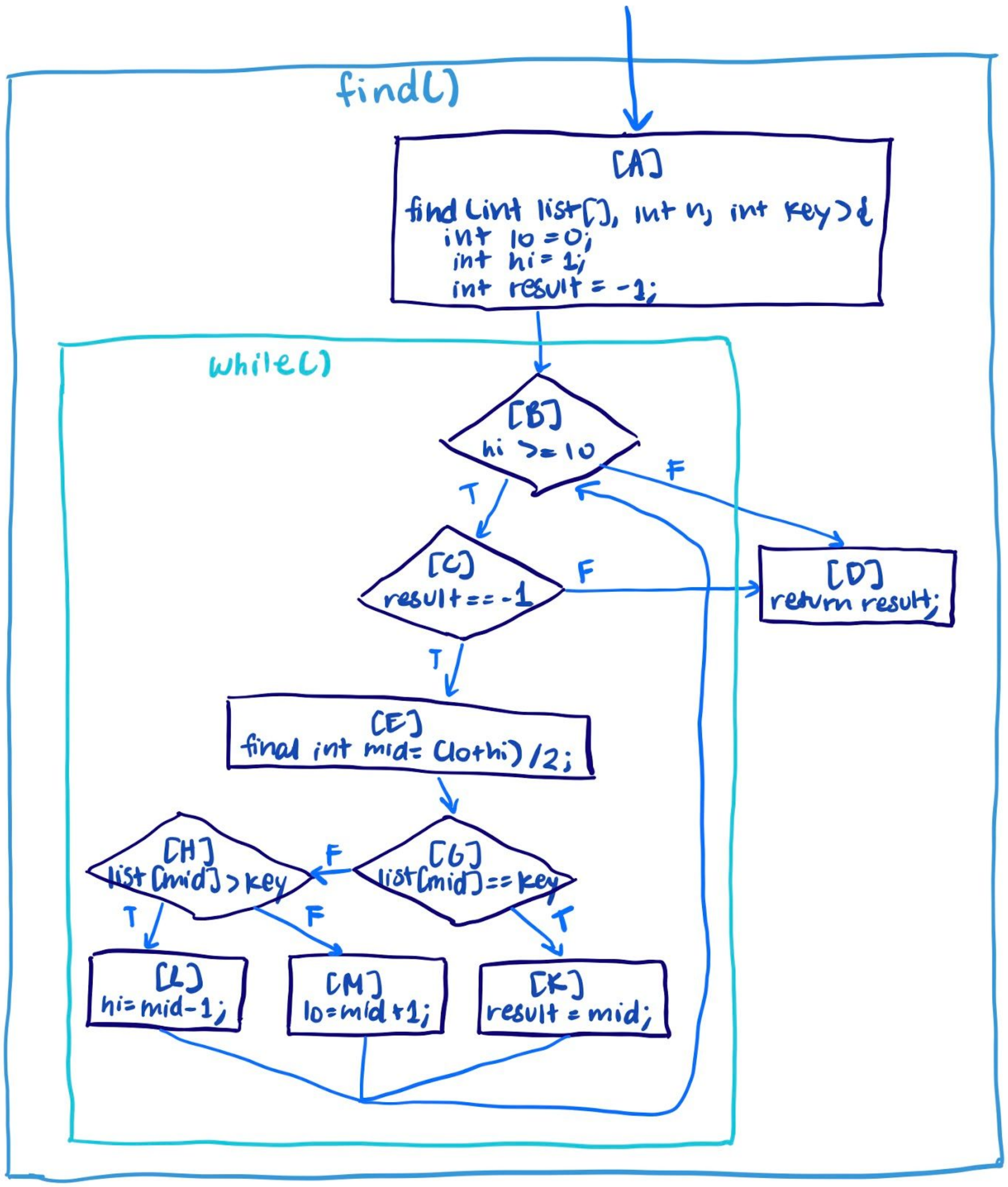
- Prerequisites: Review the material on Data-Flow based testing in Lectures 7 and 8 and the paper by Frankl and Weyuker.
- Preparation: Review the code above; please try to ensure you understand the method and the particular implementation. It is an implementation of binary search of an ordered array.

Activities

1. (10 Minutes) First individually construct the flow graph corresponding to this program.
2. (5 Minutes) Find a partner to work with in the group and check that you agree on the structure of the flow-graph for the program.
3. (10 minutes) For each variable, write down the < D, U > pairs.
4. (10 minutes) Write down tests that satisfy one of the following coverage criteria:
    a. All < D, U > pairs (b) All < D, U > paths
5. (10 minutes) As a whole class, compare the tests sets devised for the two coverage criteria and discuss which is stronger. Can you think of a test that passes one of the two criteria but fails the other?

