Software Testing Foundation Level

Chapter 4: Test Design Techniques

刘琴 Qin Liu

Use Cases/Scenario Tests

- Design various cases that reflect challenging real-world uses of the product
- For example, a home equity loan application
  - “John and Jenny Stevens have three kids…”
  - “…a house worth $400K on which they owe $350K…”
  - “… two cars worth $25K on which they owe $17K…”
  - “…incomes of $45K and 75K respectively…”
  - “… one late payment on their Visa card and one late car payment, seventeen months and thirty-five months ago, respectively…”
  - “…and they apply for a 15K home equity loan.”
- Some object-oriented design methodologies include use cases, so this can be an easy source of tests

Use Cases and Boundaries

- Revisiting our boundary conditions, do reasonable usage conditions affect boundary conditions?
- If we have a four-byte unsigned item counter variable, do we want to allow ordering 32K items just because the software supports it?
- Do we allow departure dates after arrival dates for trips?
- Is accepting a “ridiculous” input a bug?
- What do you think?

Decision Tables

- Business rules can often be specified compactly in decision tables
  - Deciding how to process an order based on size, stock on hand, state to which to ship, and so forth is often in business rules
  - These can be shown as flow-charts or as tables
  - The decision tables can make for instant test cases
- Let’s take a look at an example

ATM Decision Table

Condition	1	2	3	4	5
Valid Card	N	Y	Y	Y	Y
Valid PIN	-	N	N	Y	Y
Invalid PIN=3	-	N	Y	N	N
Balance OK	-	-	-	N	Y
Action					
Reject Card	Y	N	N	N	N
Reenter PIN	N	Y	N	N	N
Keep Card	N	N	Y	N	N
Reenter Request	N	N	N	Y	N
Dispense Cash	N	N	N	N	Y

This decision table shows the business logic for an ATM. Notice that the dashes "-" indicate conditions that aren't reached as part of this rule. The rules are mutually exclusive, in that only one rule can apply at any one moment of time.

Notice that the business logic layer is usually under the user interface layer, so at this point the basic sanity checking of the inputs should have been done.

A More Complex Decision Table

- Some police departments have handheld computers that take credit cards for moving violations
- A decision table could be used to design and test this system
- The conditions that determine the actions to take are shown in the decision table on the next page
- You might want to use boundary value analysis to adequately cover the rules
- What would you do about interaction of rules?

Police System Decision Table

Condition	1	2	3	4	5	6	7	8
License OK	No	-	-	-	-	-	-	-
Warrant	-	Yes	-	-	-	-	-	-
Registration OK	-	-	No	-	-	-	-	-
Vehicle OK	-	-	-	No	-	-	-	-
Excess Speed	-	-	-	-	1-10	11-20	21-25	>25
Action								
Arrest	Yes	Yes	-	-	-	-	-	Yes
Fix-It Ticket	-	-	Yes	Yes	-	-	-	-
Warning	-	-	-	-	Yes	-	-	-
Fine	+250	+250	+25	+25	+0	+75	+150	+250

Chapter 4: Test Design Techniques

Section 3: Exercise 2

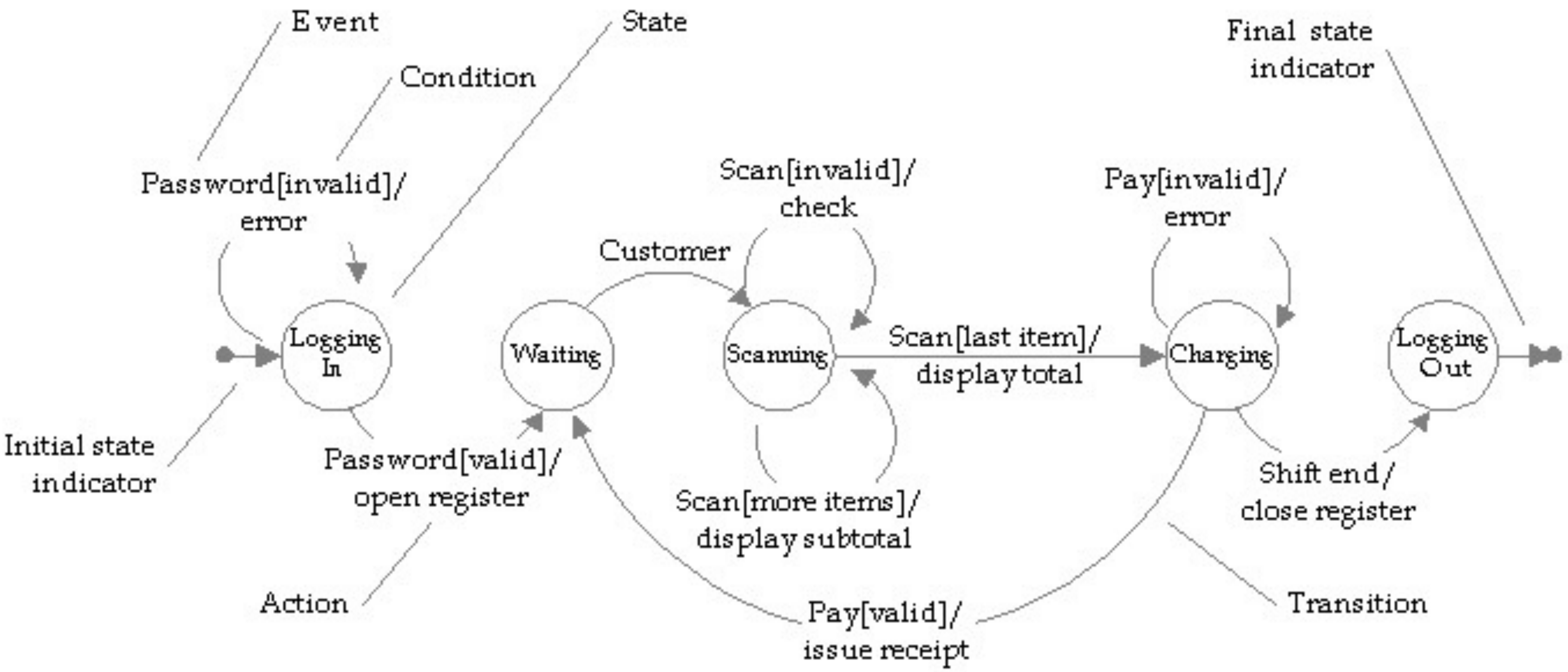
A2.1.8 : Decision Tables

- Refer to the payment decision table in the Omninet System Requirements Document.
- Develop test scenarios which adequately cover the decision table shown.
- How did you determine the level of coverage required? What implementation assumptions might change the number of tests needed?
- Discuss.

Finite-State Models

- Understand the various states the system has, including any which are initial and final
- Identify transitions, events, conditions, and actions in each state
- Use a graph or table to model the system and serve as an oracle
- For each event and condition, verify action and next state

State-Transition Diagram



A state-transition diagram for a point-of-sales system, from cashier's point of view. How would it look from the customer's point of view? From the system's? Do you see the bug?

State-Transition Tables

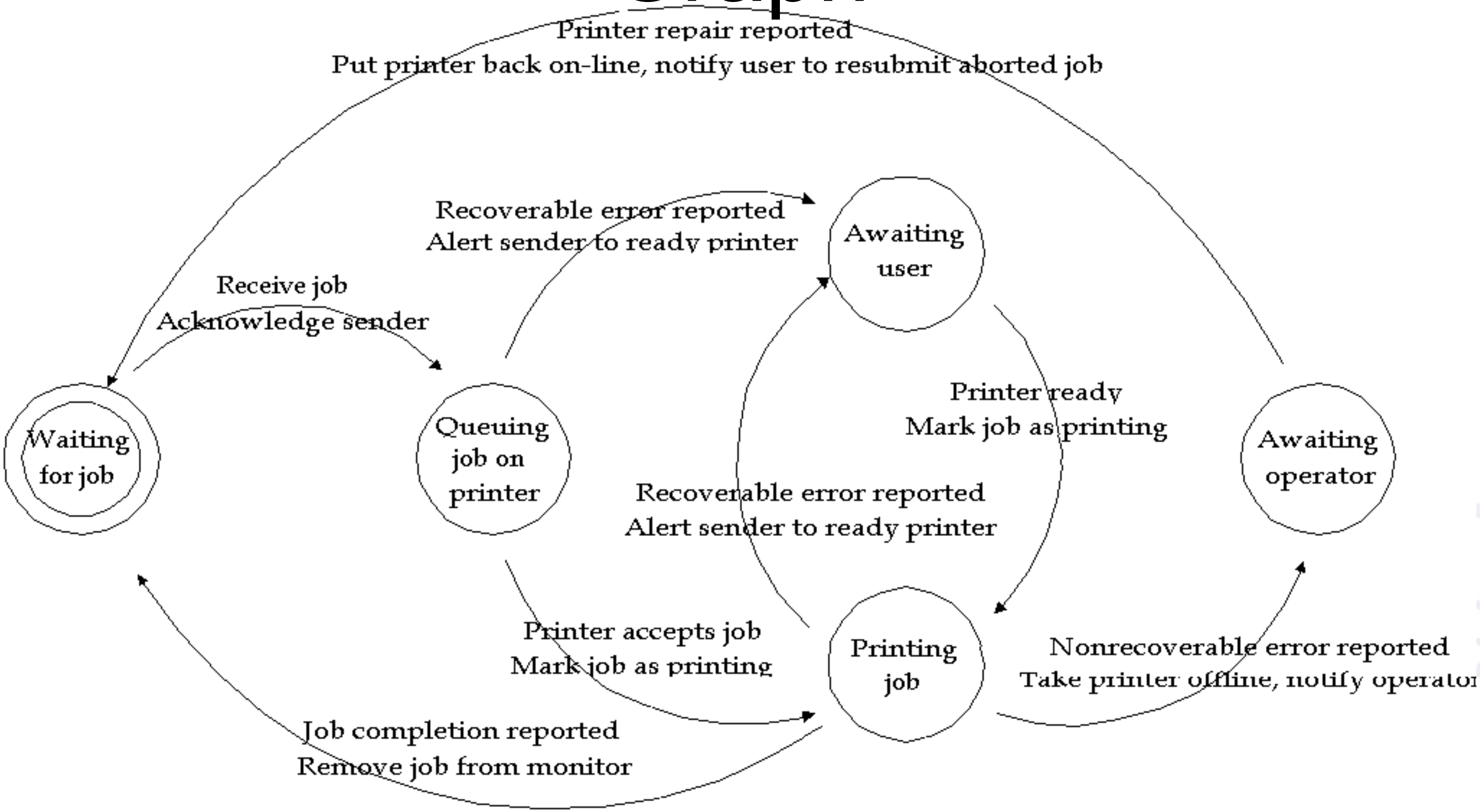
Current State	**Event[Condition]**	**Action**	**New State**
Logging In	Password[invalid]	Error	Logging In
Logging In	Password[valid]	Open register	Waiting
Logging In	Customer	[Undefined]	[Undefined]
Logging In	Scan[any]	[Undefined]	[Undefined]
Logging In	Pay[any]	[Undefined]	[Undefined]
Logging In	Shift end	[Undefined]	[Undefined]

A state-transition table can represent complex state-transitions that won't fit on a graph. (However, complexity might indicate bad design.) It can also reveal undefined situation, as does this portion of the table for the graph on the preceding page.

