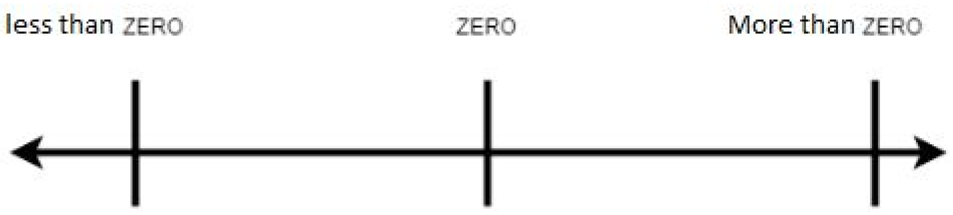
CODE AT
https://github.com/kasperpagh/TestAssignment3/tree/master/src/cphbusiness/pagh

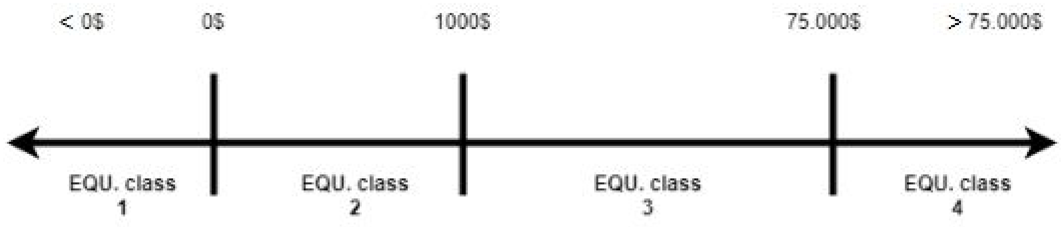
Equivalence classes

1) I have identified the following equivalence classes for the problem ***boolean isEven(int n)***:
    - Even number less than zero
    - Even number above zero
    - Zero
    - An odd number less than zero
    - An odd number above zero



Testcase #	Input	Expected output	Actual output
#1	-2	**true**	
#2	2	**true**	
#3	0	**true**	
#4	-5	**false**	
#5	5	**false**	

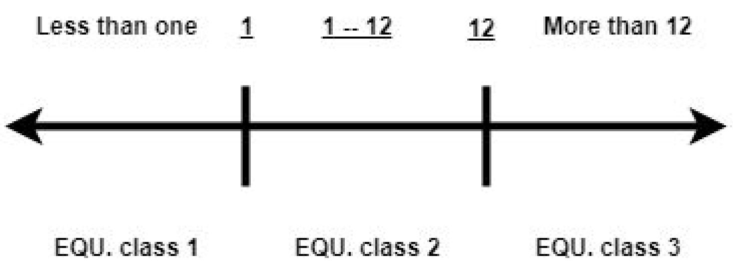
2) I have identified the following equivalence classes for the problem concerning the valid wage for mortgage application.



Testcase #	Input	Expected output	Actual output
#1	-2$	**false**	
#2	500$	**false**	

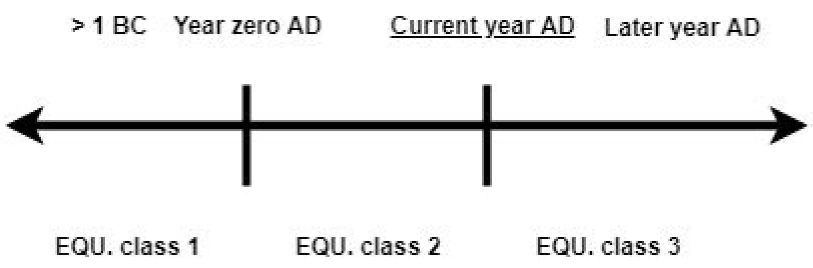
#3	35.000$	**true**	
#4	100.000$	**false**	

3) Equivalence classes for month and year. First the one for **month**.