Solutions

Control Flow Graph



The two basic conditions have been broken down in the while loop out into separate nodes. This makes it easier to track the behaviour of short-circuiting. You don't need to do this in diagrams for statement or branch coverage, but it helps when you're thinking about the different types of condition coverage, as well as data flow.

Definitions, and DU pairs
Here's a table presenting all the variables, where they are defined and where they're used.

Variable	Definitions	DU pairs
list[]	A (parameter)	AH, AG
n	A (parameter)	AA
key	A (parameter)	AH, AG
mid	E	EH, EG, EL, EM, EK
lo	A, M	AB, AE, MB, ME
hi	A, L	AB, AE, LB, LE
result	A, K	AC, AD, KC, KD

Paths for "all DU pairs", "all DU paths"
To cover "all DU pairs", the test suite must include a def-clear path from each variable definition to all uses of that variable that are reachable from the definition. To cover "all DU paths", we're looking again at covering all DU pairs def-use pairs, but this time we don't just want one path for each pair, we want all paths between each pair, modulo loop iterations.

Test for "all DU pairs"
As with previous coverage criteria, let's start with a simple test and see how well it satisfies the "all DU pairs" criterion:

```
list[] = {0,3,5,8,8,17}
n = 6
key = 8
```

Executing this, we start off on the first iteration with and `lo = 0` and `hi = 5`. `mid` will be `2` (since Java rounds ints down), and since `list[2] = 5` (`< key`), we'll execute node M and start the next iteration with `lo = 3` and `hi = 5`. `mid` will now be `4`, and since (`== key`), we'll set `result` in node K, and execution will terminate via node C on the next iteration. So our execution path will have been as follows:

A BCEGHM BCEGK BCD

We can the cross off DU pairs covered in our table (italicised):

Variable	Definitions	DU pairs
list[]	A (parameter)	*A→H, A→G*
n	A (parameter)	*A→A*
key	A (parameter)	*A→H, A→G*
mid	E	*E→H, E→G*, E→L, *E→M, E→K*

lo	A, M	*A→B, A→E, M→B, M→E*
hi	A, L	*A→B, A→E,* L→B, L→E
result	A, K	*A→C,* A→D, *K→C, K→D*

Note that because lo, hi and result can be defined in more than one place, we have to be careful that the paths we're looking at are def-clear paths, e.g. for result's A→C path, there must be no instance of node K redefining result between A and C.

Looking at the left-over DU pairs here, we need to exercise the DU pairs involving L and also exercise the AD pair which can only occur when the list is empty. To exercise the DU pairs involving node L, we can add the following test:

```
list[] = {0,3,5,8,8,17}
n = 6
key = 3
```

The execution path will be (`lo = 0, hi = 5, mid = 2; lo = 0, hi = 1, mid = 0; lo = 1, hi = 1, mid = 1; result = 1` in node K, and execution will terminate via node C on the next iteration):

A BCEGHL BCEGHM BCEGK BCD

This enables us to cross off the following remaining DU pairs: L→B, L→E, E→L. The only uncovered DU pair is A→D for result. This can be covered with a test that provides an empty input array:

```
list[] = {}
n = 0
key = 1
```

Execution will terminate straight off through the path A BD. Our final test suite to satisfy all DU pairs is as follows:

```
// Test 1
list[] = {0,3,5,8,8,17}
n = 6
key = 8

// Test 2
list[] = {0,3,5,8,8,17}
n = 6
key = 3

// Test 3
list[] = {}
n = 0
key = 1
```

Test for “all DU paths”

Start with the above all DU pairs test suite.

Since all DU paths require that the def-clear paths not include more than one iteration of any loops, we need to take care that we've not used any def-clear paths which cover more than one iteration of the main loop. This would mean, from Test 1, that we can't have any DU pairs running from the opening A to anything in the closing BCD (because two loop iterations intervene); from Test 2, that we can't have any DU pairs running from the A to the closing BCEGHL BD, or from the first loop iteration BCEGHL to the closing BD. And there's a problem there: we ticked off result's A→D path in Test 2. That's got three iterations of the loop in the middle, so we need to get it some other way. But in Test 3, we have execution path A BD — which includes A→D without an intervening K or any loop iterations, just as we'd like. So the above test doesn't break our “not more than one loop” rule, but does it contain all possible paths?

The only DU pairs between which there are more than one loop-free path are the pairs A→D and K→D for variable result: with both of these, the D can be reached either by BD or BCD. Are these covered by our current test suite?

