Test Design Techniques

Test

PBA Softwareudvikling/BSc Software Development

Tine Marbjerg

Spring 2018

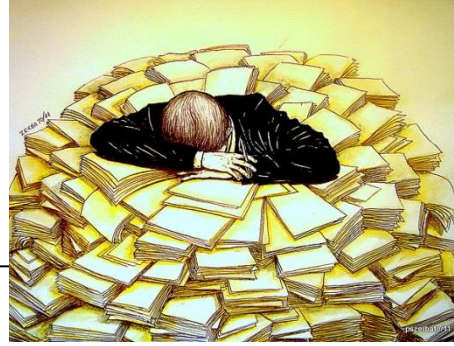
Today's Topics

- Test Development Process (Black chapter 4)
- Test Case Design (Black chapter 4)
  - black-box techniques
  - white-box techniques
  - experience-based techniques - later

Objectives

- Be able to design test cases based on black-box and white-box techniques
- Understand the concept of test coverage

The Test Development Problem

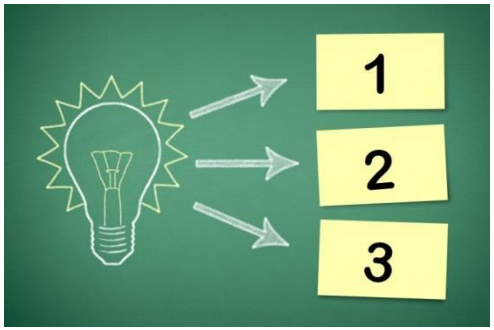


The fundamental problem of testing software

- We cannot make exhaustive testing ⇒
  - We need to have a clever testing methodology
- The tests must be carefully designed ⇒
  - Find subset of all possible tests, with highest probability of finding defects

Goal: Adequate Test Coverage

The Test Development Proces



Test analysis: **Identify test conditions** (i.e. what to test)

Test design: **Specify test cases**

Test implementation: **Specify test procedures (scripts)**

Standards for Software Test Documentation

- IEEE 829 Standard specifies a set of documents for use in **eight defined stages of software testing**
  - each stage *potentially* producing its own separate type of document, see http://en.wikipedia.org/wiki/IEEE_829

Primary course focus

- Test Plan
- Test Design Specification (identify test conditions)
- Test Case Specification (design test cases)
- Test Procedure Specification (write implementation/scripts)
- Test Item Transmittal Report
- Test Log
- Test Incident Report
- Test Summary Report

1. Test Analysis – Identify Conditions

What is a Test Condition ?

- **A test condition is simply something we could test!**
- Other (better ?) names for 'test conditions'
  - test inventory | test requirements | test objectives | test possibilities

- Test conditions are chosen based on test strategy e.g.
  - risk, models of the system, expert advice, heuristics

Examples of Test Conditions

Examples

- If we want to measure coverage of code decisions (branches):
  - *Test basis* = code
  - *List of test conditions* = decision outcomes (true/false)

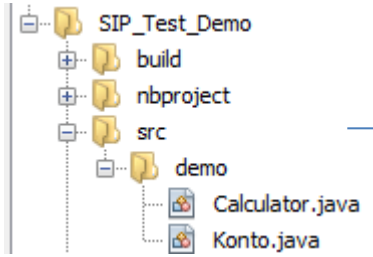
- If we have a requirement specification
  - *Test basis* = requirement specification
  - *List of test conditions* = table of contents (agile: backlog)

Test conditions can go from rather vague to more specific

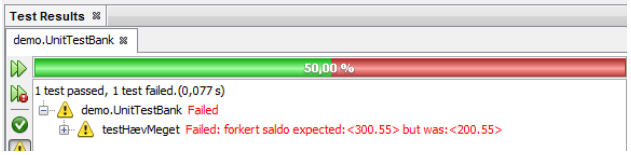
- Marketing campaign examples Black p. 72: "Teenagers in Midwest" vs. "Particular male customer on pay-as-you-go with less than 10 $ credit"

