Specification Based Techniques (Black Box)

(and a little on experience based testing)

Hans Schaefer
hans.schaefer@ieee.org
http://www.softwaretesting.no

Test Basis

Specification / change request

If we do not have this:

- **User manual**
- **Online help**
- **Standards for interfaces**
- **Business rules and process**
- **Data model, data use descriptions etc.**
- **Heuristics / experience**

In the worst case: The system itself (what we know, what we see under test)

Basic Techniques for Black Box-Testing

Focus on data variation :
- Equivalence class partitioning
- Boundary value analysis
- Special value test

Focus on logic:
- Decision tables

Focus on longer scenarios:
- Use of state transitions diagrams (state transition testing)
- Use case-based test

Test coverage for black box

Percentage of all equivalence classes tested
Percentage of all boundary values tested
Percentage of all “specialities” tested

Percentage of combinations tested
Percentage of “possible combinations” tested

Percentage of state transitions tested
Percentage of use case flows tested

Techniques for Data Variation

Data can have different values
- Input data
- Internal data
- Settings, preferences, references
- Environment data
- Output data

Focus at least on right and wrong data!
- Input (right, wrong, extreme, special, present, not present, default)
- Output (different formats, values, relations, extreme, problem messages)
- Intermediate values (normal, extreme, place problems)

What are such Techniques Good for?

Find problems with input validation
Find missing error handling
Find unclear boundaries
Find problems with too large output
Find problems with special values

Don’t Forget Data!

Where do inputs come from and outputs go to?

Input:

Keyboard, mouse, files, data bases, internal global data, network, operating system, parameters, devices...

Output:

Screen, printer, loudspeaker, files, data bases, internal global data, network, devices, operating system, parameters...

Do not forget internal storage.
Do not forget what is not easily visible!

Equivalence Class Partitioning

Definition: An equivalence class is a collection of values where YOU BELIEVE the program handles all of them in the same way.

Four main concepts:

- **Cover ALL POSSIBLE inputs and/or outputs**
- **Assume: Every value in a class is handled in principle in the same way**
- **No overlap between classes: A value belongs to exactly one class**
- **Mark every class depending of it is valid or invalid (error)**

The technique is useful for all test levels.
It may be blind for some faults.

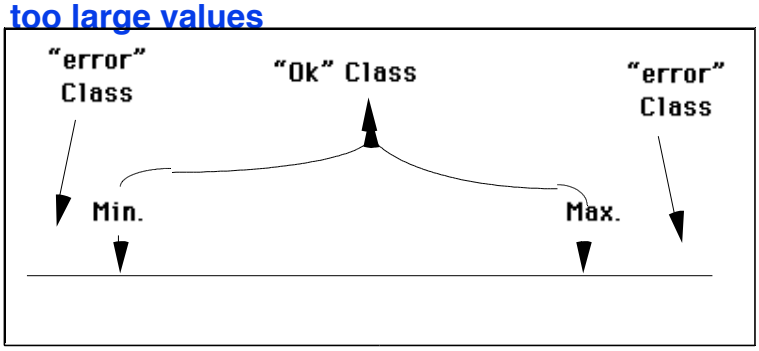
How to Find Equivalence Classes

Numerical data with a value range: 3 classes

(1) too small values

(2) allowed values

(3) too large values



How to Find Equivalence Classes

Time

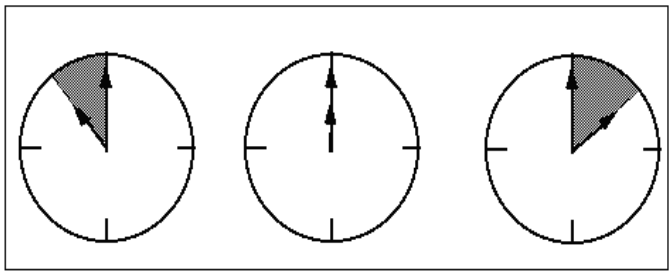
(1) too early,

(2) in allowable time range,

(3) too late.

