TEST CASE DESIGN – OUTLINE SOLUTIONS

EQUIVALENCE PARTITIONING

SOLUTION 1

Inputs 2, 4, 6 … 1000 will yield true and 1, 3, 5 … 999 will yield false.

Valid equivalence class:	(even numbers): 2, 4, 6 … 1000
Valid equivalence class:	(uneven numbers): 1, 3, 5 … 999
Invalid equivalence classes:	number <= 0, number > 1000

SOLUTION 2

Valid equivalence class:	1000 > = income <= 75,000
Invalid equivalence classes:	income < 1000, income > 75,000

SOLUTION 3

Equivalences classes (month):

- Valid equivalence classes:
  - Months with 31 days
  - Months with 30 days
  - February with 28 days
  - February with 29 days
- Invalid equivalence classes:
  - Month < 1
  - Month> 12

Equivalences classes (year):

- Valid equivalence classes:
  - Leap years*
  - Non-leap years
- Invalid equivalence classes:
  - Year < 0
  - Year > 2^{31}-1

Equivalences classes (month + year):

- Months with 31 days, non-leap year
- Months with 31 days, leap year
- Months with 30 days, non-leap year
- Months with 30 days, leap year
- February, non-leap year
- February, leap year

BOUNDARY VALUE ANALYSIS

SOLUTION 1

0 (invalid)
1 (valid)
1000 (valid)

1001(invalid)

SOLUTION 2

999 (invalid)
1000 (valid)
75,000 (valid)
75,001 (invalid)

SOLUTION 3

Boundary values (month + year)

Leap years divisible by 400
Non-leap years divisible by 100, but not with 400
Non-positive invalid months (0) and Positive invalid months (13)

DECISION TABLES

SOLUTION 1

There are 4 rules.
The first condition (Is the deductible met?) has two possible outcomes, yes or no (could also be named true or false).
The second condition (type of visit) has two possible outcomes, Doctor's office visit (D) or Hospital visit (H):

Conditions				
Deductible met?	Y	Y	N	N
Type of visit	D	H	D	H
Actions				
50 % Reimburse	X			
80 % Reimburse		X		
No reimbursement			X	X

SOLUTION 2

Conditions								
Divisible by 4	Y	Y	Y	Y	N	N	N	N
Divisible by 100	Y	Y	N	N	Y	Y	N	N
Divisible by 400	Y	N	Y	N	Y	N	Y	N
Actions								
Leap Year	X			X				
Not Leap Year		X						X
Impossible			X		X	X	X	

STATE TRANSITION

SOLUTION 1

A state diagram for `MyArrayListWithBugs` which visualizes its possible **states** and relevant **events** (which might trigger transitions) would have the states "**empty**" and "**loaded**" and the events would be "**add**", "**remove**", "**get**", "**size**". If the list has a maximum limit (the exercise does not say, though so it is just for illustration), it could instead have three states "**empty**", "**loaded**" and "**full**" like the `Stack` example below[1].

The behavior of `Stack` suggests three states "empty", loaded" and "full" and the methods "pop" and "push" are modeled as events. The first model (figure 7.4) is ambiguously defined for "push" and "pop" in the "loaded" state. It is not clear when push and pop make a transition from loaded, to full or empty respectively.

A way to solve this ambiguity is to add a guard to the transitions. A **guard** is a predicate expression **[…]** associated with the event, which must evaluate to true in order for the transition to happen. This has been added to the `Stack` in figure 7.5. Instead of just `pop`, you write for instance `pop[n == 1]` meaning pop from a list with 1 element, or `pop[n > 1]` meaning pop from a list with more than one element.