Traceability of Test Conditions

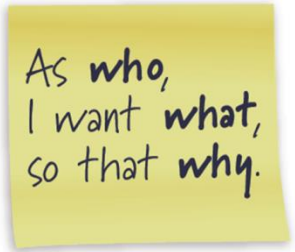
- **Test conditions should link back to their sources**
  - **Horizontal** – all test documentation at a given test level (e.g. conditions → test cases → scripts)



test case	input	exp. output
1	500.55	400.55
2	500.55	300.55



  - **Vertical** – through the layers of development documentation (e.g. requirements → code)



test case	input	exp. output
1	500.55	400.55
2	500.55	300.55

```
class Konto {
    private double saldo;

    Konto(double d) {
        saldo = d;
    }

    void hæv(int i) {
        saldo = saldo - i;
    }

    double hentSaldo() {
```

IEEE 829 Standard: Test Design Specification Template

A template for a Test Condition Specification:

1. Test design specification ID
2. Feature(s) to be tested
3. Approach refinement
4. Test identification
5. Feature pass / fail criteria

The template is not used as such in agile, but skills to identify relevant test conditions & ensuring traceability between test cases and theirs sources are always relevant!

Test Design Specification - Example

- **Test design specification ID**
  - TDS-02-20-12-B
- **Feature(s) to be tested**
  - Withdraw cash
  - Check account balance
- **Approach refinement**
  - Withdrawal not allowed between 2.00 a.m. and 4.00 a.m.
- **Test identification**
  - TC01 – withdraw $20 from a valid account with $200
  - TC17 – withdraw $100 from a valid account with $100
  - . . .
- **Feature pass / fail criteria**
  - All withdraw cash tests must pass
  - 90% of check account balance must pass

2. Test Design – Specify Test Cases

Test Cases

- Test cases in brief: A set of **test inputs**, **execution conditions** and **expected results**

- Test cases must be **specific**
  - where test conditions can be vague

Oracle = source of information about the correct behavior of the program, e.g.

- Human being's judgment (e.g. users)
- Requirement specification
- Other products e.g. (second program with different algorithm)
- Heuristic methods

IEEE 829 Standard: Test Case Specification Template

- Test case ID
- Test items
- Input specifications
- Output specifications
- Environmental needs
- Special procedural requirements
- Intercase dependencies

Again, template is not that important!
Input & Expected output are really important!

IEEE 829 Standard - Elaborated

- Expected results
  - Output (on screen)
  - Change to data or state
- Environment
  - HW or SW
    - Mock objects, stubs, other applications
  - Pre- or post conditions
- Special procedural requirements
  - Identify any special constraints on the test case(s)
    - Special set-up or wrap-up
- Intercase dependencies
  - Identify any prerequisite test cases

Test Case Specification - Example

- **Test case specification ID**
  - • TC01: withdraw $20 from a valid account with $200
- **Test items**
  - Requirement spec RS-08-12-11
  - Software code: withdrawCash(), checkBalance()
- **Input specifications**
  - 20
- **Output specifications**
  - 180
- **Environmental needs**
  - Driver_D_11, Stubs: setAccountBalance()
- **Special procedural requirements**
  - Convert dollars into cents

Simple Test Case Specification Template

Test Case	Special Notes	Input			Output	
		Var 1	Var 2	...	Expected	Actual
TC001						
TC002						
TC007						
TC009						
...						

3. Test Implementation – Specify Test Procedures & Scripts

Standard: Test procedure specification Template (IEEE 829)

- Test procedure specification identifier
- Purpose
- Special requirements
- Procedure steps

- Document describes the steps to be taken in running a set of tests and the executable order of the tests
- Test procedures / scripts are formed into a test execution schedule that specifies which procedures are to be run first
- Test schedule says when a given script should be run and by whom

Test Procedures (scripts)

The order of execution – two styles:

1. **Cascading** test cases - Test cases may build on each other (intercase dependencies) , e.g.
   - Create a record, Read the record, Update the record
2. **Independent** test cases - Each test case is entirely self contained
   - Pro: any test can be executed in any order
   - Con: each test tends to be larger and more complex