- Test 2 covers A→BCD but there are too many loop iterations in between the pair; Test 3 covers A→BD again, but with no intervening loops.
- Going back to our impossible pairs notes above, A→BCD is impossible: you can't have a def-clear path for result from setting result = -1 to a decision in which result == -1 is false.
- What about K→BD? This too is impossible: if we hit node K, we know that result is no longer -1, so we will definitely exit the loop via CD on the next iteration; in order to exit via BD before that, hi >= lo would have to be false — however, it was true on the previous iteration of the loop, and since we took the path BCEG to node K, we know that lo and hi haven't changed since we last compared them. So it will still be true in the next iteration, and we'll exit via CD.
- Test 1 covers K→BCD (but nothing for A since the presence of K means that we don't have a def-clear path from A to D). So actually our existing test suite already covers all of the DU paths that it's possible to cover.

Test that satisfies “all DU pairs”, but fails “all DU paths”
This is a little tricky, since we’ve established above that there’s no DU pair between which there exists more than one feasible path. Consequently the only approach to this would be to break the “no more than one loop” rule. For example, we could replace Test 3 with something which is good enough for all uses, but involves lots of loop iterations so it won’t satisfy all DU paths.

Here’s a new suite doing just that: Test 3 gets replaced with two new tests, one of which gets us A→BD for lo, and the other for hi (noting that for lo we need to avoid the def in node M, and for hi we need to avoid the def in node L):

	list[]	**n**	**key**	**Path**
Test 1	{0, 3, 5, 8, 8, 17}	6	8	A BCEGHM BCEGK BCD
Test 2	{0, 3, 5, 8, 8, 17}	6	3	A BCEGHL BCEGHM BCEGK BCD
Test 3	{0, 3, 5, 8, 8, 17}	6	-1	A BCEGHL BCEGHL BD
Test 4	{0, 3, 5, 8, 8, 17}	6	18	A BCEGHM BCEGHM BCEGHM BD

Tutorial 4: Mutation Testing

Question Specification

Consider the following program:

```
public int Segment(int t[], int l, int u) {
    // Assumes t is in ascending order, and l < u,
    // counts the length of the segment
    // of t with each element l < t[i] < u

    int k = 0;

    for (int i = 0; i < t.length && t[i] < u; i++) {
        if (t[i] > 1) { k++; }
    }

    return k;
}
```

This is not a particularly good example of programming but it is useful for the purposes of this tutorial.

- Prerequisites: Review the lecture on Mutation Testing and the papers by Offutt et al included in the required readings.
- Preparation: Review the code above; please try to ensure you understand the method and the implementation. The program assumes the array t is in ascending order and given two integers l and u it finds the length k of the sequence t[j], t[j + 1],... t[j +k−1] with:
    - `((j = 0 or t[j - 1] <= l) and l < t[j]) and ((t[j + k - 1] < u)` and
    - `(j + k = t.length or u <= t[j + k])` so k is the length of the longest subsequence of t with all elements greater than l and smaller than u.

Activities

1. (10 Minutes) In your groups, construct a test suite for this method. Each test case in the suite needs to specify the size of t, the values of the elements of t, the values l and u, and the expected result of the test. Write your test suite down on a separate sheet of paper. Use a simple coverage criterion to judge the adequacy of your test suite. You might just want to use statement coverage. The choice is yours.
2. (10 Minutes) In your groups, construct three or four mutants of the above program. Each mutant should be derived from the original program using one application of one of the following rules (i.e. each mutant contains only one mutation): (a) Changing constants: a constant c in the program may be replaced by c + 1 or c − 1. (b) One-off in relations: < may be replaced by <= and vice versa; > may be replaced by >= and vice versa. (c) Inverted relations: < or <= may be replaced by > or >= and vice versa. (d) Duplicate statements: any statement s; can be replaced by s;s; (e) Deleted statements: you may delete any statement.
3. (15 Minutes) Pass your test suite to a neighbouring group. With the test suite you have just received: for each of your mutants work out whether or not it is killed by the test suite.

4. (10 Minutes) If there is at least one mutant which is not killed in at least one of the groups do the following: (a) Get the tutor to write the unkilled mutants on the board. (b) In your groups check if any of the mutants on the board is killed by the test set you used in part 3. (c) Remove any mutant that is killed by at least one test suite. (d) If the board is empty of mutants go to stage 5, otherwise go to stage 6.
5. (10 Minutes) If there are no mutants which are unkilled by all the test suites so the following: (a) Get the tutor to write the test suites on the board. (b) In your groups, attempt to create a mutant that is not killed by any of the test suites on the board. (c) If you can create such a mutant write it on the board. If you cannot create such a mutant try to invent other mutation rules that let you create such a mutant.
6. (10 Minutes) To get here you must have a mutant on the board that has evaded death by test suite. In groups, try to strengthen your test suite until it kills the mutant on the board. Write the strengthened test suites on the board.
7. If there is time discuss how effective you think Mutation testing is in strengthening test suites.