Finite-State Models

- Let’s look at another example
- Print server (graph next page) can be:
  - Awaiting job
  - Queuing job
  - Printing job
  - Awaiting user intervention
  - Awaiting operator intervention
- Printer server responds to events (user or printer inputs) based on state

Print Server State Machine Graph



Chapter 4: Test Design Techniques

Section 3: Exercise 3

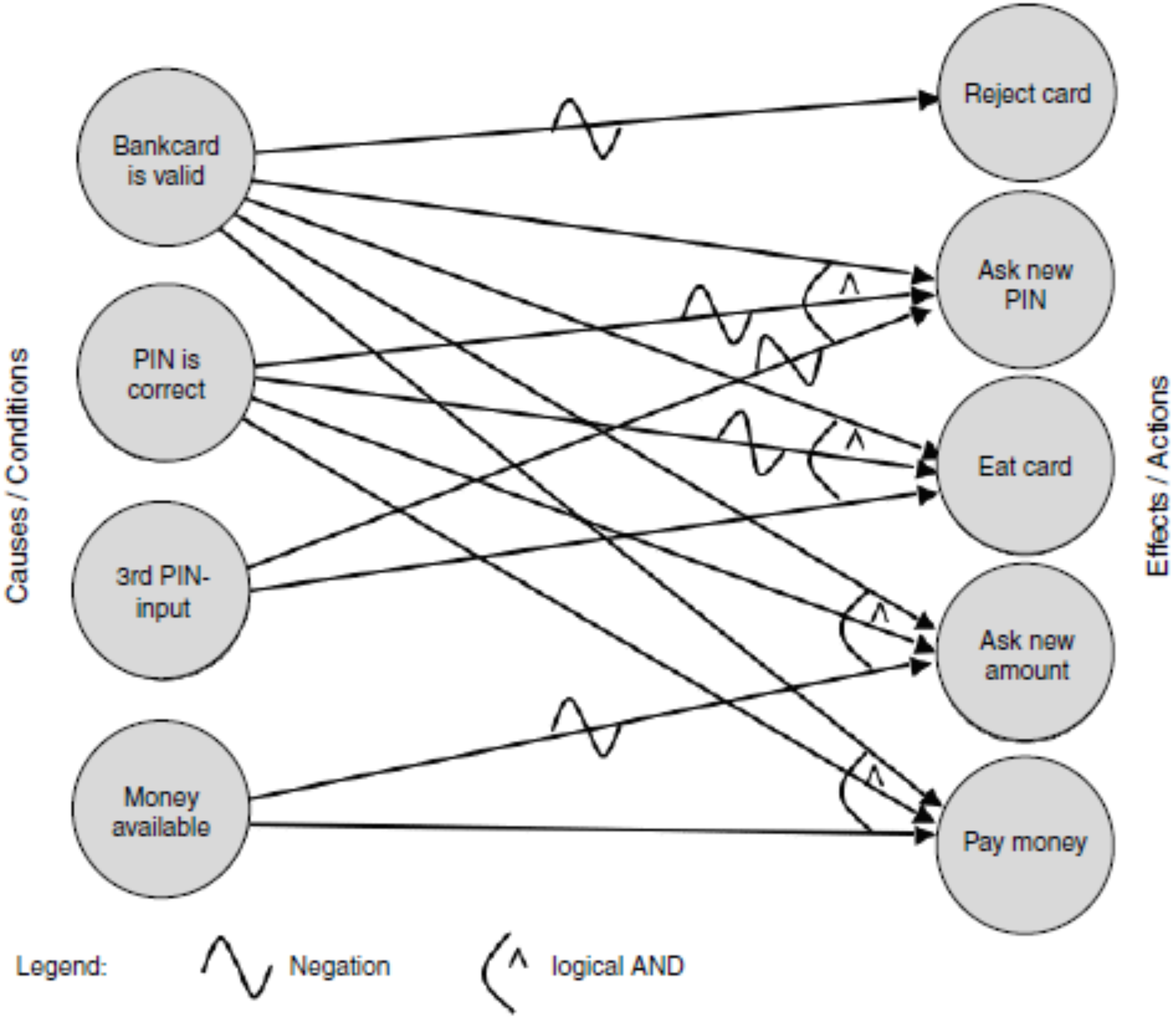
A2.1.9 : Kiosk States

- The Omninet public Internet access kiosks can be in various states, based on receipt of payment, active sessions, and so forth.
- Refer to the Marketing Requirements Document and the System Requirements Document for Omninet
- Draw a state diagram for the kiosk.
- Cover the state diagram with test cases.
- Discuss.

Logic Based Testing（LBT）

- Previously introduced techniques treat different input data independently
  - No dependency is explicitly considered
- Cause-Effect Graphing
  - Using dependencies for deriving test cases
- Precondition: possible to find causes and effects from specification
- Cause: condition, consisting of input values
- Conditions are connected with logical operators

LBT Example: Cause-Effect Graph for an ATM



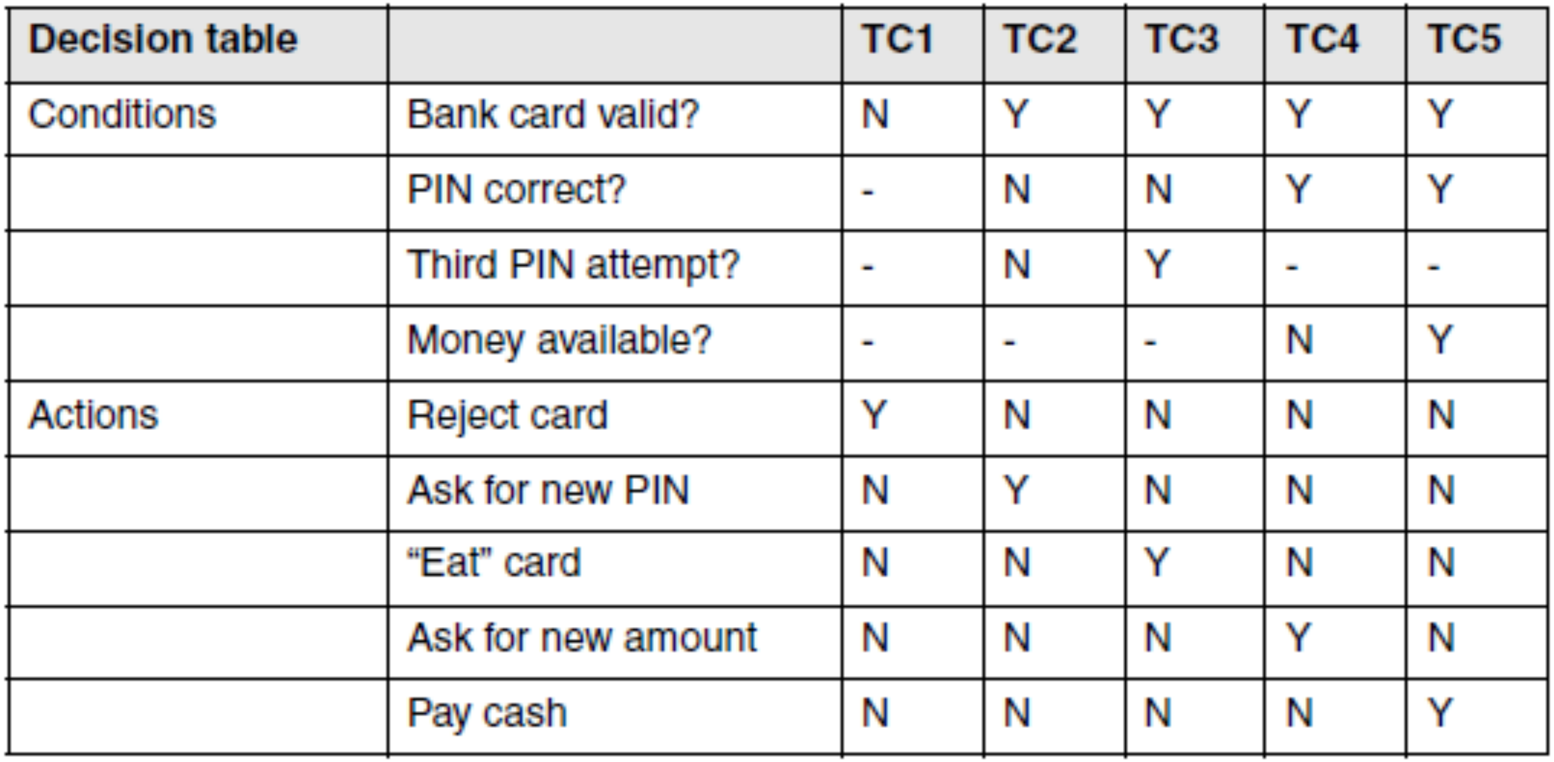
LBT: Derive Decision Table

- Graph transformed into decision table, for deriving test cases

  1. Choose an effect.
  2. Looking in the graph, find combinations of causes that have this effect and combinations that do not have this effect.
  3. Add one column into the table for every one of these cause-effect combinations. Include the caused states of the remaining effects.
  4. Check to see if decision table entries occur several times, and if they do, delete them.