Test Procedures - Examples

Test procedure DB15: Set up customers for marketing campaign

Step 1: Open database with write privilege

Step 2: Set up customer Bob Flounders male, 62, Hudsonville, contract

Step 3: Set up customer Jim Green male, 17, Grand Rapids, pay-as-you-go, $8.64

Step 4: …

Test procedure MC03: Special offers for low-credit teenagers

Step 1: Get details for Jim Green from database

Step 2: Send text message offering double credit

Step 3: Jim Green requests $20 credit, $40 credited

Automated test procedure

- An automation script is written in a programming language that can be interpreted by tool

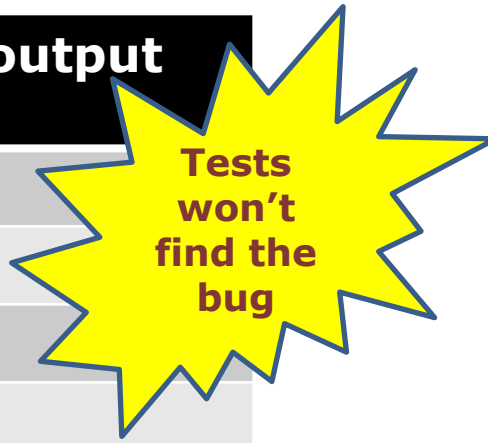
Test Design Techniques

Help select relevant subset of tests

The Impossibility Of Testing Everything - Example

```
int myMethod(int j) {
    j = j - 1; // should be j = j + 1
    j = j / 30000;
    return j;
}
```

input(j)	Expected output	Actual output
1	0	0
42	0	0
40000	1	1
-64000	-2	-2



Example from *Testing Object-Oriented Systems* by Robert Binder

The Impossibility Of Testing Everything 2

```
int myMethod(int j) {
    j = j - 1; // should be j = j + 1
    j = j / 30000;
    return j;
}
```

- If 16 bit system (Java short data type):
  - 2 bytes, signed, -32,768 to 32,767 = **65,536 possible inputs**

- Very few of the possible input values will find this defect. What is the chance that you will pick these values?