- Before
- Same time
- After something else

(4) Impossible data values



How to Find Equivalence Classes

Discrete values: Many classes

(1)...(n) One class for every allowed value, and

(n+1) for ”something else" (not allowed).

Condition: Two classes.

(1) Condition fulfilled

(2) Condition not fulfilled.

How to Find Equivalence Classes

Existence of an input: Two classes

(1) input is found / given

(2) input is not found / given

Format / data type of an input: Two or more classes

(1) correct format and data type (maybe alternatives)

(2) not correct

More rules to apply

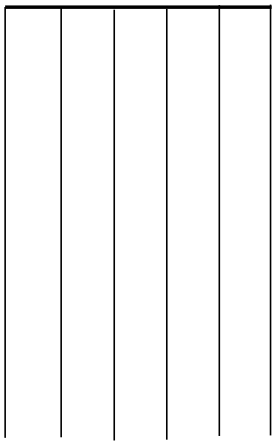
- If you believe some values are handled differently, partition into subclasses! (Pessimist-rule)
- Do this with all inputs and outputs.
- If data are checked before, there may be no "invalid" classes.

You may make the class partition hierarchic, if classes are dependent on each other (see logic based methods - decision tables, classification tree method (->www.systematic-testing.com).

Example for a test table

Function 1
- Input channel 1
  - Data structure 1
    - Data element 1
    - Data element 2
      - Class 1 OK
      - Class 2 error
      - Class 3 OK
    - Data element 3
  - Data structure 2
- Input channel 2
- Output channel 1

Columns for test cases



Data selection from Class-partitioning

Work steps:

(1) Choose test data and cover all input-classes.

(2) Check which output-classes (effects) are covered. Try to cover all of them.

(3) Choose combinations of classes if you think you need. (covering cause and effect combinations - see later)

Data not relevant for *"this"* test should be included using standard values.

Details

For "OK" input classes:

Choose data such that they cover as many "OK" classes as possible (all inputs "OK")

For "error" input classes: Do not combine!

Choose data such that they only cover one "error" class, and everything else is "OK" (any random "OK" classes as far as possible) (otherwise, bugs may be "masked")

Start with global input. (File, database, tables, ...)

Choose global input in such a way that all other classes have a chance to be covered

Do you want to test combinations?

Necessary, but not in foundation syllabus.

Explosive growth in number of test cases.

Possible to combine pair wise between parameters.

Possible compromise: Put the most important combinations into the equivalence class table.

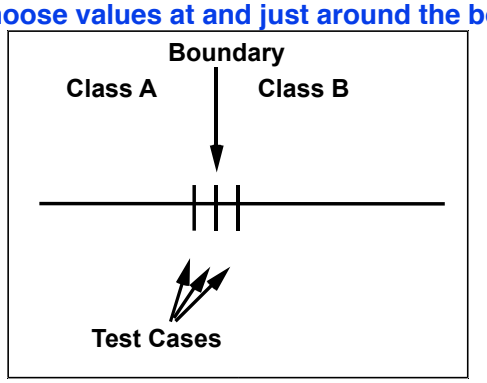
Example: Last day in a month. Four classes. Classes for February also depend on the kind of year!

Boundary value analysis

Boundary value uncover these faults:

Wrong operator: <, <=, >, >=, ”off-by-one”, Maybe also ==, <>

- **(1) Choose boundaries for all inputs and outputs.**
- **(2) Choose values at and just around the boundaries**



”just around” =
+ or -1
+ or - tolerance

Details for boundary value analysis

- Just below minimum
- Minimum
- Just above minimum

- Just below maximum
- Maximum
- Just above maximum

Smaller method:
If two out of three cases:
At least one value in each equivalence class!

Maximal (safe) method:
Two values near boundary in both equivalence classes.