LBT: Derive Decision Table

- Not all possible combinations, but only interesting ones

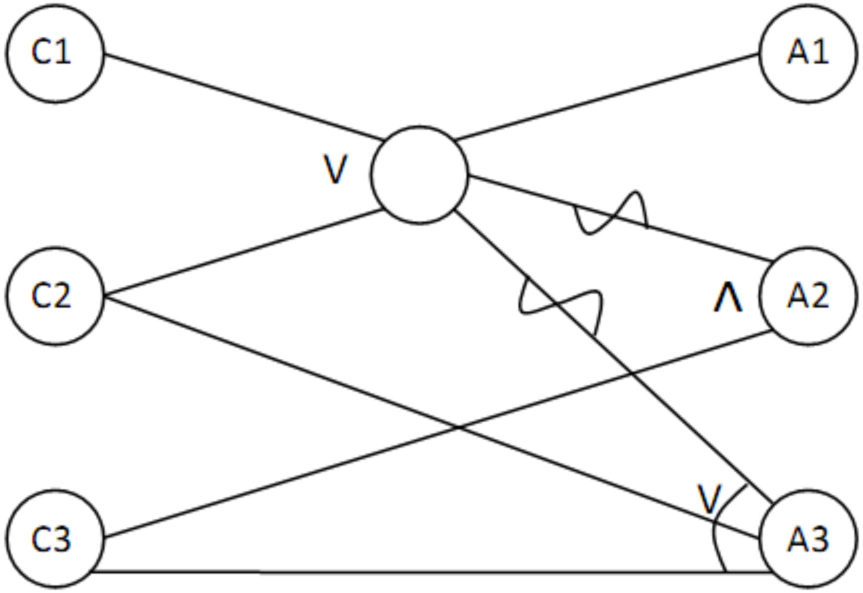


Decision table		TC1	TC2	TC3	TC4	TC5
Conditions	Bank card valid?	N	Y	Y	Y	Y
	PIN correct?	-	N	N	Y	Y
	Third PIN attempt?	-	N	Y	-	-
	Money available?	-	-	-	N	Y
Actions	Reject card	Y	N	N	N	N
	Ask for new PIN	N	Y	N	N	N
	“Eat” card	N	N	Y	N	N
	Ask for new amount	N	N	N	Y	N
	Pay cash	N	N	N	N	Y

LBT: Derive Decision Table (cont.)

- Objective of test based on decision tables: executing “interesting” combinations of inputs
- Upper half: inputs (causes)
- Lower half: effects
- Each cause and effect should at least have values “YES” and “NO” once
- Each column is a rule, which can derive test cases
- Test exit criteria
  - Minimum requirement: every column is executed by at least one test case

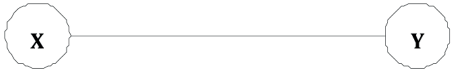
LBT Example: Cheque debit function



Cause-effect graph of the cheque debit function (see below for notation)

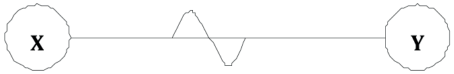
LBT Example: Cheque debit function

Identity



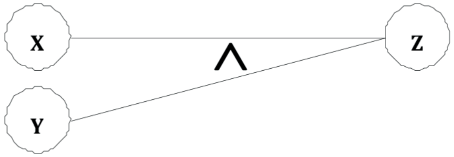
Node Y is true only if X is true
If X = T then Y = T else Y = F

Not



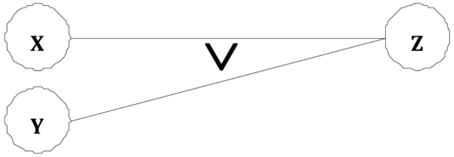
Node Y is true only if X is false
If X = F then Y = T else Y = F

And



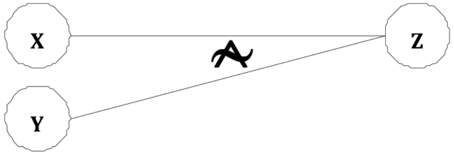
Node Z is true only if both X and Y are true
If X = T and Y = T then Z = T else Z = F

Or



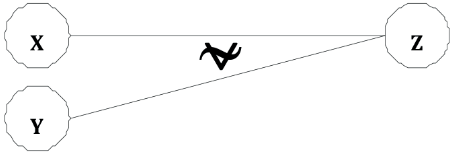
Node Z is true only if either X or Y are true
If X = T or Y = T then Z = T else Z = F

Nand



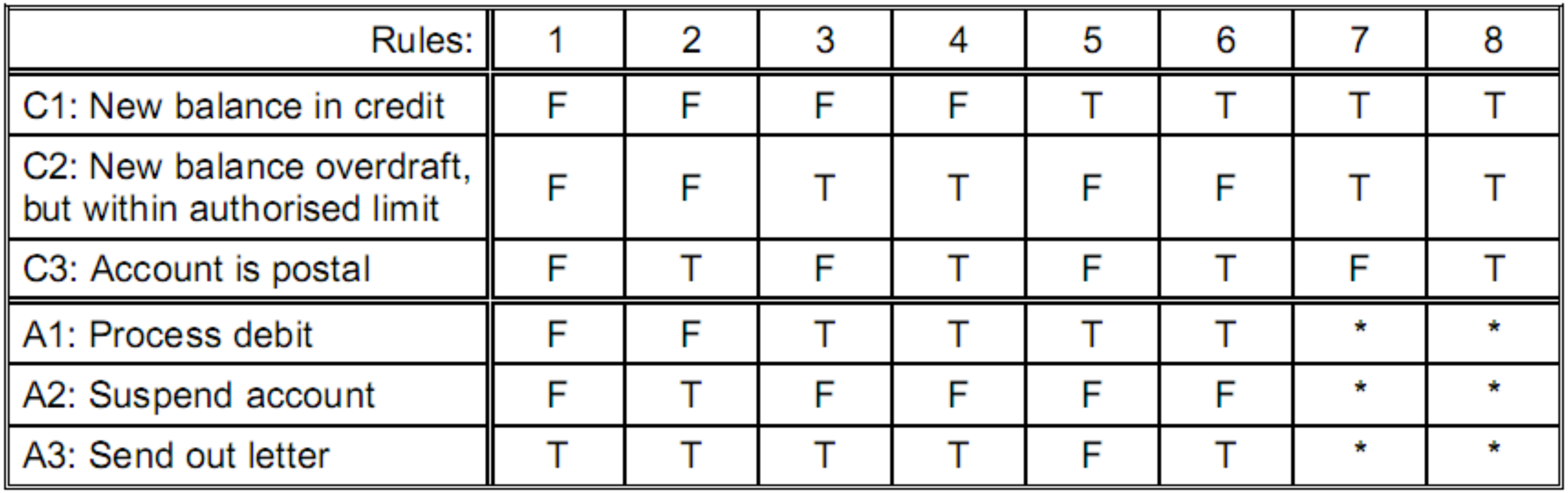
Node Z is true only if either X or Y or both are false
If X = F or Y = F then Z = F else Z = T

Nor



Node Z is true only if neither X nor Y are true
If X = T or Y = T then Z = F else Z = T

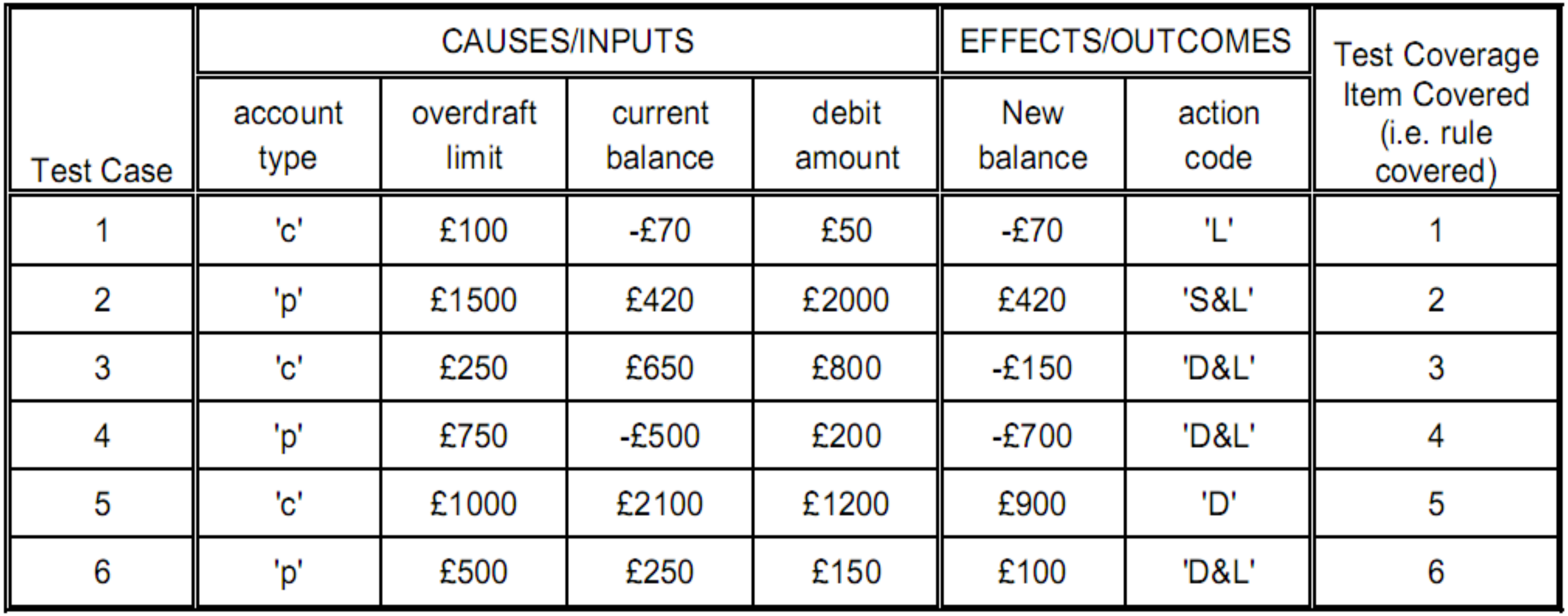
LBT: Derive DT



Rules:	1	2	3	4	5	6	7	8
C1: New balance in credit	F	F	F	F	T	T	T	T
C2: New balance overdraft, but within authorised limit	F	F	T	T	F	F	T	T
C3: Account is postal	F	T	F	T	F	T	F	T
A1: Process debit	F	F	T	T	T	T	*	*
A2: Suspend account	F	T	F	F	F	F	*	*
A3: Send out letter	T	T	T	T	F	T	*	*