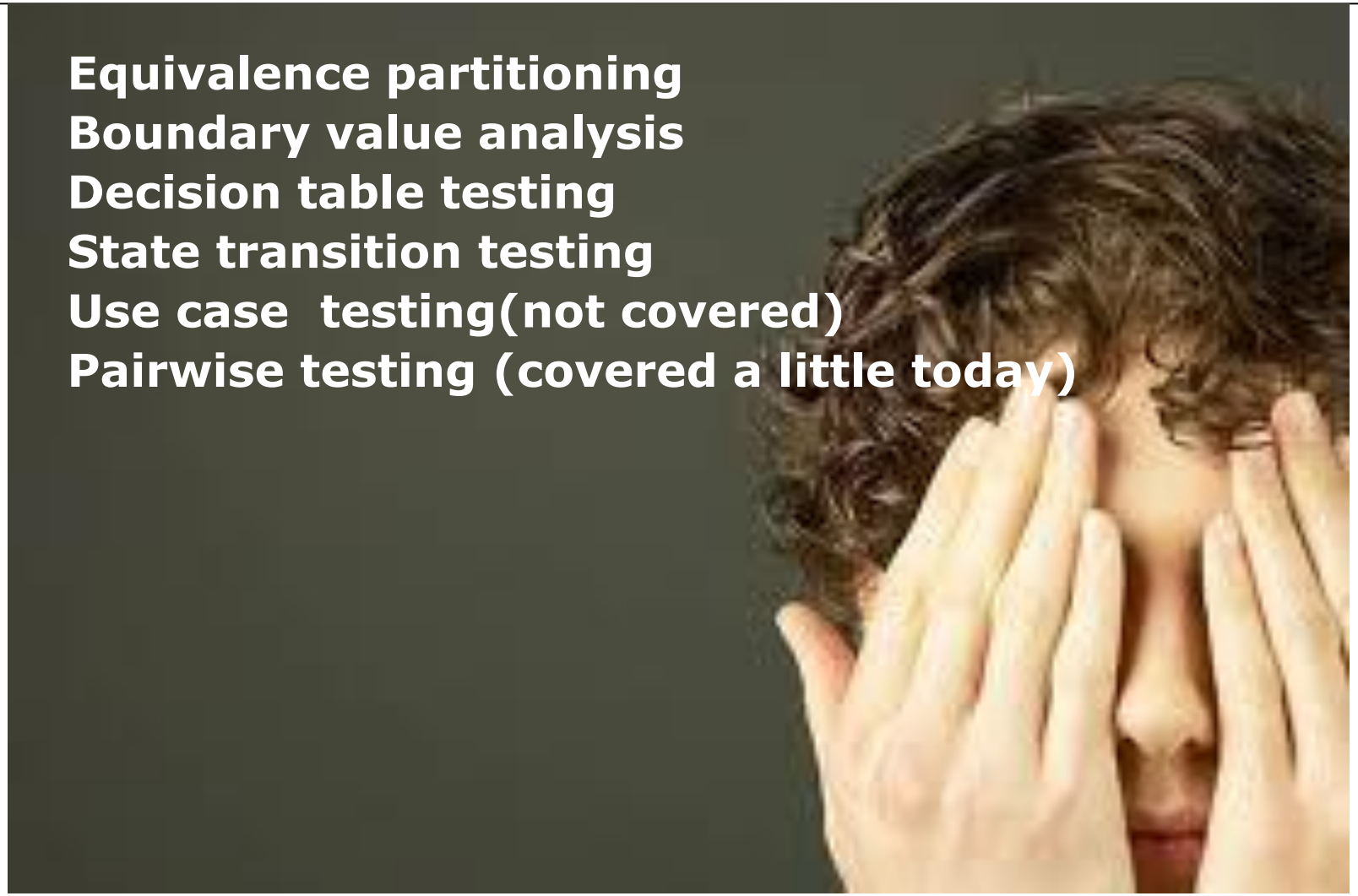
  -30001, -30000, -1, 0, 29999, and 30000

Test Design Techniques

- Static testing techniques
  - Generally used before any tests are executed on the software
- Specification-based (black-box) techniques
  - Input/output driven
  - Focus on the functional externals
  - Applied on all test levels where specification exists
- Structure-based (white-box) techniques
  - Logic driven
  - Focus on internal structure of the software
  - Primarily unit and integration test level (good tool support)
- Experience-based techniques
  - People's knowledge and skills are prime contributor to test case design
  - Complements the above, especially if no (or inadequate) spec.
  - Test cases are derived less systematic, but may be more effective ☺

Black Box Techniques

Black-Box Techniques



Equivalence Partitioning

- Aims at minimizing the number of test cases based on groups of *similar* input values
- Input values are partitioned into *equivalence classes* if they result in the same program behavior

✓ A test case is written for each partition

✓ Invalid input are separate equivalence classes

OBS! *Equivalence class is the same as equivalence partition*

Equivalence Partitioning - Example

```
// pre: 0 < age
// post: returns true if age >= 18, otherwise false
public boolean legalAge(int age)
```

Equivalence classes	Test case (legalAge)
age <= 0	invalid: -1
0 < age <18	not legal age: 10
18 <= age	legal age: 20

Equivalence Partitioning - Exercise

In a tax payment system, an employee has £4000 of salary tax free, the next £1500 is taxed at 10%, the next £28000 after that is taxed at 22% and any further amount is taxed at 40%.