Testcase #	Input	Expected output	Actual output
#1	-42	**ERROR**	
#2	0	**ERROR**	
#3	6	**true**	
#4	24	**ERROR**	

Then the one for year, normally i would think that it would be reasonable to limit this function to AD (0-cur. date). But since the assignment doesn't say so, i have decided to include BC- thus leaving us with the following partitions. (NB. i assume BC years are given as negative integers, **such that year 4 BC is: -4**). This however results in pretty boring partitions since all values are valid



Testcase #	Input	Expected output	Actual output
#1	-42 BC	**true**	
#2	543 AD	**true**	
#3	6.500 AD	**true**	

Boundary analysis

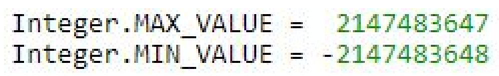
1) Boundary for ***boolean isEven(int n)***:

My general strategy for boundaries is to first identify the boundaries themselves, and then test the following:

- One value just below the boundary.
- The boundary itself (for example: zero, Integer.MAX_VALUE etc)
- One value just above the boundary

In this case however the boundaries are very open, since every conceivable number is either even or odd. The obvious boundaries are therefore going to be Integer.MAX_VALUE and MIN_LENGTH and zero (which arguably is not that interesting in this case).

Reminder:



```
Integer.MAX_VALUE =  2147483647
Integer.MIN_VALUE = -2147483648
```

Testcase #	Input	Expected output
#1	(Integer.MAX_VALUE -1)	**true**
#2	**Integer.MAX_VALUE**	**false**
#3	(Integer.MAX_VALUE +1)	**true**
#4	(Integer.MIN_VALUE -1)	**false**
#5	**Integer.MIN_VALUE**	**true**
#6	(Integer.MIN_VALUE +1)	**false**
#7	-1	**false**
#8	**0**	**true**
#9	1	**false**

2) The only interesting boundaries for this problem is 1.000$ and 75.000$

Testcase #	Input	Expected output
#1	-1$	**false**
#2	0$	**false**
#3	1$	**false**

#4	999$	**false**
#5	**1.000$**	**true**
#6	1.001$	**true**
#7	74.999$	**false**
#8	**75.000$**	**true**
#9	75.001$	**false**

3) Here the interesting boundaries are as follows: month 1 (lowest valid val.) and 12 (highest valid val.). Year 0 AD and current year AD is the interesting values in regard to the year.

Testcase # (MONTH)	Input	Expected output
#1	0	**ERROR**
#2	**1 (jan)**	**true**
#3	2 (feb)	**true**
#4	11 (nov)	**true**
#5	**12 (dec)**	**true**
#6	13	**ERROR**

Testcase # (YEAR)	Input	Expected output
#1	-1 BC	**true**
#2	**0 AD**	**true**
#3	1 AD	**true**
#4	2017 AD	**true**
#5	**2018 AD (current)**	**true**
#6	2019 AD	**true**

Decision tables

1) Only rule 1 and rule 2 results in refunded payment.

CONDITIONS	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**
Doctor visit	True	false	false	true	false	true	True	false
Hospital visit	false	true	false	true	false	true	false	true
Deductible paid in full	True	true	true	true	false	false	false	false
ACTION								
Refunded 0%	false	false	false	false	false	false	**True**	**true**
Refunded 50%	**True**	false	false	false	false	false	false	false
Refunded 80%	false	**true**	false	false	false	false	false	false
ERROR message	false	false	**True**	**True**	**True**	**True**	false	false

2)

CONDITIONS	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**	**Rule 9**
Divisible by 4	True	false	true	false	false	true	false	false	true
Divisible by 100	True	false	false	true	false	true	true	true	false
Divisible by 400	True	false	false	false	true	false	true	false	true
ACTION									
is leap year?	**true**	false	false	false	false	false	**true**	false	**true**

State transition

1) To draw the state diagram for the *ArrayListWithBugs* i start by finding the various states, actions and events.

Events:
The events in this case, is the same as the methods of the class, seeing as the only way for an object to "do anything" is through its methods. The only exception to this is is the method ***int size()*** since calling it doesn't result in a change of the objects state, but simply informs the users.