Decision table of the cheque debit function

LBT Example: Derive Test Cases



Test Case	CAUSES/INPUTS				EFFECTS/OUTCOMES		Test Coverage Item Covered (i.e. rule covered)
	account type	overdraft limit	current balance	debit amount	New balance	action code	
1	'c'	£100	-£70	£50	-£70	'L'	1
2	'p'	£1500	£420	£2000	£420	'S&L'	2
3	'c'	£250	£650	£800	-£150	'D&L'	3
4	'p'	£750	-£500	£200	-£700	'D&L'	4
5	'c'	£1000	£2100	£1200	£900	'D'	5
6	'p'	£500	£250	£150	£100	'D&L'	6

Test case table of the cheque debit function

LBT: Discussion

- Could "Logic based analysis/testing, e.g.,Cause-Effect Graph" be helpful for designing A2.1.12, Webpage?
- Create functional tests for accepting orders
  - The system accepts a five-digit numeric item ID number from 00000 to 99999 (integer / string?)
  - Item IDs are sorted by price, with the cheapest items having the lower (close to 00000) item ID numbers and the most expensive items having the higher (close to 99999) item ID numbers
  - The system accepts a quantity to be ordered, from 1 to 99
  - If the user enters a previously-ordered item ID and a 0 quantity to be ordered, that item is removed from the shopping cart
  - The maximum total order is $999.99

Extension: Combinatorial Test Techniques (CTT)

- Consider the test basis for a test item travel_preference, which records the travel preferences of staff members of an Australian organisation that travel to major Australian capital cities for work purposes. Each set of travel preferences is chosen through a series of radio buttons, which consist of the following input value choices:
- Destination = Paris, London, Sydney
- Class = First, Business, Economy
- Seat = Aisle, Window

- If a combination of correct inputs is provided to the program, it will output "Accept" otherwise it will output
- "Reject".

When calculating coverage for any technique, the following formula shall be used:

$C=\left(N/T\times100\right)\%$ *where:*

- *— C is the coverage achieved by a specific test design technique;*
- *— N is the number of test coverage items covered by executed test cases;*
- *— T is the total number of test coverage items identified by the test design technique.*

CTT Example

FS1: travel_preference function

Every combinatorial technique shares a common approach to deriving test conditions. That is, test conditions correspond to each parameter (P) of the test item taking on one specific value (V), resulting in one P-V pair.

This is repeated until all parameters are paired with their corresponding values.

For the example above, this results in the following P-V pairs:

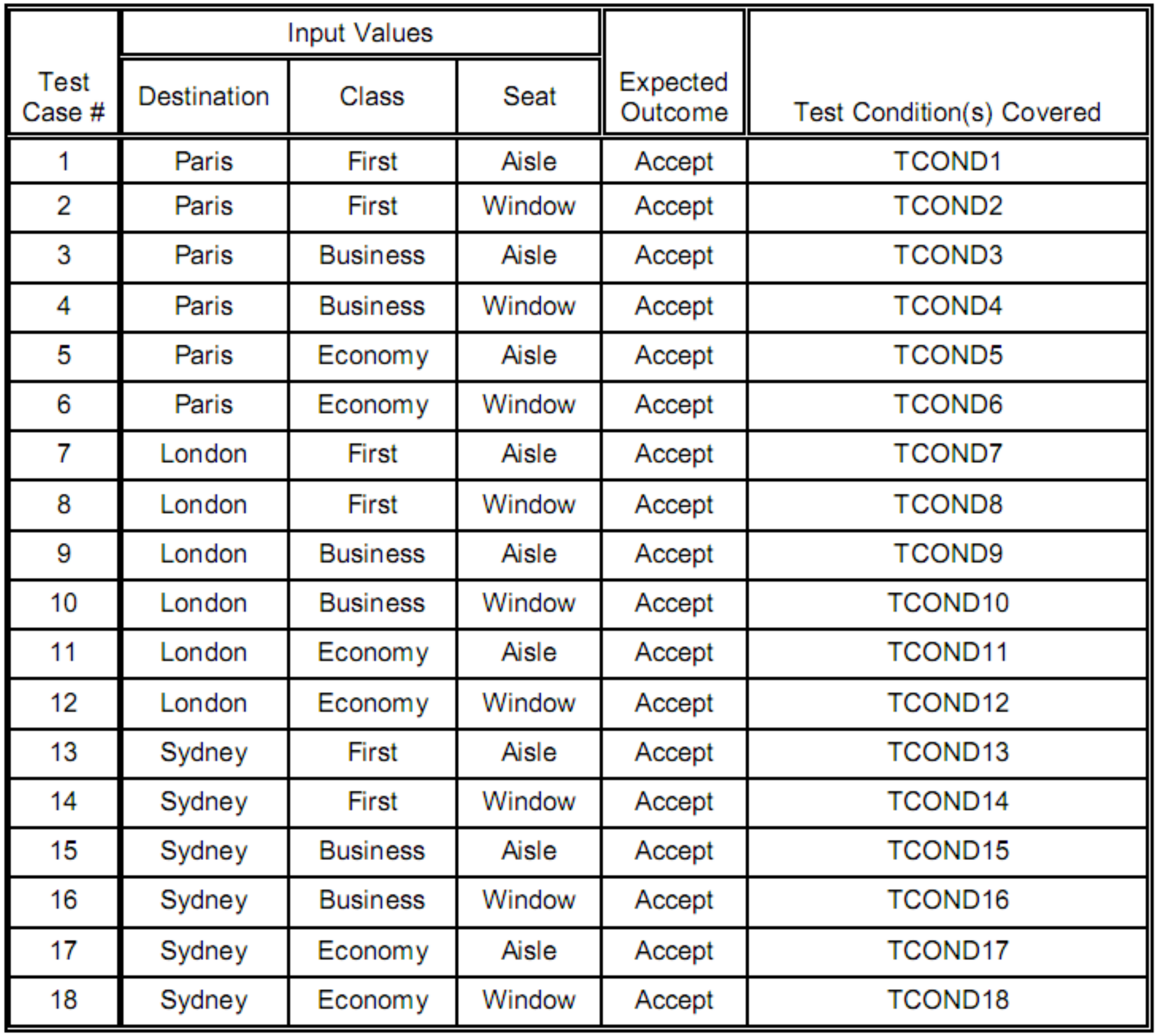
- PV1: Destination – Paris
- PV2: Destination – London
- PV3: Destination – Sydney
- PV4: Class – First
- PV5: Class – Business
- PV6: Class – Economy
- PV7: Seat – Aisle
- PV8: Seat – Window

CTT Example: All Combinations

*In all combinations testing, the test coverage items are **the unique combinations of P-V pairs**, made up of one P-V pair for each test item parameter.*

TCOVER1:	Destination – Paris,	Class – First,	Seat – Aisle	(for PV 1, 4, 7)
TCOVER2:	Destination – Paris,	Class – First,	Seat – Window	(for PV 1, 4, 8)
TCOVER3:	Destination – Paris,	Class – Business,	Seat – Aisle	(for PV 1, 5, 7)
TCOVER4:	Destination – Paris,	Class – Business,	Seat – Window	(for PV 1, 5, 8)
TCOVER5:	Destination – Paris,	Class – Economy,	Seat – Aisle	(for PV 1, 6, 7)
TCOVER6:	Destination – Paris,	Class – Economy,	Seat – Window	(for PV 1, 6, 8)
TCOVER7:	Destination – London,	Class – First,	Seat – Aisle	(for PV 2, 4, 7)
TCOVER8:	Destination – London,	Class – First,	Seat – Window	(for PV 2, 4, 8)
TCOVER9:	Destination – London,	Class – Business,	Seat – Aisle	(for PV 2, 5, 7)
TCOVER10:	Destination – London,	Class – Business,	Seat – Window	(for PV 2, 5, 8)
TCOVER11:	Destination – London,	Class – Economy,	Seat – Aisle	(for PV 2, 6, 7)
TCOVER12:	Destination – London,	Class – Economy,	Seat – Window	(for PV 2, 6, 8)
TCOVER13:	Destination – Sydney,	Class – First,	Seat – Aisle	(for PV 3, 4, 7)
TCOVER14:	Destination – Sydney,	Class – First,	Seat – Window	(for PV 3, 4, 8)
TCOVER15:	Destination – Sydney,	Class – Business,	Seat – Aisle	(for PV 3, 5, 7)
TCOVER16:	Destination – Sydney,	Class – Business,	Seat – Window	(for PV 3, 5, 8)
TCOVER17:	Destination – Sydney,	Class – Economy,	Seat – Aisle	(for PV 3, 6, 7)
TCOVER18:	Destination – Sydney,	Class – Economy,	Seat – Window	(for PV 3, 6, 8)