What are the equivalence classes and which of these values fall into three different equivalence classes?

a) £4000; £5000; £5500
b) £32001; £34000; £36500
c) £28000; £28001; £32001
d) £4000; £4200; £5600

Boundary Value Analysis

- Errors often show at the boundaries between equivalence classes

- Choose minimum and maximum values from an equivalence class together with first or last value respectively in adjacent equivalence classes + “normal” value in the middle

Boundary Values - Same Example Again

```
// pre: 0 < age
// post: returns true if age >= 18, otherwise false
public boolean legalAge(int age)
```

Test cases (not legal age)

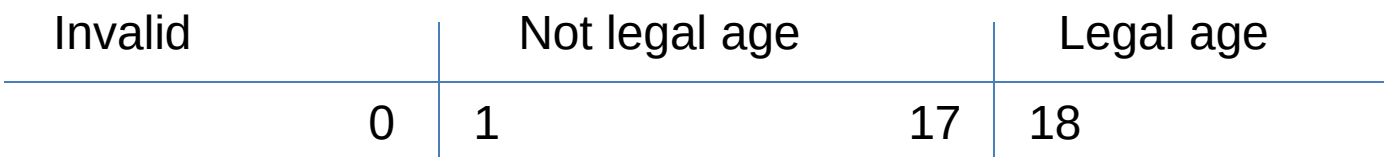
0 (adjacent partition)

1 (min. value - close to boundary)

10 (normal input)

17 (max. value - close to boundary)

18 (adjacent partition)



Open Boundaries

- Open boundaries are more difficult to test, but there are ways to approach them.
- The best solution is to find out what the boundary should be specified as ☺
  - Look in spec
  - Ask product owner
  - investigate other related areas of the system. Ex.:
    - The field that holds the account balance figure may be only six figures plus two decimal figures. This would give a maximum account balance of $999 999.99 so we could use that as our maximum boundary value.

Boundary Value - Exercise

Title text box allows 1 - 30 characters

Find test cases with BVA (boundary value analysis)

Equivalence/Boundary Testing - Issues

- They don't explore **combinations of input**
  - A program can fail because of a combination of certain values causes an error

- **Decision tables** are models of complicated logic
  - Good for *combination of inputs* that result in *different actions* being taken
  - Input-output behavior can be transformed into a boolean function
  - A systematic way of stating complex business logic

Decision Tables for combination of input Ex.– Loan Application Black 4.3.2

- You enter loan amount of monthly payment or number of years to take to pay it back (term of the loan). If you enter both, system will make a compromise between them if they conflict:

TABLE 4.2 Empty decision table

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered				
Term of loan has been entered				

In our case 2^2

Two to the power of number of things to be combined

TABLE 4.3 Decision table with input combinations

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F

TABLE 4.4 Decision table with combinations and outcomes

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

Decision table helps find alternative scenario

- What if customer doesn't enter anything?
- Making a decision table can help find unclarity