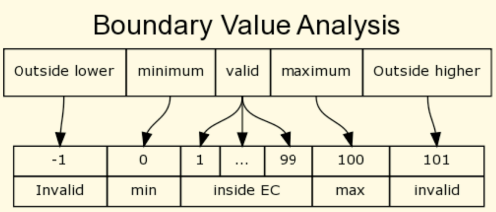
- First, last, element in a list, buffer or file.
- Fastest, slowest signal arrival times
- Much input, no input (empty and full table, file, buffer)
- Change of day, month, year.
- Boundary values for several dimensions.

Notes On Notation

- < means ”less than”
- <= means ”less than or equal to”
- > means ”greater than”
- >= means ”greater than or equal to”
- [1 .. 12] means ”from and including 1 to and including 12” - like for month number
- (1 .. 12) means ”between 1 and 12” (1 and 12 are not included here)

Boundary Value Analysis

Valid: Integers from and including 0 to and including 100



Boundary Values For Repetitions

List- and table processing

(1) Zero in the list
(2) One in the list
(3) Several in the list (especially 2, max - 1)
(4) Maximum list length
(5) Too many in the list

Valid And Invalid Boundaries

Valid boundaries are boundaries in valid equivalence classes, invalid boundaries the opposite.

Example

- Valid area [1 ... 100]
- Valid boundaries = 1, 2, 99, 100
- Invalid boundaries = 0, 101

2 and 99 are less important! (same class as 1 and 100)

More On Boundary Values (not for exam)

Boundaries can be forgotten to describe (never declared) (but implemented in the program). -> Think which boundaries COULD be interesting.
There may be boundaries in reality, but forgotten in the program. -> check specification and domain.
Boundaries may be copied from some other place (another module or program or system). -> Check if interfaces or used software or hardware imply boundaries.
Boundaries may be dynamic. For example when several users or threads share memory or resources. -> Think about dynamic boundaries.
Boundaries may be hidden in the algorithm. -> Think about possible boundaries in the mathematics, in results remembered on the way or in outputs.

Error guessing / Special Value Test

Also: fault-attack

Some values often show faults.

Make test cases with values where the used functions works in a special way.

You need knowledge about typical faults!

Maintain a living list, depending on methods, programming language, platform, application domain etc.

Use this method after systematic methods!

-> Exploratory testing

"The in-built bad nature of things":
Implicit conditions and restrictions.

Error Guessing / Special Value Test

Make test data for any special values.

- Zero, minus one and plus one for arithmetic functions
- 90 degrees and multiples for angle functions
- Max and min values due to hardware (32 bits or else)
- Empty string, special and national characters for text functions
- Special characters used in programming language, file system etc:
- "#$%&/()=`´*@:-.;_< >
- Empty data area for buffer handling
- "magic numbers" in general
- Default values
- For database fields, example: `Mr. <" </XML> % , "Brian O'Date"`

Very important for security tests against Cross Site Scripting!

More: Error Guessing

- Empty input lists
- Quit just after start
- Database full or not accessible
- Input missing
- Too much input (repeated too many times)
- Wrong correlation between inputs
- ”Forbidden” situations
- ”Impossible” situations
- Reset of timer
- National characters
- Numbers with blanks, leading plus, dots for thousands, ...

General ideas:

- Unclear items
- Inconsistencies
- Misunderstandings
- What if...
- What you would like
- Not well analyzed areas

Techniques Focusing On Logic

Systems often have complex decisions.

A good specification method is decision tables.

Tests can be derived from these.

The method may lead to much work, if all combinations are considered.

What Are Logic Tests Good For?

Find logic faults

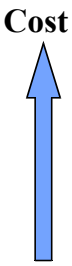
- AND - OR exchanged
- Conditions forgotten
- Conditions wrong way (NOT)
- Faults in combining conditions (parenthesis setting)

Analyze: Possible Relations Between Data

- Combinations
- Order
- Syntax
- Conditions across fields
- If things really matter: Check all input data elements against each other.

Possible Logic Tests

- All combinations of decisions and actions
- Systematic use of **decision tables**
- Pair wise combinations
- Every condition yes and no, by itself



How To Make A Decision Table

Start from the table over equivalence classes (in column 1).