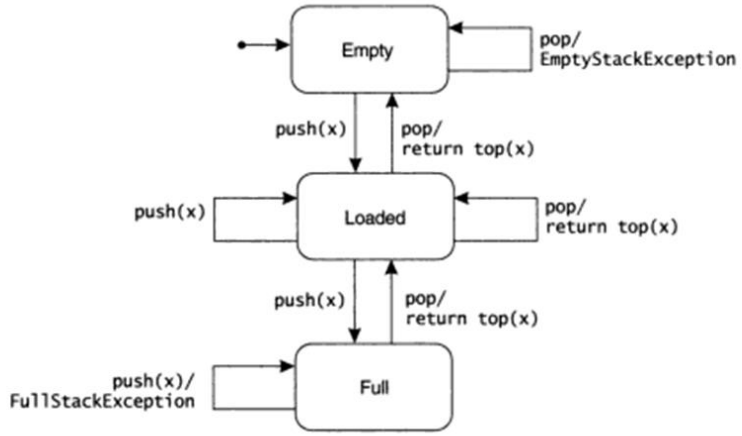

FIGURE 7.4 State machine model of Stack without guards.

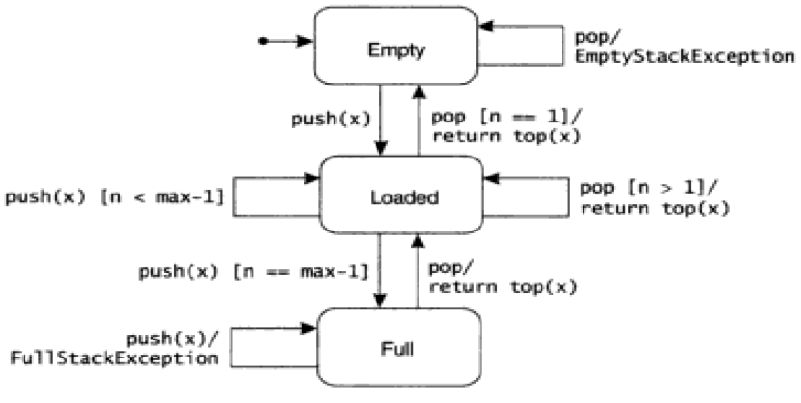

FIGURE 7.5 State machine model of Stack with guards.

[1] The `Stack` example is from Robert V. Binder: Testing Object-Oriented Systems, Models, Patterns, and Tools (2002).

SOLUTION 2-4

An example of test cases for code such as `MyArrayListWithBugs`'s method `Object remove(int index)` is shown below.

Notice that it is an assumption that that `size()` and `add()` have been tested and work!

For lists, it is a good idea to systematically test every operation with an empty list, a list with one element and a list with more than one element (for instance 5 elements).

Test case	Start condition	Input (index)	Expected output	Expected state after test	Result
empty list	size = 0	0	exception	size = 0	
1 element	size = 1	0	1. element	size = 0	
1. position	size = 5	0	1. element	size = 4	
2. position	size = 5	1	2. element	Size = 4	
last position	size = 5	4	5. element	size = 4	
too large index	size = 5	6	exception	size = 5	
negative index	size = 5	-1	exception	size = 5	

The importance of this exercise is to find (most of) the bugs by systematic test case design (incremental test, bug fixing and retesting).

SOLUTION 6

For state diagrams, test coverage could be:

a) To bring the object around into all its possible states (100 % state coverage)
b) To produce all possible transitions (100 % transition coverage)

Potentially, that can be a very large number of states and transitions. Therefore it is a good idea to group them in equivalence classes (states that are handles the same way). As you have probably experienced during your testing efforts of the list, different behavior for the list revolves around an empty list, a list with one element, and lists with more than one element, making them into three different equivalence classes (this is also how the example test cases for the remove method are designed – they concentrate on these three equivalence classes).

In *A Practitioner's Guide to Software Test Design* (2004) by Lee Copeland, four different levels of coverage for state transition are described. You are not supposed to master his following test coverage calculations in details as part of the curriculum. They primarily serve the purpose of illustrating how test coverage can be measured on other structural items than code lines.

i. You can create a set of test cases such that all **states** are "visited" at least once under test. Generally this is a weak level of test coverage.