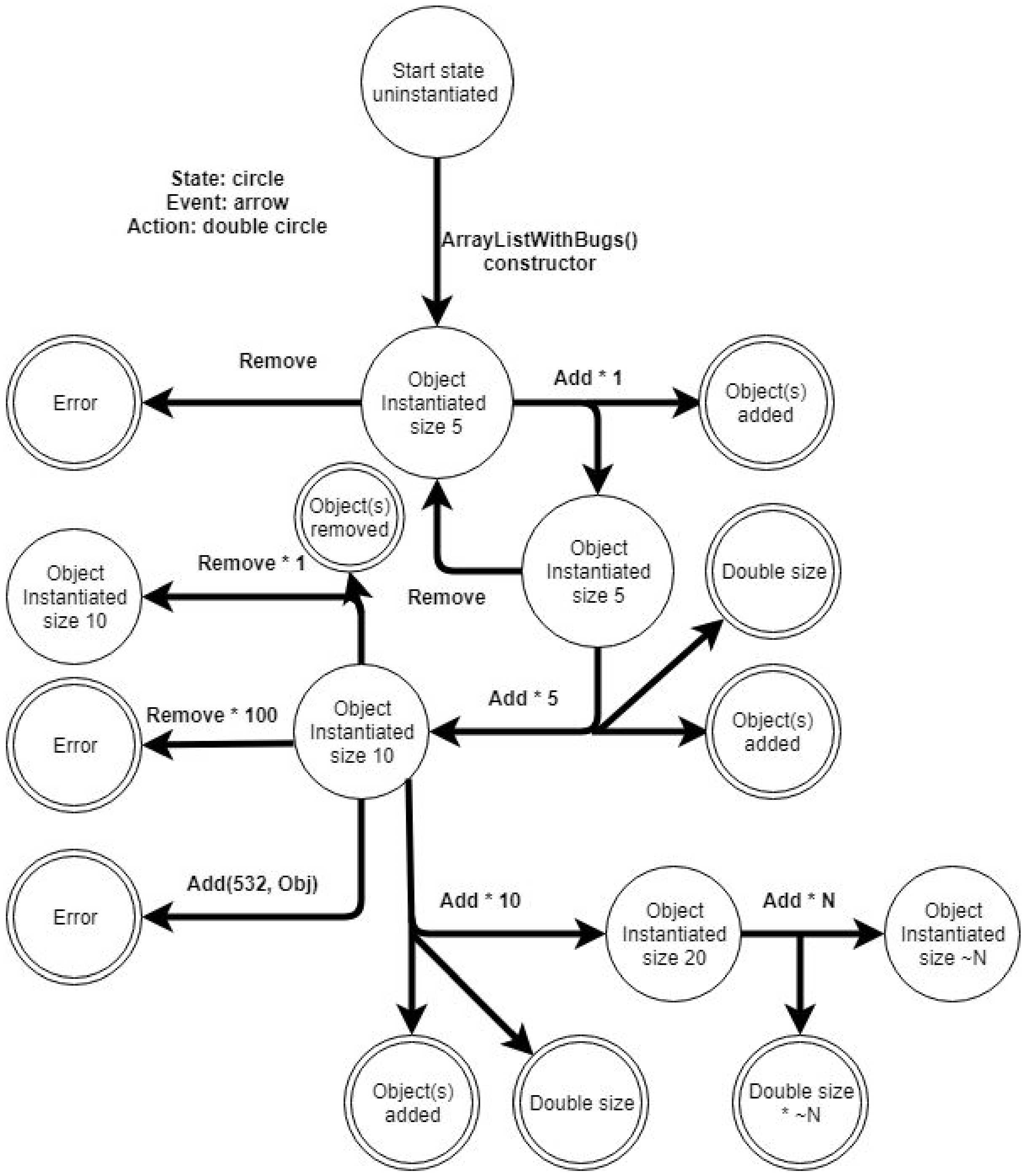
States:
The best measure for the states of an ArrayListWithBugs is, in my opinion, its size and whether it has yet been instantiated.