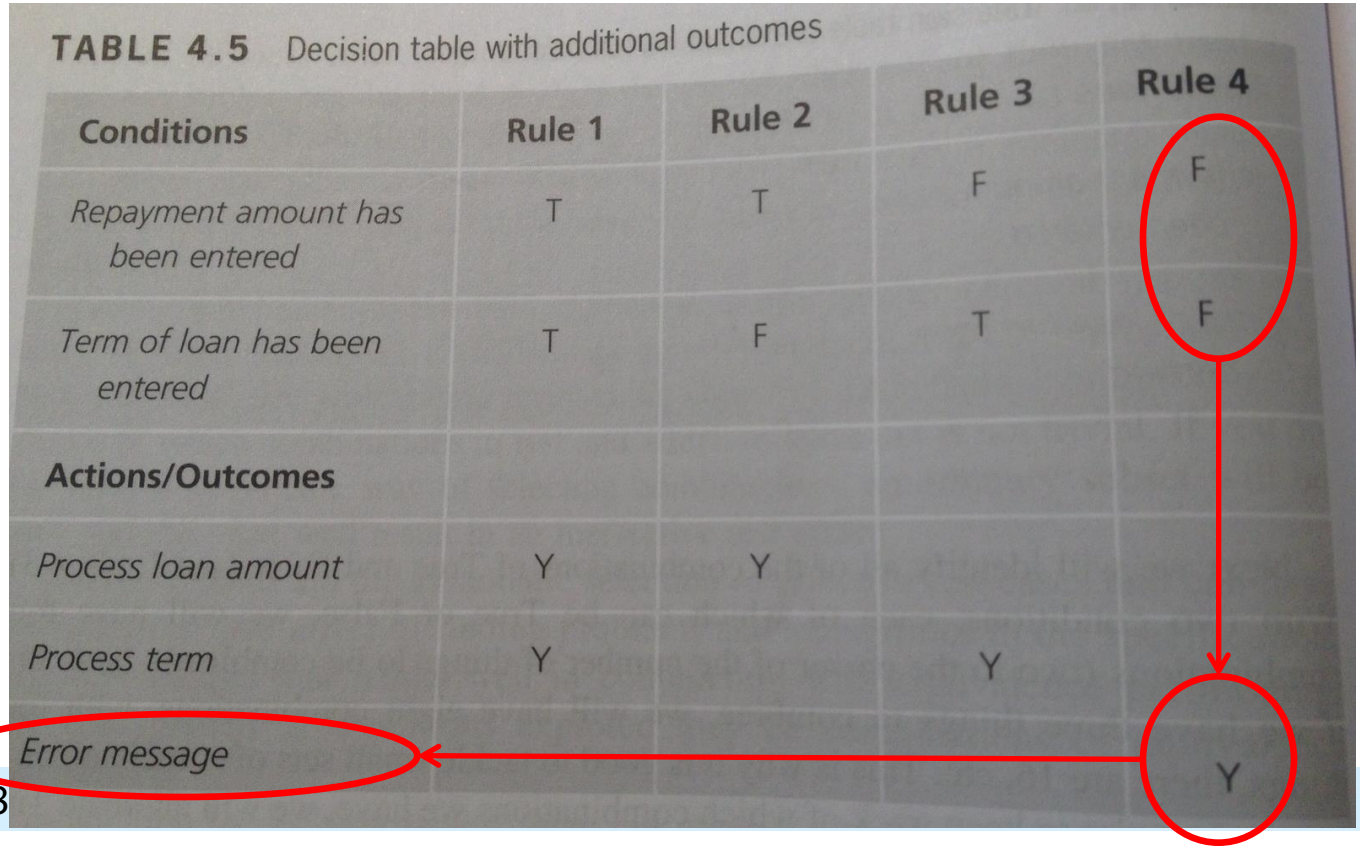
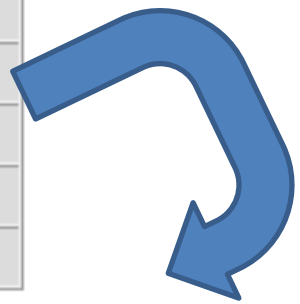


TABLE 4.5 Decision table with additional outcomes

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	
Error message				Y

If actions are mutually exclusive, we can list actions in one row

TABLE 4.6 Decision table with changed outcomes				
Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount		Y		
Process term			Y	
Error message	Y			Y



Different format

TABLE 4.7 Decision table with outcomes in one row				
Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Result	Error message	Process loan amount	Process term	Error message
			Y	
	Y			Y

How to Use Decision Tables in Testing

- One test for each column/rule
- Advantage?
  - We might test a combination of things otherwise not tested
- We might want to prioritize and test the most important combinations
  - Having the full table gives overview of combinations for selection

Decision Table Testing - Exercise

ATM: Withdrawal is granted if requested amount is covered by the balance or credit is allowed to cover the withdrawal amount.

Design decision table based on above business rules:

- Identify conditions
- Fill out input combinations
- Determine actions

State Transition Testing

- State-transition diagrams are used when
  - system must remember something about what has happened before
  - valid and invalid orders of operations exist

- Test cases are designed to investigate valid and invalid state transitions

States *that the software may occupy (e.g. open/closed - funded/insufficient funds);*

Transitions *from one state to another (not all transitions are allowed);*

Events *that cause a transition (closing a file or withdrawing money);*

Actions *that result from a transition (an error message or being given your cash).*

State Diagram – Example

Black 4.3.3.