Make a table with many empty columns to the right.

Make a row for every condition or every equivalence class.

Write the outputs or effects below the input equivalence classes (in column 1).

Example: ATM

Conditions (causes)
Valid card (y/n)
first PIN correct
1-2 times wrong PIN
3 incorrect PIN
Money available (y/n)
Effects (output)
Reject card (y/n)
Try again! (y/n)
Eat card (y/n)
Cash out (y/n)

Decision Table Continued

Make columns for every combination of conditions.
(If they all are logical values, the number is 2^n otherwise the product of the numbers of equivalence classes).

Get rid of conditions that are impossible ("limited entry decision table").

List all effects for every combination of conditions.

Make sure every condition and effect is present at least once as YES and as NO.

Example Decision Table (ATM)

Condition #	1	2	3	4	5	6	7	Check, both Y and N?
Valid card	N	Y	Y	Y	Y	Y	Y	OK
first PIN correct	-	N	N	Y	Y	N	N	OK
3 incorrect PIN	-	N	N	N	N	Y	Y	OK
1-2 wrong PIN	-	Y	Y	N	N	N	N	OK
Money available	-	N	Y	N	Y	N	Y	OK
Output								
Reject card	Y	N	N	N	N	N	N	OK
Try again!	N	Y	Y	N	N	Y	Y	OK
Eat card	N	N	N	N	N	Y	Y	OK
Cash out	N	N	Y	N	Y	N	N	OK

Number of correct/incorrect PIN trials is mutually exclusive.
Test case 1 covers all possible combinations of PIN and money: You cannot go further with an invalid card!

Decision Table Completeness Check

Every column gets a counter:

If Y and N everywhere: Counter := 1

For every condition where a dash is given:

Counter := Counter * number of equivalence classes for this condition.

Sum of all counter values should be 2^n (for n conditions) or the product of number of equivalence classes.

Sum on next page = 12.

Example Decision Table (ATM)

Condition #	1	2	3	4	5	6	7	Check, both Y and N?
Valid card	N	Y	Y	Y	Y	Y	Y	OK
first PIN correct	-	N	N	Y	Y	N	N	OK
3 incorrect PIN	-	N	N	N	N	Y	Y	OK
1-2 wrong PIN	-	Y	Y	N	N	N	N	OK
Money available	-	N	Y	N	Y	N	Y	OK
Counter	6	1	1	1	1	1	1	12
Output								
Reject card	Y	N	N	N	N	N	N	OK
Try again!	N	Y	Y	N	N	Y	Y	OK
Eat card	N	N	N	N	N	Y	Y	OK
Cash out	N	N	Y	N	Y	N	N	OK

Number of correct/incorrect PIN trials is mutually exclusive!

Decision Table: Deleting Unnecessary Columns

We can still delete some cases:

If columns have identical outputs (effects),
And only one input conditions varies,
then they can be combined.

Replace the respective input which is not interesting with a dash.

Continue doing this until all columns differ.

Example Decision Table (ATM)

Condition #	1	2	3	4	5	6	7	Check, both Y and N?
Valid card	N	Y	Y	Y	Y	Y	Y	OK
first PIN correct	-	N	N	Y	Y	N	N	OK
3 incorrect PIN	-	N	N	N	N	Y	Y	OK
1-2 wrong PIN	-	Y	Y	N	N	N	N	OK
Money available	-	N	Y	N	Y	N	Y	OK
Counter	6	1	1	1	1	1	1	12
Output								
Reject card	Y	N	N	N	N	N	N	OK
Try again!	N	Y	Y	N	N	Y	Y	OK
Eat card	N	N	N	N	N	Y	Y	OK
Cash out	N	N	Y	N	Y	N	N	OK

Money available is not interesting to check!