- **uninstantiated** (ie. constructor has not yet been called)
- **Instantiated** (constructor has been called, but nothing else. The size is 5)
- Whenever the method ***getLongList()*** is called (in the *add()* methods), i would argue that the state changes (since the memory consumption is larger etc). I will denote these states by their size (ie. 5, 10, 20 etc).
- **Empty** this state differs from Instantiated, since the list can be empty, but have a size of 40 (or some other value depending on how many times ***getLongList()*** has been called).

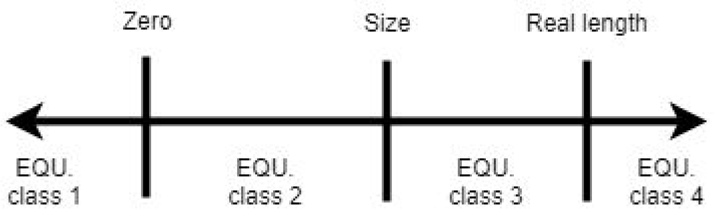
Actions:

- Object is added to the list.
- Object is deleted from the list.
- The list doubles in size.
- Error msg. (IndexOutOfBounds) is thrown

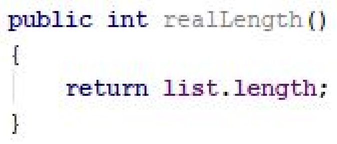
Diagram on next page


Start state uninstantiated
State: circle
Event: arrow
Action: double circle
ArrayListWithBugs() constructor
Remove
Error
Object Instantiated size 5
Add * 1
Object(s) added
Object(s) removed
Object Instantiated size 5
Remove * 1
Object Instantiated size 10
Remove
Double size
Add * 5
Remove * 100
Error
Object Instantiated size 10
Object(s) added
Error
Add(532, Obj)
Add * 10
Object Instantiated size 20
Add * N
Object Instantiated size ~N
Object(s) added
Double size
Double size * ~N

2) Testcases for the ***MyArrayListWithBugs*** class. The first thing to do is identify boundary values and equivalence classes for the indices (relevant for ***get(index), add(index, obj), remove(index)***).



Due to the above these test cases are for both get, add and remove.
(NB. I have introduced another method (***realLength()***, below) to allow for testing the state of the list. I know that normally introducing code JUST for the purpose of testing is very bad practice - but i fail to see how to determine the state of the list otherwise).



```
public int realLength()
{
    return list.length;
}
```

FOR ADD METHOD INDEX FOR EMPTY LIST

Testcase #	Input	Expected output	Actual output
#1	-1	ERROR	
#2	0	element is added	
#3	1	ERROR	
#4	size()-1	ERROR	
#5	size()	element is added	
#6	size()+1	ERROR	
#7	realLength()-1	ERROR	
#8	realLength()	ERROR	
#9	realLength()+1	ERROR	
#10	Integer.MAX_VALUE-1	ERROR	
#11	Integer.MAX_VALUE	ERROR	
#12	Integer.MAX_VALUE+1	ERROR	
#13	Integer.MIN_VALUE-1	ERROR	
#14	Integer.MIN_VALUE	ERROR	
#15	Integer.MIN_VALUE+1	ERROR	

#16	**-5038**	**ERROR**	
#17	**Math.floor(size()/2)**	**element is added**	
#18			
#19	**realLength()+2**	**ERROR**	

FOR REMOVE METHOD INDEX FOR LIST OF SIZE SIX(6)

Testcase #	Input	Expected output	Actual output
#20	**-1**	**ERROR**	
#21	**0**	**element is removed**	
#22	**1**	**element is removed**	
#23	**size()-1**	**element is removed**	
#24	**size()**	**ERROR**	
#25	**size()+1**	**ERROR**	
#26	**realLength()-1**	**ERROR**	
#27	**realLength()**	**ERROR**	
#28	**realLength()+1**	**ERROR**	
#29	**Integer.MAX_VALUE-1**	**ERROR**	
#30	**Integer.MAX_VALUE**	**ERROR**	
#31	**Integer.MAX_VALUE+1**	**ERROR**	
#32	**Integer.MIN_VALUE-1**	**ERROR**	
#33	**Integer.MIN_VALUE**	**ERROR**	
#34	**Integer.MIN_VALUE+1**	**ERROR**	
#35	**-5038**	**ERROR**	
#36	**Math.floor(size()/2)**	**element is removed**	
#38	**realLength()+2**	**ERROR**	