CTT Example: All Combinations



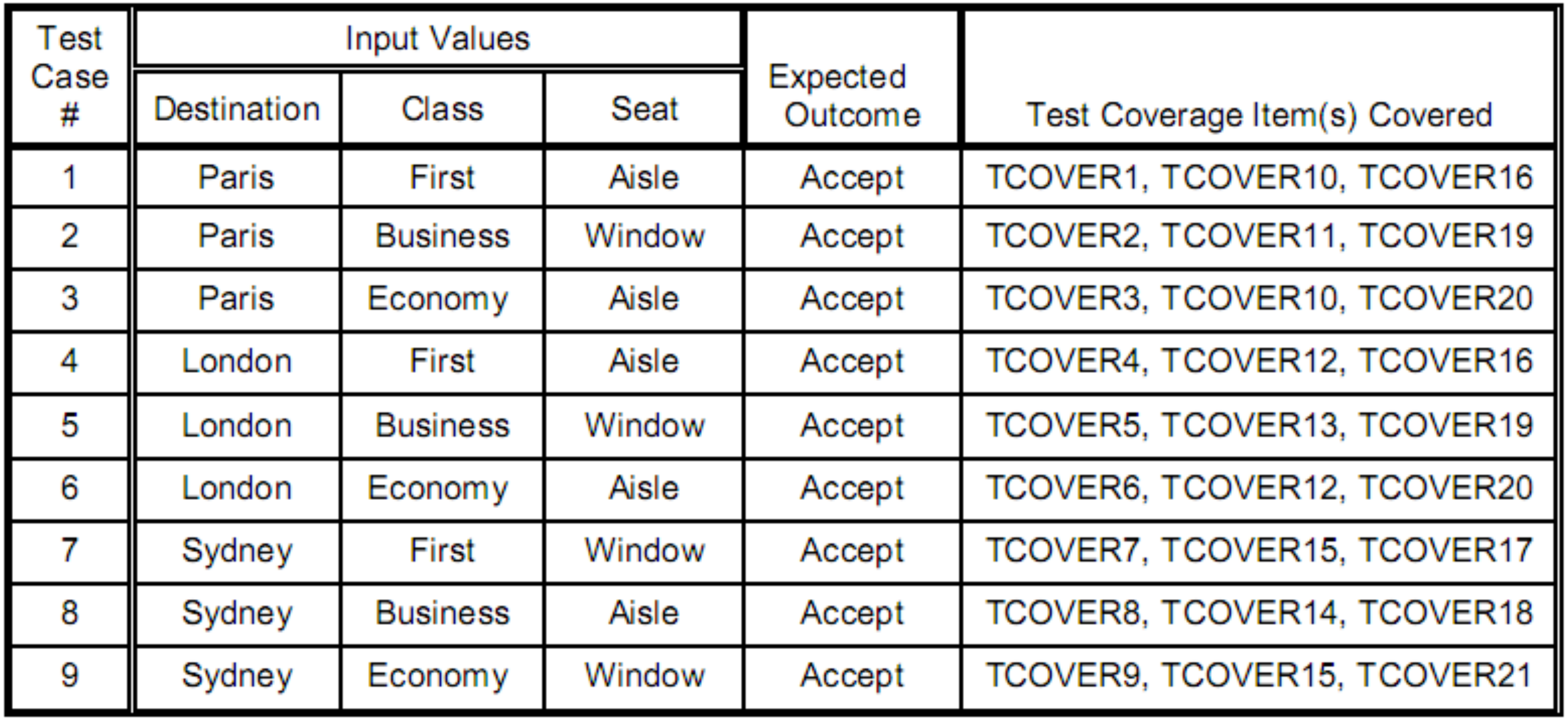
Test Case #	Input Values			Expected Outcome	Test Condition(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOND1
2	Paris	First	Window	Accept	TCOND2
3	Paris	Business	Aisle	Accept	TCOND3
4	Paris	Business	Window	Accept	TCOND4
5	Paris	Economy	Aisle	Accept	TCOND5
6	Paris	Economy	Window	Accept	TCOND6
7	London	First	Aisle	Accept	TCOND7
8	London	First	Window	Accept	TCOND8
9	London	Business	Aisle	Accept	TCOND9
10	London	Business	Window	Accept	TCOND10
11	London	Economy	Aisle	Accept	TCOND11
12	London	Economy	Window	Accept	TCOND12
13	Sydney	First	Aisle	Accept	TCOND13
14	Sydney	First	Window	Accept	TCOND14
15	Sydney	Business	Aisle	Accept	TCOND15
16	Sydney	Business	Window	Accept	TCOND16
17	Sydney	Economy	Aisle	Accept	TCOND17
18	Sydney	Economy	Window	Accept	TCOND18

CTT Example: Pair-Wise testing

TCOVER1:	Paris, First	(for PV1, PV4)
TCOVER2:	Paris, Business	(for PV1, PV5)
TCOVER3:	Paris, Economy	(for PV1, PV6)
TCOVER4:	London, First	(for PV2, PV4)
TCOVER5:	London, Business	(for PV2, PV5)
TCOVER6:	London, Economy	(for PV2, PV6)
TCOVER7:	Sydney, First	(for PV3, PV4)
TCOVER8:	Sydney, Business	(for PV3, PV5)
TCOVER9:	Sydney, Economy	(for PV3, PV6)
TCOVER10:	Paris, Aisle	(for PV1, PV7)
TCOVER11:	Paris, Window	(for PV1, PV8)
TCOVER12:	London, Aisle	(for PV2, PV7)
TCOVER13:	London, Window	(for PV2, PV8)
TCOVER14:	Sydney, Aisle	(for PV3, PV7)
TCOVER15:	Sydney, Window	(for PV3, PV8)
TCOVER16:	First, Aisle	(for PV4, PV7)
TCOVER17:	First, Window	(for PV4, PV8)
TCOVER18:	Business, Aisle	(for PV5, PV7)
TCOVER19:	Business, Window	(for PV5, PV8)
TCOVER20:	Economy, Aisle	(for PV6, PV7)
TCOVER21:	Economy, Window	(for PV6, PV8)

*In pair-wise testing, test coverage items are identified as **the unique pairs of P-V pairs** for different parameters. For the travel_preference example, the following test coverage items can be defined:*

CTT Example: Pair-Wise testing

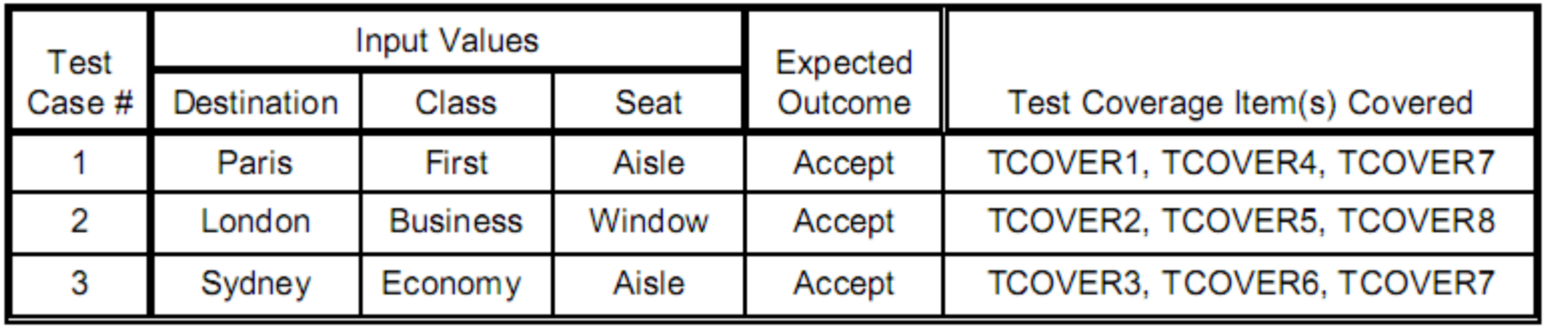


Test Case #	Input Values			Expected Outcome	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOVER1, TCOVER10, TCOVER16
2	Paris	Business	Window	Accept	TCOVER2, TCOVER11, TCOVER19
3	Paris	Economy	Aisle	Accept	TCOVER3, TCOVER10, TCOVER20
4	London	First	Aisle	Accept	TCOVER4, TCOVER12, TCOVER16
5	London	Business	Window	Accept	TCOVER5, TCOVER13, TCOVER19
6	London	Economy	Aisle	Accept	TCOVER6, TCOVER12, TCOVER20
7	Sydney	First	Window	Accept	TCOVER7, TCOVER15, TCOVER17
8	Sydney	Business	Aisle	Accept	TCOVER8, TCOVER14, TCOVER18
9	Sydney	Economy	Window	Accept	TCOVER9, TCOVER15, TCOVER21

$C=\left(N/T \times 100\right)\%$ where: Coverage for pair-wise testing shall be calculated using the following definitions:

- *— $C_{pair\text{-}wise}$ is pair-wise coverage;*
- *— N is the number of **unique pairs of P-V pairs** covered by executed test cases;*
- *— T is the total number of unique pairs of P-V pairs.*

CTT Example: Each Choice testing



Test Case #	Input Values			Expected Outcome	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	Paris	First	Aisle	Accept	TCOVER1, TCOVER4, TCOVER7
2	London	Business	Window	Accept	TCOVER2, TCOVER5, TCOVER8
3	Sydney	Economy	Aisle	Accept	TCOVER3, TCOVER6, TCOVER7

TCOVER1:	Destination – Paris	(for PV1)
TCOVER2:	Destination – London	(for PV2)
TCOVER3:	Destination – Sydney	(for PV3)
TCOVER4:	Class – First	(for PV4)
TCOVER5:	Class – Business	(for PV5)
TCOVER6:	Class – Economy	(for PV6)
TCOVER7:	Seat – Aisle	(for PV7)
TCOVER8:	Seat – Window	(for PV8)

C=(N/T ×100)% where:

Coverage for each choice testing shall be calculated using the following definitions:

- *Ceach_choice is each choice coverage;*
- *N is the number of P-V pairs covered by executed test cases;*
- *T is the total number of unique P-V pairs.*

CTT Example : - Base Choice testing

Test coverage items for base choice testing are chosen by selecting a “base choice” value for each parameter. For example, the base choice can be chosen from the operational profile, from the main path in use case testing or from the test coverage items that are derived during equivalence partitioning. In this example, the operational profile may indicate that the following input values should be chosen as the base choice:

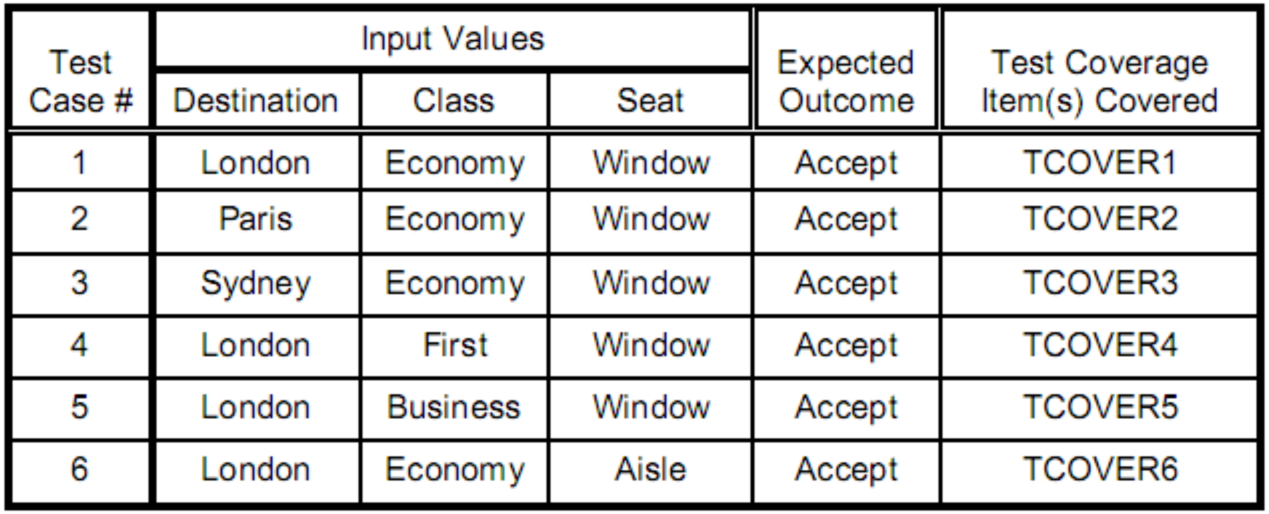
TCOVER1:	Destination – London,	Class – Economy,	Seat – Window	(PV2, PV6 & PV8)

The remaining test coverage items are derived by identifying all remaining P-V pairs:

TCOVER2	Destination – Paris,	Class – Economy,	Seat – Window	(PV1, PV6 & PV8)
TCOVER3	Destination – Sydney,	Class – Economy,	Seat – Window	(PV3, PV6 & PV8)
TCOVER4	Destination – London,	Class – First,	Seat – Window	(PV2, PV4 & PV8)
TCOVER5	Destination – London,	Class – Business,	Seat – Window	(PV2, PV5 & PV8)
TCOVER6	Destination – London,	Class – Economy,	Seat – Aisle	(PV2, PV6 & PV7)

CTT Example : - Base Choice testing

input parameter value used in 'base choice testing' that is normally selected based on being a representative or typical value for the parameter

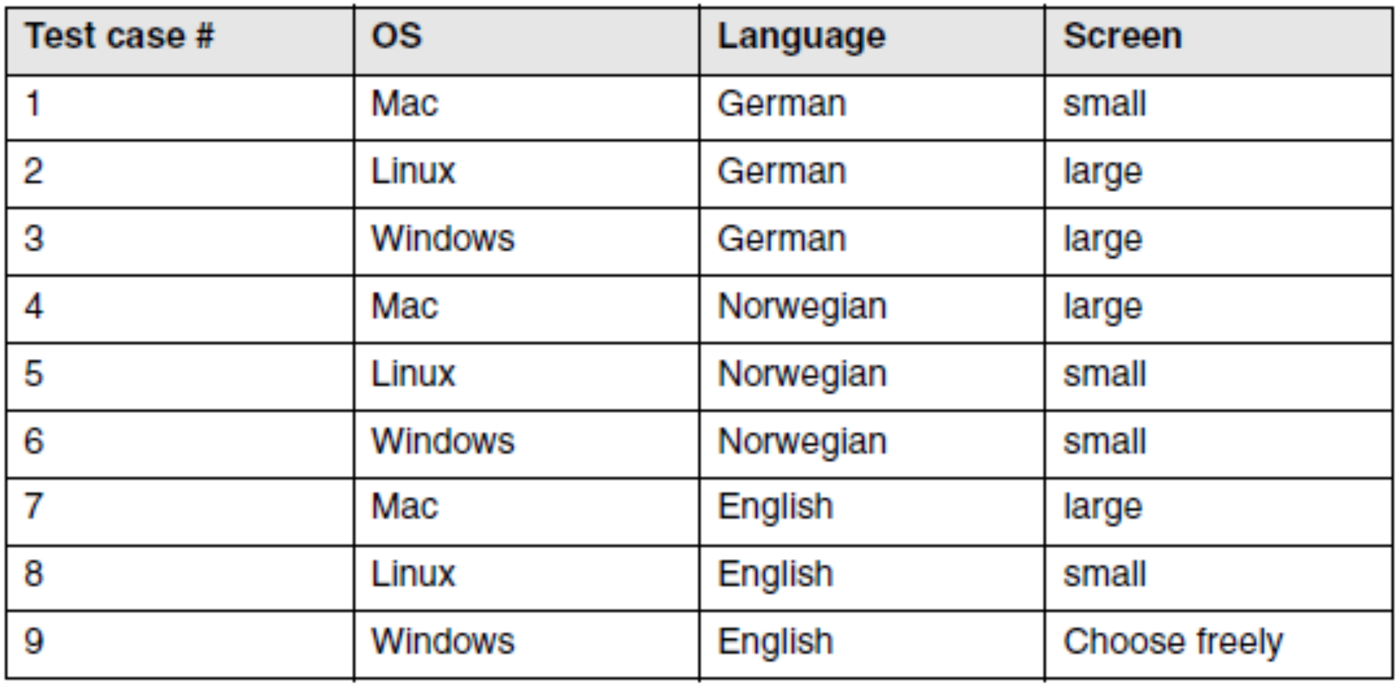


Test Case #	Input Values			Expected Outcome	Test Coverage Item(s) Covered
	Destination	Class	Seat		
1	London	Economy	Window	Accept	TCOVER1
2	Paris	Economy	Window	Accept	TCOVER2
3	Sydney	Economy	Window	Accept	TCOVER3
4	London	First	Window	Accept	TCOVER4
5	London	Business	Window	Accept	TCOVER5
6	London	Economy	Aisle	Accept	TCOVER6

CTT Discussion:

Apply Combinatorial Testing Technique

- DreamCar subsystem
  - Operating system (Mac, Linux, or Windows)
  - Language (German, Norwegian, English)
  - Screen size (small, large)



Test case #	OS	Language	Screen
1	Mac	German	small
2	Linux	German	large
3	Windows	German	large
4	Mac	Norwegian	large
5	Linux	Norwegian	small
6	Windows	Norwegian	small
7	Mac	English	large
8	Linux	English	small
9	Windows	English	Choose freely

Extension SBT: Scenario Based Testing

Consider a test item withdraw_cash that forms part of the system that drives an Automated Teller Machine (ATM), and which has the following test basis:

The withdraw_cash function allows customers with bank accounts to withdraw funds from their account via an ATM. A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM. After the withdrawal is complete, the account balance is debited by the withdrawn amount, a receipt for the withdrawal is printed, and the ATM is available and ready for the next user.

The following scenarios have been specified as being required by the customer:

The Main Scenario

□ Successful withdrawal of funds from account.

Alternate Scenarios

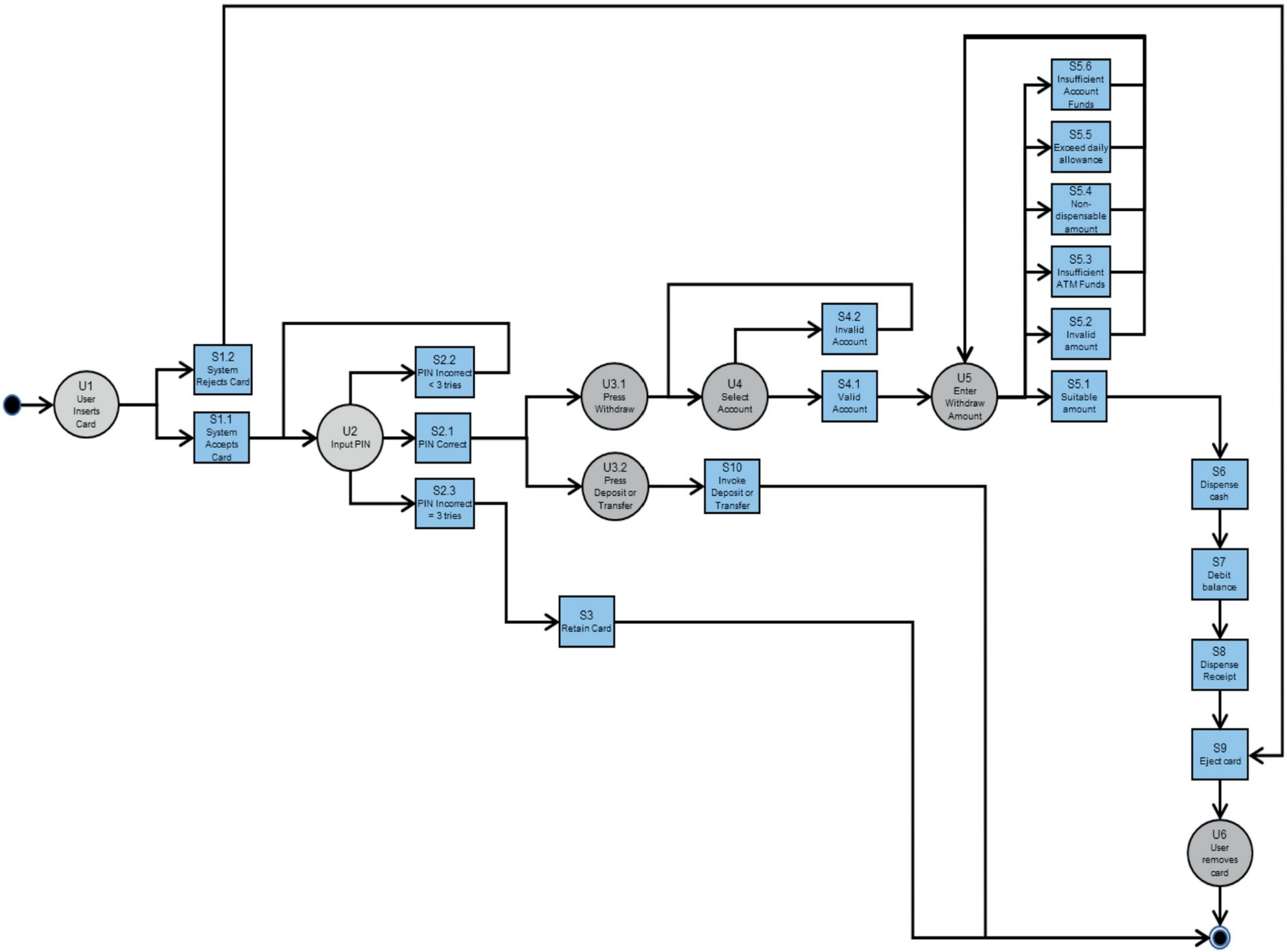
Disapproval of a withdrawal, because:

- □ the user's bank card is rejected as it is unrecognised by the ATM
- □ the user enters their PIN incorrectly up to 3 times

SBT Example: withdraw_cash function

- □ the user enters their PIN incorrectly three or more times, with the ATM retaining the card
- □ the user selects deposit or transfer instead of withdrawal
- □ the user selects an incorrect account that does not exist on the entered card
- □ the withdrawal amount entered by the user is invalid
- □ there is insufficient cash in the ATM
- □ the user enters a non-dispensable amount
- □ the user enters an amount that exceeds their daily allowance
- □ there are insufficient funds in the user’s bank account

FS1: withdraw_cash function


U1 User Inserts Card
S1.2 System Rejects Card
S1.1 System Accepts Card
U2 Input PIN
S2.2 PIN Incorrect < 3 tries
S2.1 PIN Correct
S2.3 PIN Incorrect = 3 tries
S3 Retain Card
U3.1 Press Withdraw
U3.2 Press Deposit or Transfer
S10 Invoke Deposit or Transfer
U4 Select Account
S4.2 Invalid Account
S4.1 Valid Account
U5 Enter Withdraw Amount
S5.6 Insufficient Account Funds
S5.5 Exceed daily allowance
S5.4 Non-dispensable amount
S5.3 Insufficient ATM Funds
S5.2 Invalid amount
S5.1 Suitable amount
S6 Dispense cash
S7 Debit balance
S8 Dispense Receipt
S9 Eject card
U6 User removes card

SBT Example:

- TCOVER1: Successful withdrawal of funds
- TCOVER2: User’s card is unrecognized by ATM
- TCOVER3: User enters PIN incorrectly < 3 times
- TCOVER4: User enters pin incorrectly 3 times
- TCOVER5: User selects deposit or transfer
- TCOVER6: User selects incorrect account
- TCOVER7: User enters invalid withdrawal amount
- TCOVER8: Insufficient cash in the ATM
- TCOVER9: User enters non-dispensable amount
- TCOVER10:Userentersamountexceedingdaily allowance
- TCOVER11: Insufficient funds in user’s account

STB Example: Derive Test Cases

Test case #	1
Test case name	Successful withdrawal of funds
Scenario path exercised	U1, S1.1, U2, S2.1, U3.1, U4, S4.1, U5, S5.1, S6, S7, S8, S9, U6
Input	Valid card with valid customer account – assume 293910982246 is valid
	Valid PIN – assume 5652 is valid and matches card
	ATM Balance – $50,000
	Customer Account Balance – $100
	Withdrawal amount – $50
Pre-condition	A withdrawal can only be made by a user with an open bank account, a valid card and matching pin, and a working ATM
Expected result	Withdrawal has successfully been made from customer account
	ATM balance is $49,950
	Customer account balance is $50
	ATM is open, operational and awaiting a customer card as input
Test coverage item	TCOVER1

ST: Questions

How to Determine required rest coverage?

Coverage of typical and alternative scenarios?

Chapter 4:
Test Design Techniques

Section 4:
Structure-based or white box techniques

Structure-based Techniques

- Key concepts
  - Code coverage
  - Statement and decision coverage
  - Control-flow test design techniques
- Terms to remember

Structure-based (White Box) Fundamentals

- Structure-based tests are based on how the system works inside
  - Determine and achieve a level of coverage of control flows based on code analysis
  - Determine and achieve a level of coverage of data flows based on code and data analysis
  - Determine and achieve a level of coverage of interfaces, classes, call flows, and the like based on analysis of APIs, system design, etc.
- Coverage of structure is a way to check for gaps in specification- and experience-based tests

Code Coverage

- Levels of code coverage
  - Statement coverage: every statement executed
  - Branch (decision) coverage: every branch (decision) taken each way, true and false
  - Condition coverage: every combination of true and false conditions evaluated (i.e., the whole truth table)
  - Multicondition decision coverage: only those combinations of conditions that can influence the decision
  - Loop coverage: All loop paths taken zero, once, and multiple (ideally, maximum) times
- Does statement coverage imply branch (decision) coverage?

Code Coverage Example

- What test values for `n` do we need to cover all the statements?
  - n < 0, n > 0
- Does that get us branch coverage?
  - No, we also need n=0
- Does that get us condition coverage?
  - Yes: no compound conditions
- How about loop coverage?
  - Need to cover n = 1 and n = max

```
#include <stdio.h>
main()
{
  int i, n, f;
  printf(“n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d\n", n, f);
  }
  return n;
}
```

Code Coverage as a Test Design Tool

- By themselves, black box techniques can fail to cover as much as 75% or more of the statements
  - Is this a problem?
  - Depends on what is not covered!
- Code coverage tools can instrument a program to monitor code coverage during testing
- Gaps in code coverage can lead to more test cases to achieve higher coverage levels

McCabe Cyclomatic Complexity

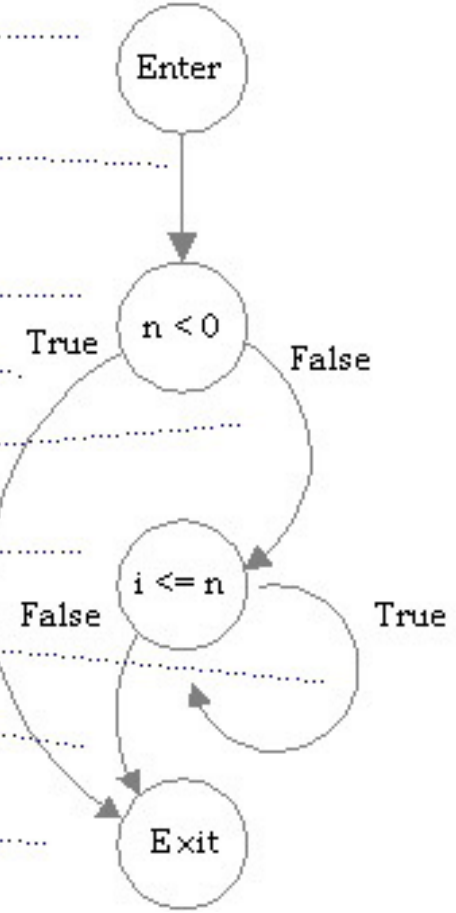
- McCabe's Cyclomatic Complexity measures control flow complexity
  - Measured by drawing a directed graph
  - Nodes represent entries, exits, decisions
  - Edges represent non-branching statements
- It has some useful testing implications
  - High-complexity modules are inherently buggy and regression-prone
  - The number of basis paths through the graph is equal to the number of basis tests to cover the graph
- Let's see how…

Cyclomatic Complexity for Factorial

Program

```
main()
{
  int i, n, f;
  printf("n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d.\n", n, f);
  }
  return n;
}
```

Flow Diagram



Cyclomatic Complexity

C = #R +1 =
2 + 1 = 3

or

C = #E - #N +2 =
5 - 4 + 2 = 3

Definitions

C = Cyclomatic complexity

R = Enclosed region

E = Edge (arrow)

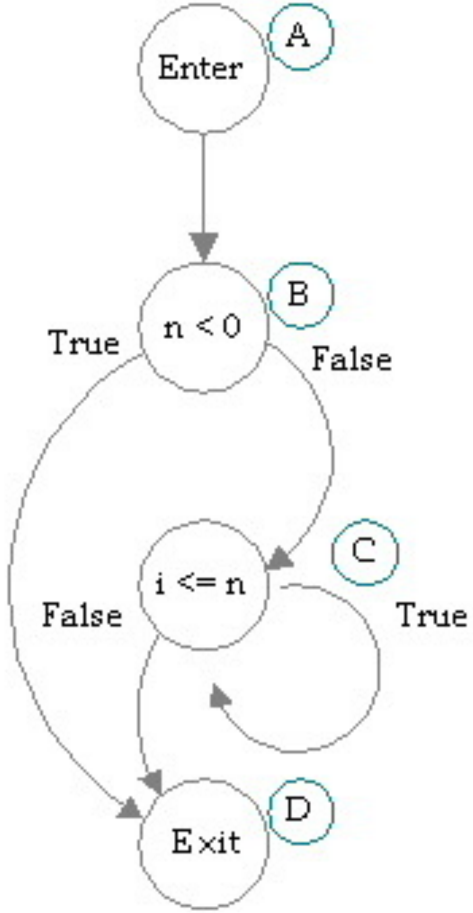
N = Node (bubble)

Basis Paths and Tests

Program

```
main()
{
  int i, n, f;
  printf("n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d.\n", n, f);
  }
   return n;
}
```

Flow Diagram



Basis Paths

1. ABD
2. ABCD
3. ABCCD

Basis Tests

	Input	Expect
1.	-1	Invalid: -1
2.	0	0! = 1.
3.	1	1! = 1.