Example Decision Table (ATM)

Condition #	1	2	3	4	5	6		Check, both Y and N?
Valid card	N	Y	Y	Y	Y	Y		OK
first PIN correct	-	N	N	Y	Y	N		OK
3 incorrect PIN	-	N	N	N	N	Y		OK
1-2 wrong PIN	-	Y	Y	N	N	N		OK
Money available	-	N	Y	N	Y	-		OK
Counter	6	1	1	1	1	2		12
Output								
Reject card	Y	N	N	N	N	N		OK
Try again!	N	Y	Y	N	N	Y		OK
Eat card	N	N	N	N	N	Y		OK
Cash out	N	N	Y	N	Y	N		OK

This is the final result: One test case per column!

Techniques With Focus On Longer Scenarios

Test based on state transitions diagrams

Test based on use cases

Test Based On States And State Transitions

Does the program change its behavior based on the history of inputs?
Is the order of actions important?

Think of a mobile phone: It could be a camera, SMS machine or phone.

In this case:

- Find every state
- Find every transition
- What triggers a transition?
- Result of every transition
- Possible ”guards” (conditions for transitions)
- Even wrong inputs to the states
- Make a state transition diagram!
- Test all transitions

What Is A State Based Test Good For?

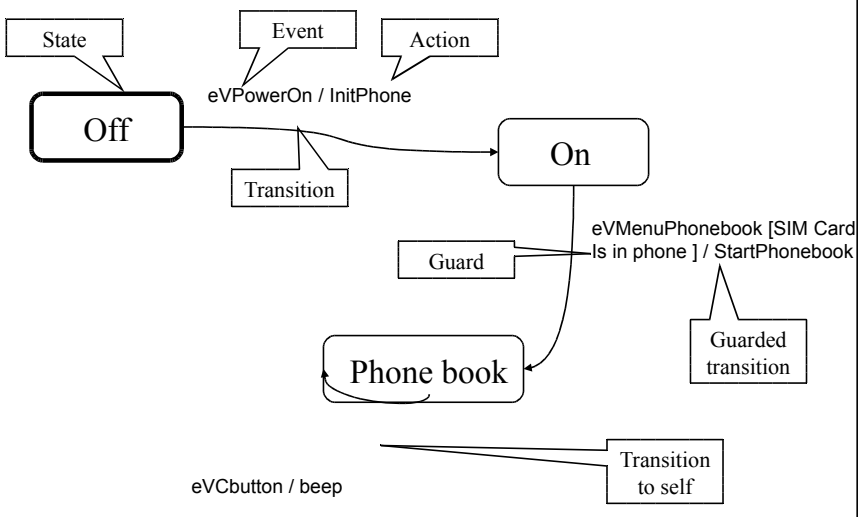
Find faults where history of input plays a role.

Find faults where input possibilities are forgotten.

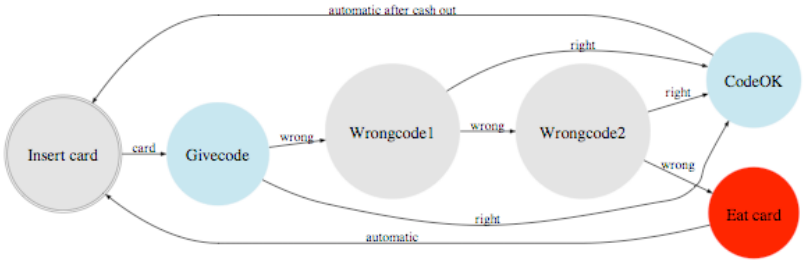
Find faults in longer scenarios or dialogs.

Many real time systems are implemented using state machines.