Solutions

Test suites

Test suite	Test no.	t[]	l	u	Expected	Statement coverage	Branch coverage	MC/DC	All-uses coverage
0	1	{1, 2, 3, 4, 5}	1	5	3	YES	YES	YES	YES
	2	{ }	0	1	0				
	3	{1, 2}	0	1	0				
1	1	{ }	1	2	0	YES	YES	YES	NO
	2	{1, 2}	1	2	0				
	3	{0, 1, 2}	0	2	1				
2	1	{5}	2	4	0	YES	NO	NO	YES
	2	{3}	2	4	1				
	3	{3, 4, 5}	2	6	3				
3	1	{ }	1	10	0	YES	YES	YES	NO
	2	{3, 4, 5}	1	2	0				
	3	{2, 4, 6}	3	5	1				
4	1	{ }	1	10	0	YES	NO	NO	NO
	2	{0}	0	0	0				
	3	{0}	-1	1	1				
5	1	{3, 4, 5, 6}	4	8	2	YES	YES	NO	YES
	2	{ }	4	8	0				
	3	{9}	9	10	0				
6	1	{ }	1	1	0	NO	NO	NO	NO
	2	{2}	0	0	0				
	3	{2}	3	3	0				
7	1	{ }	1	1	0	YES	YES	YES	YES
	2	{1, 2, 3, 4}	1	4	2				
	3	{5, 6, 7, 8}	1	2	0				
8	1	{0, 3, 5}	0	5	1	YES	YES	YES	NO
	2	{0, 1, 2, 3}	1	3	1				
	3	{ }	0	1	0				

Mutants
Here are ten sample mutants:

```
// Mutant M0: change i=0 to i=1
for (int i = 1; i < t.length && t[i] < u; i++)

// Mutant M1: change < to <= in i < t.length
for (int i = 0; i <= t.length && t[i] < u; i++)

// Mutant M2: change < to <= in t[i] < u
for (int i = 0; i < t.length && t[i] <= u; i++)

// Mutant M3: change > to >= in if statement
    if (t[i] >= l)

// Mutant M4: change < to > in i < t.length
for (int i = 0; i > t.length && t[i] < u; i++)

// Mutant M5: change < to > in t[i] < u
for (int i = 0; i < t.length && t[i] > u; i++)

// Mutant M6: change > to < in if statement
    if (t[i] < l)

// Mutant M7: statement duplication
        { k++; k++ }

// Mutant M8: change types
byte k = 0;

// Mutant M9: change expressions
    k = 1 << k;

// Mutant M10: statement duplication
    if (t[i] > l)   if(t[i] > l)
```

Note that the last two break the rules set out in Activity 2: they're from when we got desperate after all our mutants got killed by well-written test suites.

- Mutant M8 takes advantage of the fact that test suites often focus on small number; using a byte will store counts up to 127, but any test which is supposed to return a larger number will fail.
- Mutant M9 is also trying to trick small test suites: using k = 1 << k; will generate the sequence of values 0, 1, 2, 4, 8 … for k instead of 0, 1, 2, 3, 4, … this mutant will survive any test suite which doesn't contain a test with a return value larger than 2.

Mutation adequacy
Below I will show how each test suite does against each of the mutants (Y indicates that the test suite kills the mutant, while a N indicates that the mutant survived the test suite). In the final column I record each test suite's mutation adequacy as a percentage.

Note that Mutant M10 survives all of the test suites, and on closer examination, unlike M8, it turns out to be an equivalent mutant (duplicating the if statement is like saying if(A && A) - exactly the same as if(A), so long as A has no side effects). Consequently M10 is not counted when computing mutation adequacy.

The formula for mutation adequacy is as follows:

$$mutation\ adequacy\ = \frac{no.\ of\ killed\ mutants}{total\ no.\ of\ mutants - no.\ of\ equivalent\ mutants}$$

Test suite	Statement Coverage	Branch coverage	MC/DC	All-uses coverage	M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	Mutation adequacy
0	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y	N	80%
1	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	70%
2	Y	N	N	Y	Y	Y	N	N	Y	Y	Y	Y	N	Y	N	70%
3	Y	Y	Y	N	N	Y	N	N	Y	Y	N	Y	N	N	N	40%
4	Y	N	N	N	Y	Y	N	N	Y	Y	Y	Y	N	N	N	60%
5	Y	Y	N	Y	N	Y	N	Y	Y	Y	Y	Y	N	N	N	60%
6	N	N	N	N	N	Y	N	N	N	Y	Y	N	N	N	N	30%
7	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	70%
8	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	70%

It's interesting to see that while the suite with the worst mutation adequacy has the worst coverage and the suite with the best mutation adequacy has full coverage, there's no simple correlation between test suite coverage and mutation adequacy in between these two.