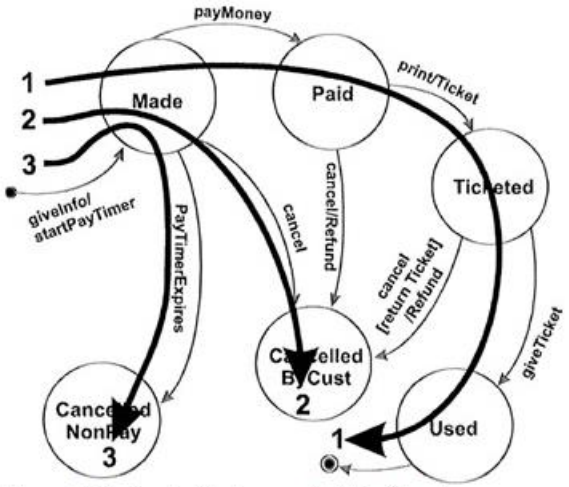

Figure 7-10: A set of test cases that "visit" each state.

ii. Create a set of test cases such that all **events** are triggered at least once under test. Note that it might be the same test cases as for state coverage. Again, this is a weak level of coverage.

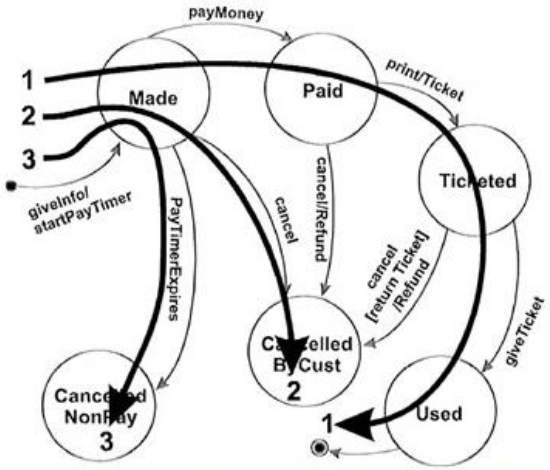

Figure 7-11: A set of test cases that trigger all events at least once.

iii. Create a set of test cases such that all **paths** are executed at least once under test. While this level is the most preferred because of its level of coverage, it may not be feasible. If the state-transition diagram has loops, then the number of possible paths may be infinite. For example, given a system with two states, A and B, where A transitions to B and B transitions to A. A few of the possible paths are:

A→B
A→B→A
A→B→A→B→A→B
A→B→A→B→A→B→A
...

and so on forever. Testing of loops such as this can be important if they may result in accumulating computational errors or resource loss (locks without corresponding releases, memory leaks, etc.).

iv. Create a set of test cases such that all **transitions** are exercised at least once under test. This level of testing provides a good level of coverage without generating large numbers of tests. This level is generally the one recommended:

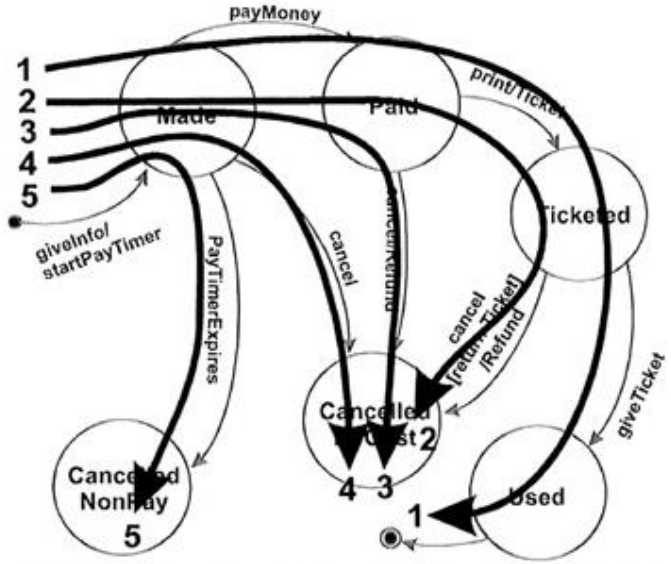

Figure 7-12: A set of test cases that trigger all transitions at least once.