State Transition Diagrams

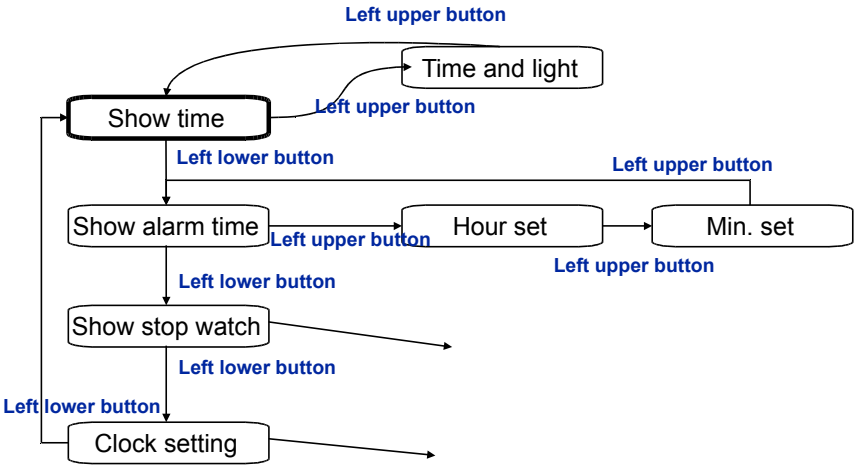


Example ATM



Example Digital Watch

How does your digital clock work?



Specification Methods

State diagram
State table
“Finite state machine”
Syntax description for input
Communication protocol
State tree

Test Case Design Methods

(State cover)
Transition (or Branch) cover
State-event cover
Switch cover
More advanced criteria

Faults In Models And Implementation

(not for exam)

Faults in the model ----------> REVIEW

- Start state not defined
- Guard coupled to state (not transition)
- Guards overlap, wrong, missing, extra
- States wrong, missing, extra
- Transitions wrong, missing, extra
- Several transitions from same state with same input (not deterministic)
- …

Faults in the implementation ----------> TEST

- Extra / missing / corrupt / wrong state
- Missing / wrong action
- Deadlocks
- Sneak paths
- Trap doors …

Designing Test Cases

Define a set of input sequences starting from the start state and if possible coming back to it.

Example for digital watch.

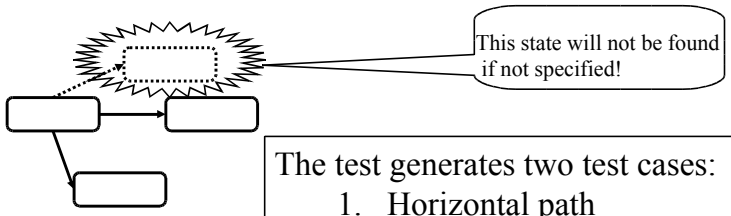
Test step nr.	Input + Guards	Start state	Expected next state	Expected output
1	Left upper button just at alarm time	Show alarm time	Show alarm time	Alarm stop, Show alarm time
2	Left upper button	Show alarm time	Time set	Time blinks

Start with test cases that test typical flows, design more complex ones later. Do not forget impossible or invalid transitions!

Transition (or Branch) Cover

Every state-transition, i.e. every branch, is used (at least) once.

This test reliably detects output (or operation) errors.
The test does not detect forgotten states.



The test generates two test cases:
1. Horizontal path
2. "Down" path

Generating A Transition / Branch Cover By Transition Tour

Start at the start state.
Choose events which run through the state machine in such a way as to cover every transition in the diagram at least once.
(Chinese Postman algorithm)

The state machine under test is assumed to be minimal (no duplicate states), and strongly connected (no islands).

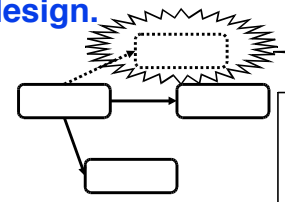
The method can be applied to both fully and incompletely specified state machines (Fully specified = a transition for every event specified in every state).

State-event Cover

Execute every event in every state. Check that this is handled correctly.

This test is equivalent to branch cover for completely specified state machines.

This test detects transitions that are forgotten in the design.

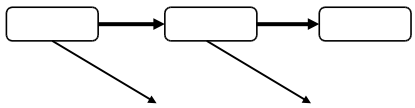


The test generates three test cases:

1. Horizontal path
2. "Down" path
3. "Up" path

Switch Cover (not for exam)

A switch (or 1-switch) is a transition-to-transition pair (combination of two transitions).