- State diagram for PIN entry
  - Seven states, but only four possible events (card inserted, enter PIN, PIN OK, and PIN not OK)

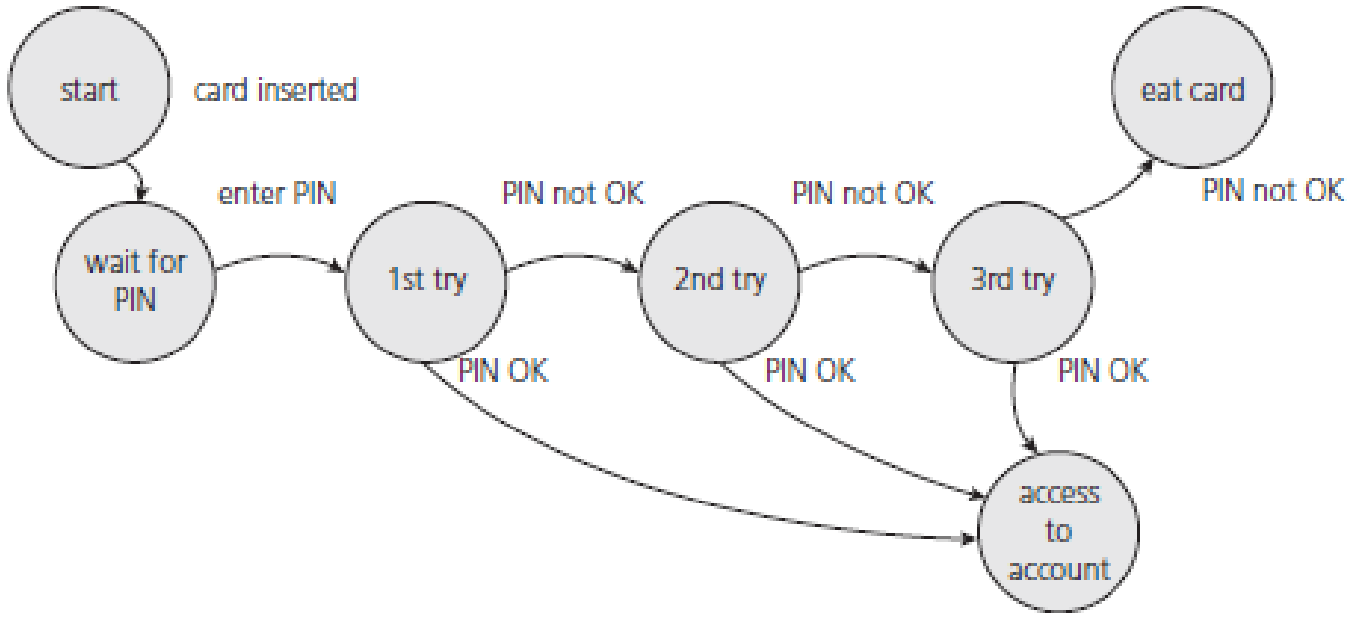


FIGURE 4.2 State diagram for PIN entry

State Table – Example Black 4.3.3.

- State diagrams may be easier to comprehend
- State tables may be easier to use in a complete and systematic manner
  - Gives a fuller overview of valid and invalid transitions (= the negative tests)

TABLE 4.9 State table for the PIN example

	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

Combination of many input values

Conventional Test Cases

Example:

- If we have three variables (A,B,C), each can have 3 values say (Red, Green, and Blue).
- The possible combinations in conventional test cases would be 27 i.e. 3^3

All Combinations for Three Variables of Three Levels Each

	A	B	C
1	Red	Red	Red
2	Red	Red	Green
3	Red	Red	Blue
4	Red	Green	Red
5	Red	Green	Green
6	Red	Green	Blue
7	Red	Blue	Red
8	Red	Blue	Green
9	Red	Blue	Blue
10	Blue	Red	Red
11	Blue	Red	Green
12	Blue	Red	Blue
13	Blue	Green	Red
14	Blue	Green	Green
15	Blue	Green	Blue
16	Blue	Blue	Red
17	Blue	Blue	Green
18	Blue	Blue	Blue
19	Green	Red	Red
20	Green	Red	Green
21	Green	Red	Blue
22	Green	Green	Red
23	Green	Green	Green
24	Green	Green	Blue
25	Green	Blue	Red
26	Green	Blue	Green
27	Green	Blue	Blue

Source: https://www.slideshare.net/princebhanwra/orthogonal-array-testing

Pairwise testing

- Systematic way to test pairwise interaction

Example:

- If we have three variables (A,B,C), each can have 3 values say (Red, Green, and Blue).
- The possible combinations in OATS test cases would be 9.