Chapter 4:
Test Design Techniques

Section 4:
Exercise

A2.1.15 : Hexadecimal Converter

- On the next page you'll find a simple C program that accepts a string with hexadecimal characters (among other unwanted characters). It ignores the other characters and converts the hexadecimal characters to a numeric representation.
- If you test with input strings "059", "ace", and "ACD" what level of coverage will you achieve?
- What input strings could you add to achieve statement and branch coverage? Would those be sufficient for testing this program?
- Discuss.

Exercise: Hex Converter Program

```
main()
{
    int c;
    unsigned long int hexnum, nhex;
    hexnum = nhex = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
            case '0': case '1': case '2': case '3':
            case '4': case '5': case '6': case '7':
            case '8': case '9':
                /* Convert a decimal digit */
                nhex++;
                hexnum *= 0x10;
                hexnum += (c - '0');
                break;
            case 'a': case 'b': case 'c':
            case 'd': case 'e': case 'f':
            /* Convert a lower case hex digit */
                nhex++;
                hexnum *= 0x10;
                hexnum += (c - 'a' + 0xa);
                break;
            case 'A': case 'B': case 'C':
            case 'D': case 'E': case 'F':
                /* Convert an upper case hex digit */
                nhex++;
                hexnum *= 0x10;
                hexnum += (c - 'A' + 0xA);
                break;
            default:
                /* Skip any non-hex characters */
                break;
        }
    }
    printf("Got %d hex digits: %x\n", nhex, hexnum);
    return 0;
}
```

Chapter 4:
Test Design Techniques

Section 5:
Experience-based techniques

Experience-based Techniques

- Key concepts
  - Reasons for writing test cases based on intuition, experience and knowledge
  - Comparing experience-based techniques with specification-based testing techniques
- Terms to remember

Experience-based Tests

- Experienced-based tests are based on the tester's…
  - …skill and intuition
  - …experience with similar applications
  - …experience with similar technologies
- Rather than being pre-designed, experience-based tests are often created during test execution (i.e., test strategy is dynamic)
- Tests are frequently "time-boxed" (i.e., brief periods of testing focused on specific test conditions)
- Examples include error guessing, bug hunting, "breaking software" based on checklists or bug taxonomies, and exploratory testing

Common Experience-Based Approaches

- Most professional approaches to experience-based testing do not create tests entirely during execution
- May be guided by:
  - Checklist
  - Bug taxonomy
  - List of attacks
  - Bug hunt approach
  - Set of test charters
- These guidelines are prepared in advance to some level of detail
- Purely on-the-fly testing (ad hoc testing) is common, but usually (ineffective) manual random testing

Dynamic Test Strategies

Advantages

- Effective at finding bugs
- Resists pesticide paradox due to high variance
- Efficient (lightweight record-keeping)
- Good check on prepared tests
- Fun and creative

Disadvantages

- Gappy coverage, especially under pressure
- Difficult to estimate
- No bug prevention
- Extensive debriefing and discussion doesn’t scale to large teams
- Not all testers have the necessary level of skill, experience

Exploratory Testing Case Study

Staff	7 Technicians	3 Engineers + 1 Mgr.
Experience	<10 years total	> 20 years total
Test Type	Precise scripts	Chartered exploratory
Test Hrs/Day	42	6
Bugs Found	928 (78%)	261 (22%)
Effectiveness	22	44

This case study shows exploratory testing as about twice as effective at finding bugs on an hour-per-hour basis. It is significantly more efficient, because the scripted tests required extensive effort to create. However, to what extent is the relative level of experience important? What is the value of testing beyond finding bugs? What did the exploratory tests cover? What is the re-use value of the scripts?

Chapter 4: Test Design Techniques

Section 5: Exercise

Exercise: Dynamic vs. Pre-designed Tests

Major factors or concerns	Dynamic	Pre-designed
1.Do thorough regression testing		
2.Maximize bug finding efficiency		
3.Use testers with limited experience		
4.Automate half of the test cases		
5.Deliver precise, accurate estimates		
6.Test without test preparation time		
7.Test a rapidly changing UI		
8.Use a distributed test effort		

Put a "+" in the column where the factor or concern motivates towards using the approach and a "-" in the column where the factor or concern motivates away from using the approach.

Chapter 4:
Test Design Techniques

Section 6:
Choosing test techniques

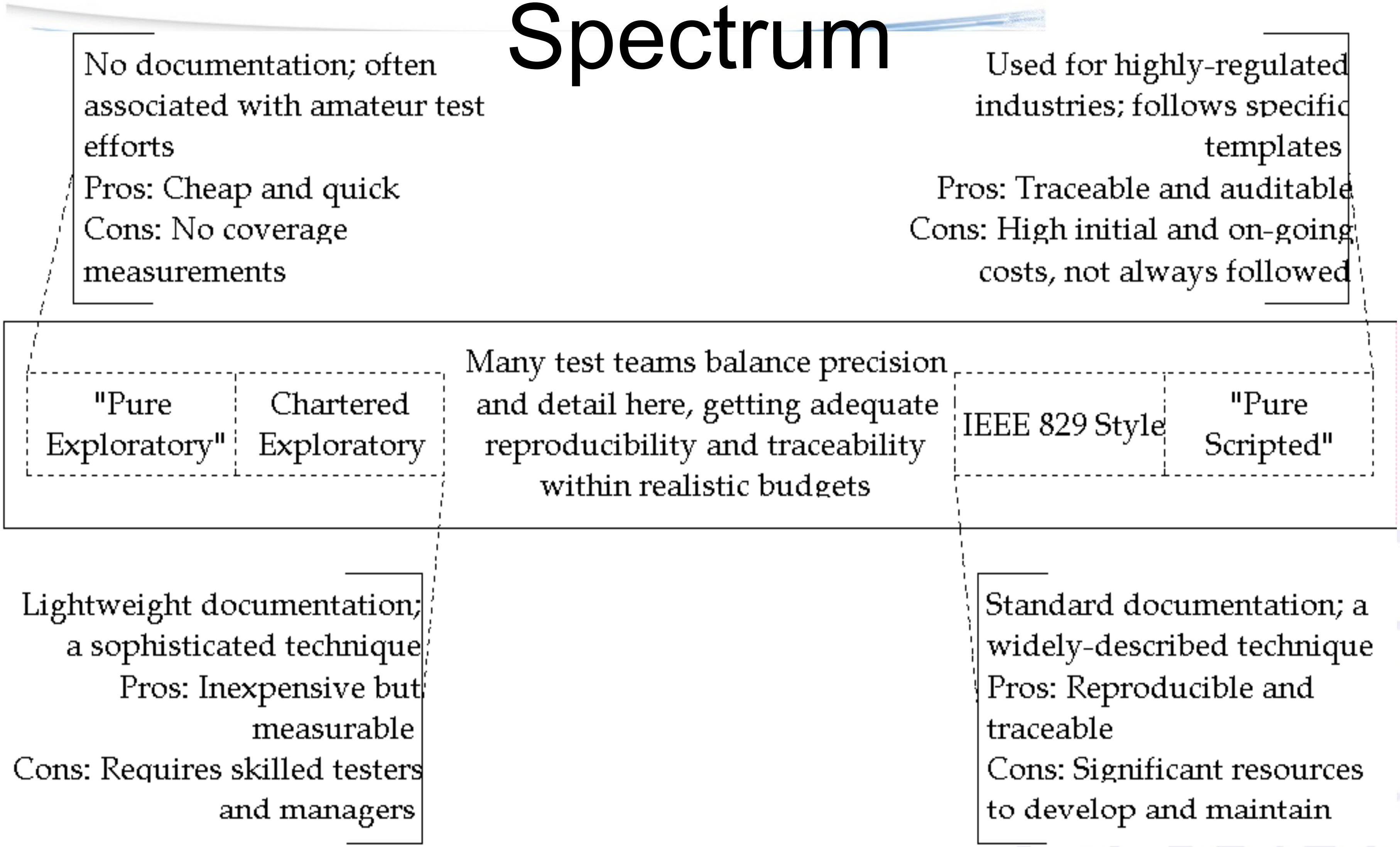
Choosing Test Techniques

- Key concept
  - The factors that influence the selection of appropriate test design techniques
- Terms to remember

Factors in Choosing Techniques

- Type of system
- Regulatory standards
- Customer or contractual requirements
- Level and type of risk
- Test objectives
- Documentation available
- Knowledge of testers
- Time and budget
- Development life cycle
- Previous experiences on types of defects found
- Others?

The Dynamic/Pre-designed Spectrum



No documentation; often associated with amateur test efforts
Pros: Cheap and quick
Cons: No coverage measurements

Used for highly-regulated industries; follows specific templates
Pros: Traceable and auditable
Cons: High initial and on-going costs, not always followed

"Pure Exploratory"

Chartered Exploratory

Many test teams balance precision and detail here, getting adequate reproducibility and traceability within realistic budgets

IEEE 829 Style

"Pure Scripted"

Lightweight documentation; a sophisticated technique
Pros: Inexpensive but measurable
Cons: Requires skilled testers and managers

Standard documentation; a widely-described technique
Pros: Reproducible and traceable
Cons: Significant resources to develop and maintain

Test Case Detail and Precision

- Trade-offs on the test documentation spectrum
  - Precise tests allow less skilled testers, but not very flexible
  - Imprecise tests can cover more conditions, but not very reproducible, especially across multiple testers
  - Precise tests provide transparent test criteria, but are hard and expensive to maintain
  - Imprecise tests are quick to write, but coverage can be hard to define and measure
- The degree to which a test effort is dynamic can be measured by counting the number of words of documentation (in your test cases and test procedures) per test hour of test execution

Chapter 4:
Test Design Techniques

Section 6:
Exercise

Exercise: Omninet Test Techniques

- List the factors discussed in this section that would affect the choice of test techniques and extent of documentation for Omninet.
- Discuss.

上海市嘉定区曹安公路4800号，同济大学嘉定校区软件学院