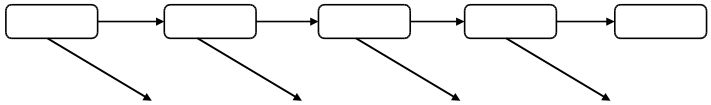


Every switch starting in every transition must be executed.

With this test design, Operation errors are detected.
State-transition errors will be detected if states are "1-distinguishable".
I.e. if, for each pair of states, there is at least one input which, when applied to the pair, yields different outputs.
Equivalent to branch coverage for program code, starting from every state.

N-Switch Cover (not for exam)

n-switch = A sequence of consecutive transitions (branches) of length n+1.



Impossible to do in practice!

If n = the number of states - 1
Then
You reliably find
Output (operation) errors
Transfer errors
Missing state

Summary For Test Of State Transitions

Run object life histories through the system.
Example: Life of an account, order etc.
Every test case back to start (i.e. Start screen).
Exceptions, user errors, cancel!
Make some long histories (soap operas).

Use Case-Based Test

The method checks business flows, even across several systems or system parts.

- Normal flow (everything correct)
- Alternative scenarios (user errors, input errors, equipment failures, alarms, online-help...)
- Cancel, repetitions
- Even misuse (Internet search -> misuse cases)

Use Case-Based Test

- A USE CASE describes "something the program shall do", and is triggered by an external actor.
- It is about who is trying to achieve what with the system, and in what context.

What if input is wrong or left out?
Exceptions?
User errors?
Equipment failures?
Set up parameters?

What Is Use Case-Based Test Good For?

- Problems and risks in the process flow through the system
- Problems due to long scenarios
- For system and acceptance test
- Finds interaction faults that test of single components does not find

Work Steps For Use Case-Based Test

1. >= 1 test case for normal flow
2. >= 1 test case for every alternative flow
3. Possibly special combinations of partial flows
   (maybe even combinations of different use cases after each other)
4. Define test data

Test Data For Use Case-Based Test

Use the normal data-based methods

(equivalence classes, boundary values, special values, decision tables)

For Reference: Design By Contract

Design by Contract is a mechanism pioneered by Eiffel that characterizes every software element by answering three questions:

- What does it expect?
- What does it guarantee?
- What does it maintain?

Answers take the form of preconditions, postconditions, and invariants. For example, starting a car has the precondition that the ignition is turned on and the postcondition that the engine is running. The invariant, applying to all operations of the class CAR, includes such properties as dashboard controls are illuminated if and only if ignition is on. With Design by Contract, such properties are not expressed in separate requirements or design documents but become part of the software; languages such as Eiffel and Spec#, and language extensions such as JML, include syntax keywords such as require, ensure, and invariant to state contracts. Applications cover many software tasks: analysis, to make sure requirements are precise yet abstract; design and implementation, to obtain software with fewer faults since it is built to a precise specification; automatic documentation, through tools extracting the contracts; support for managers, enabling them to understand program essentials free from implementation details; better control over language mechanisms such as inheritance and exceptions; and, with runtime contract monitoring, improvements in testing and debugging, which AutoTest (EiffelStudio) takes further.

Reference 1. B. Meyer, Applying Design by Contract, IEEE Computer, Oct. 1992, pp. 40-51.

Literature

- Glenford Myers, The Art of Software Testing, John Wiley, 1979
- Boris Beizer, Software Testing Techniques, Van Nostrand Reinhold, 1990.
- Boris Beizer, Black box-testing, 1995
- James Whittaker, How to Break Software, Addison-Wesley, 2002.
- Linz, Spillner, Schaefer, Software Testing Foundations, 3rd ed., Rocky Nook, 2011.
- Paul C. Jorgensen, Software Testing - A Craftman's Approach, 2nd ed., CRC Press 2002.
- Graham Bath and Judy McKay, The Software Test Engineer's Handbook, A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates, Rocky Nook, 2008.