All-Pairs Array, Three Variables of Three Levels Each

	A	B	C
2	Red	Red	Green
4	Red	Green	Red
9	Red	Blue	Blue
12	Blue	Red	Blue
14	Blue	Green	Green
16	Blue	Blue	Red
19	Green	Red	Red
24	Green	Green	Blue
26	Green	Blue	Green

OAT = Orthogonal Array Testing

Pairwise testing advantagers

Guarantees testing the pair-wise combinations of all the selected variables.

Creates an efficient and concise test set with many fewer test cases than testing all combinations of all variables.

Creates a test set that has an even distribution of all pair-wise combinations.

Exercises some of the complex combinations of all the variables.

Is simpler to generate and less error prone than test sets created by hand.

Available tools & articles

- We will look more at this technique later
- http://www.pairwise.org/tools.asp
- http://pairwisetesting.com/tools.html
- http://pairwisetesting.com/articles/

White Box Techniques

[Black] sec. 4.4

Painful results from accidently leaving gaps in coverage

White-Box Techniques – Test Coverage

- Test coverage can be measured based on a number of different structural elements in a system or component

- The percentage to which a specified coverage item has been exercised by a test suite

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

Code coverage

- Statement coverage
- Decision coverage

- White box testing generally requires *better programming skills* than black box testing

- Code coverage techniques are best used on areas of software code where more thorough testing is required, e.g.
  - Safety-critical code, vital code, complex code

- Measurement of code coverage can be supported by *tools*

Statement Coverage 1

Also known as line coverage → Covers the true conditions

$$\text{Statement Coverage} = \frac{\text{No. of statements exercised}}{\text{Total no. of statements}} \times 100\ \%$$

For 100 % statement coverage, test cases must be designed such that all statements in the program are traversed at least once

Statement Coverage 2

Black code sample 4.1

A = 12, B = 10 gives 100 % statement coverage

```
READ A
READ B
IF A > B THEN C = 0
ENDIF
```

Statement Coverage 3

Black code sample 4.2

A = 2, B = 3 gives 83 % statement coverage (if all lines in pseudo example are considered to be statements)

```
READ A
READ B
C = A + 2 * B
IF C > 50 THEN
   PRINT 'Large C'
ENDIF
```

A = 20, B = 25 gives 100 % statement coverage

Decision Coverage

$$\text{Decision Coverage} = \frac{\text{No. of decisions exercised}}{\text{Total no. of decisions}} \times 100\ \%$$

- Is a stronger logic-coverage criterion where both True and False outcome for each decision must be covered in a test case

Decision Coverage

Black code sample 4.3

A = 20, B = 15 gives 100 % statement coverage

```
READ A
READ B
C = A - 2 * B
IF C < 0 THEN
   PRINT 'C negative'
ENDIF
```

A = 20, B = 15 **plus**
A = 10, B = 2 gives 100 % decision coverage

Coverage Items - Examples

- Typically code, but could be …
  - requirements, menu options or screens (system level)
  - interfaces (integration testing)
  - EP: percentage of equivalence partitions exercised (we could measure valid and invalid partition coverage separately if this makes sense).
  - BVA: percentage of boundaries exercised
  - Decision tables: percentage of business rules or decision table columns tested.
  - State transition testing: there are a number of possible coverage measures:
    - Percentage of states visited
    - Percentage of (valid) transitions exercised
    - …

Choosing Test Techniques

- There are no best technique

- Specification-based testing can find missing things in the code (i.e. missing requirements)
- Structure-based can only test what is there already, i.e. test the quality of the existing code – including not used code
- Experience-based testing can find things missing in the specification AND in the code