FOR GET METHOD INDEX FOR LIST OF SIZE 3 (three)

Testcase #	Input	Expected output	Actual output
#39	-1	ERROR	
#40	0	element is retrieved	
#41	1	element is retrieved	
#42	size()-1	element is retrieved	
#43	size()	ERROR	
#44	size()+1	ERROR	
#45	realLength()-1	ERROR	
#46	realLength()	ERROR	
#47	realLength()+1	ERROR	
#48	Integer.MAX_VALUE-1	ERROR	
#49	Integer.MAX_VALUE	ERROR	
#50	Integer.MAX_VALUE+1	ERROR	
#51	Integer.MIN_VALUE-1	ERROR	
#52	Integer.MIN_VALUE	ERROR	
#53	Integer.MIN_VALUE+1	ERROR	
#54	-5038	ERROR	
#55	Math.floor(size()/2)	element is retrieved	
#57	realLength()+2	ERROR	

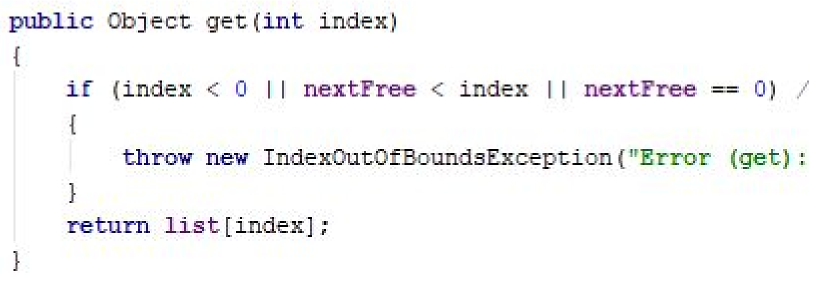
Further tests:

Testcase #	Input	Expected output	Actual output
#58	**1) Add object *new String("test")***	**size = 1**	
#59	**2) Get object**	**"test"**	
#60	**3) Remove object**	**size = 0**	
#61	**4) Add 6 objects**	**size = 6 AND realLength = 10**	
#62	**5) Remove 6 objects**	**size = 0 AND realLength = 10**	

Testcase #	Input	Expected output	Actual output
#63	**Remove one object from empty list**	**ERROR**	
#64	**2) Get object from empty list**	**ERROR**	

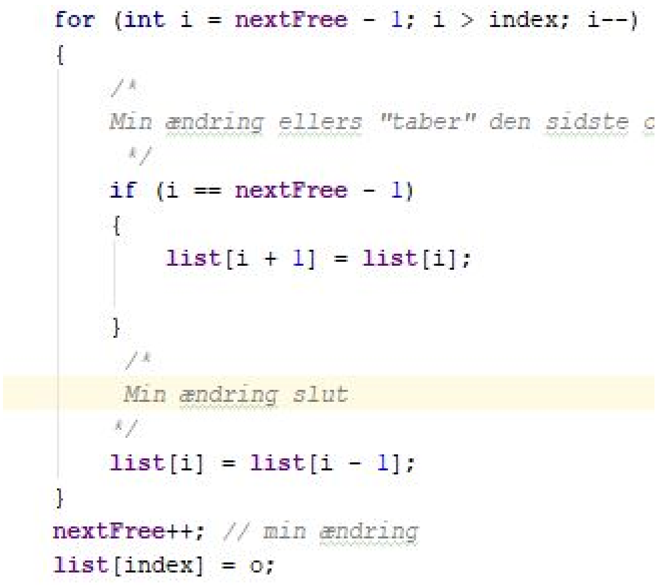
4) I have found the following errors in the MyArrayListWithBugs code:

Error in the *get(int index)* method, Which results in the the object stored at index 0 to be inaccessible be course an exception is always thrown. My solution is to change the less-than-or-equals-to operator to a less than, and add a special case for when the list is empty, such that any get call to the list will give an exception when it's empty.



```
public Object get(int index)
{
    if (index < 0 || nextFree < index || nextFree == 0) /
    {
        throw new IndexOutOfBoundsException("Error (get):
    }
    return list[index];
}
```

Error in the *add(int index, Object obj)* method:, which results in the last object in the list being being dropped every time the method is called. To fix this i introduced an if statement within the for loop to give special treatment to the last element.

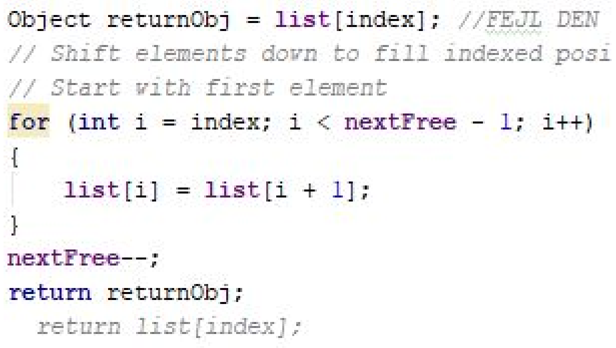


```
for (int i = nextFree - 1; i > index; i--)
{
    /*
    Min ændring ellers "taber" den sidste c
     */
    if (i == nextFree - 1)
    {
        list[i + 1] = list[i];

    }
     /*
     Min ændring slut
     */
    list[i] = list[i - 1];
}
nextFree++; // min ændring
list[index] = o;
```

The method was also didn't increment the nextFree variable, so i also did that.

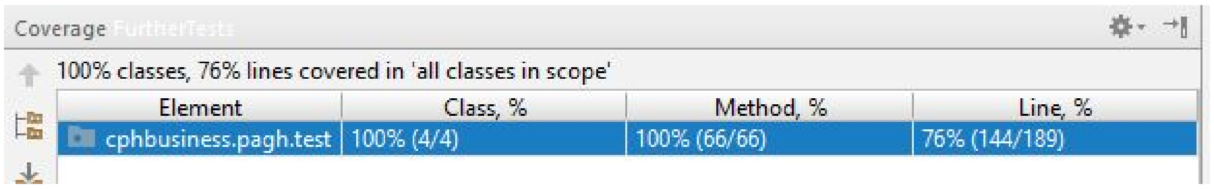
Error in the *remove(int index) method:* Which resulted in the wrong object being returned (it was intended to return the deleted object). I fixed this by storing an temporary copy of the object in question before the loop starts messing with the list's indices.



```
Object returnObj = list[index]; //FEJL DEN
// Shift elements down to fill indexed posi
// Start with first element
for (int i = index; i < nextFree - 1; i++)
{
    list[i] = list[i + 1];
}
nextFree--;
return returnObj;
  return list[index];
```

5) A state table would, in this case, probably have been a better tool than a state diagram. Not that there's anything wrong with the diagrams per se, but they are better suited to model smaller scale domains (such as the pin code example from your slides). If the thing you are trying to model is complex, then the diagram is going to be next to impossible to understand at a glance.

6) I have coverage of 100% of all methods and 76% of all LoCs.


Coverage FurtherTests

100% classes, 76% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
cphbusiness.pagh.test	100% (4/4)	100% (66/66)	76% (144/189)

I would argue this is an ok amount of coverage, but on the other hand it could also be argued that such a relatively short class, should have 100% coverage. So all in all i would call it sufficient coverage.

In regards to my states and transitions, i have tried to test them all working under the assumption that the states after list.size() == 5, is pretty equivalent to one another. We're I to drop that assumption there is an infinite number of states to explore which frankly would take more time than i have on hand!