1 Overview	2 Life cycle components	3 Infrastructure components	4 Management components	5 Standards and Organizing
6 Static tesing	7 Dynamic testing	8 Test management	9 Tools	

Dynamic techniques

Learning objectives

- Explain the characteristics and differences between **specification-based** testing**, structure-based** testing and **experience-based** testing
- Compare the terms **test condition**, **test case** and **test procedure**
- Write test cases from given software models using techniques: **equivalence partitioning, boundary value analysis, decision tables, state transition testing**
- Write test cases from given control flows using techniques: **statement coverage, decision coverage**

References

- Dorothy Grahamet, Erik van Veenendaal, Isabel Evans, Rex Black. *Foundations of software testing: ISTQB Certification*

- Lee Copeland (2004). *A Practitioner's Guide to Software Test Design*. Artech House. ISBN:158053791x

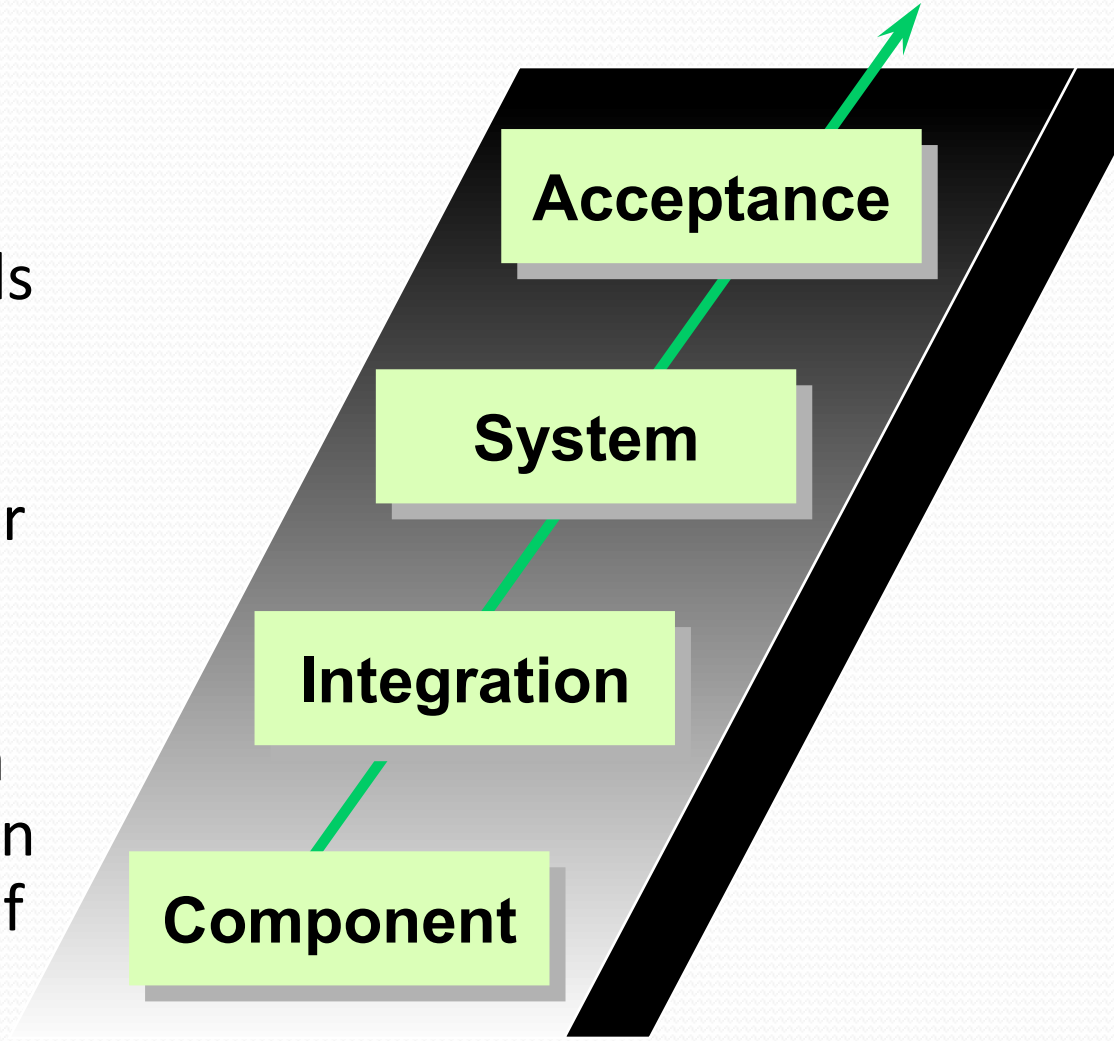
Contents

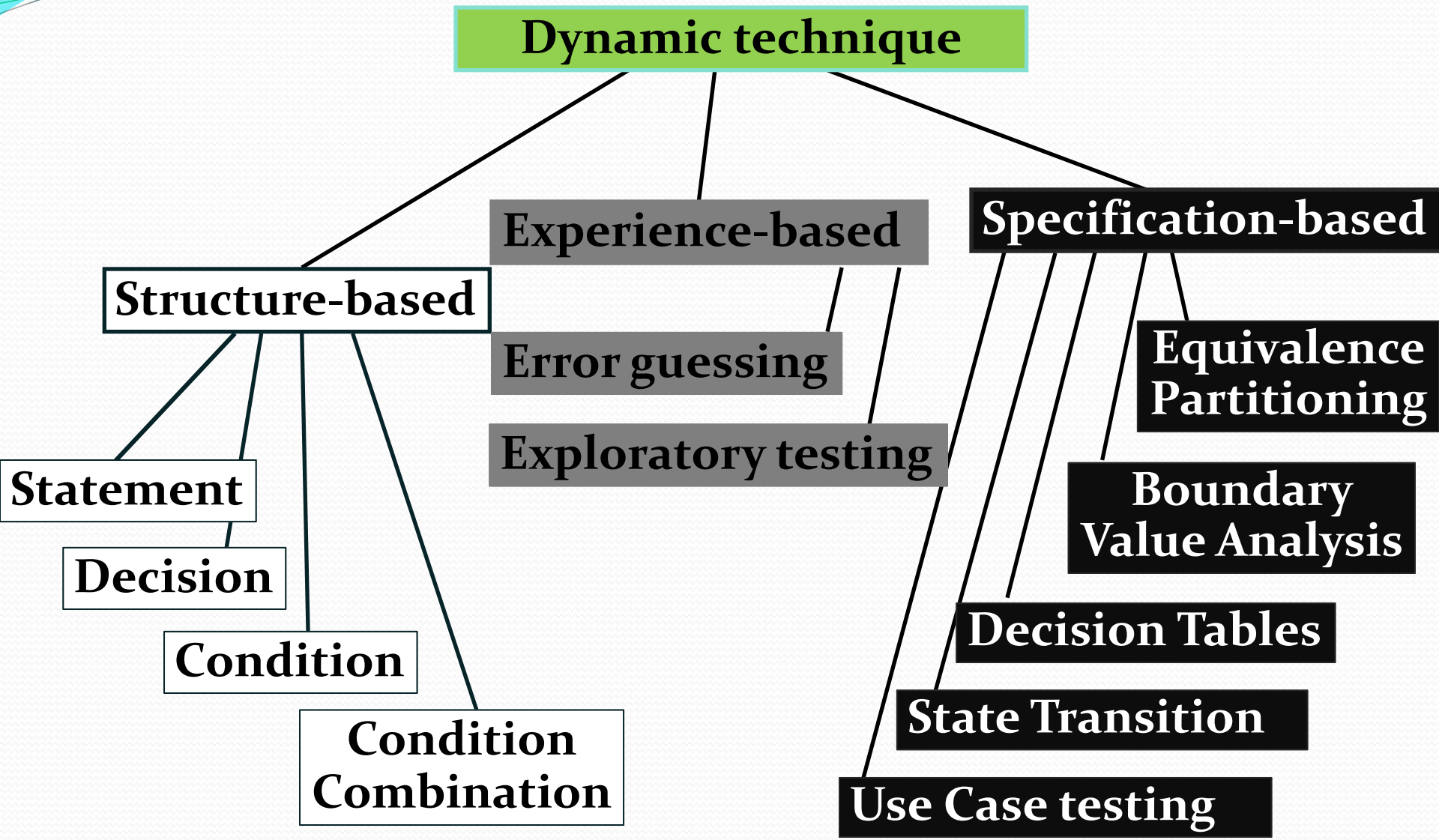
Categories of dynamic techniques

- Specification-based (black-box techniques)
  - view software as black-box with input and output
- Structure-based (white-box or glass-box techniques)
  - see the internal structure of the software
- Experience-based
  - use the tester's experience, knowledge and intuition

Where to apply?

- **Black box** appropriate at all levels but dominates higher levels of testing
- **White box** used predominately at lower levels
- **Experience-based techniques** used when there is no specification or inadequate or out of date




Dynamic technique
Structure-based
Experience-based
Specification-based
Statement
Decision
Condition
Condition Combination
Error guessing
Exploratory testing
Equivalence Partitioning
Boundary Value Analysis
Decision Tables
State Transition
Use Case testing

Contents

Dynamic techniques

Test condition – Test case – Test procedure

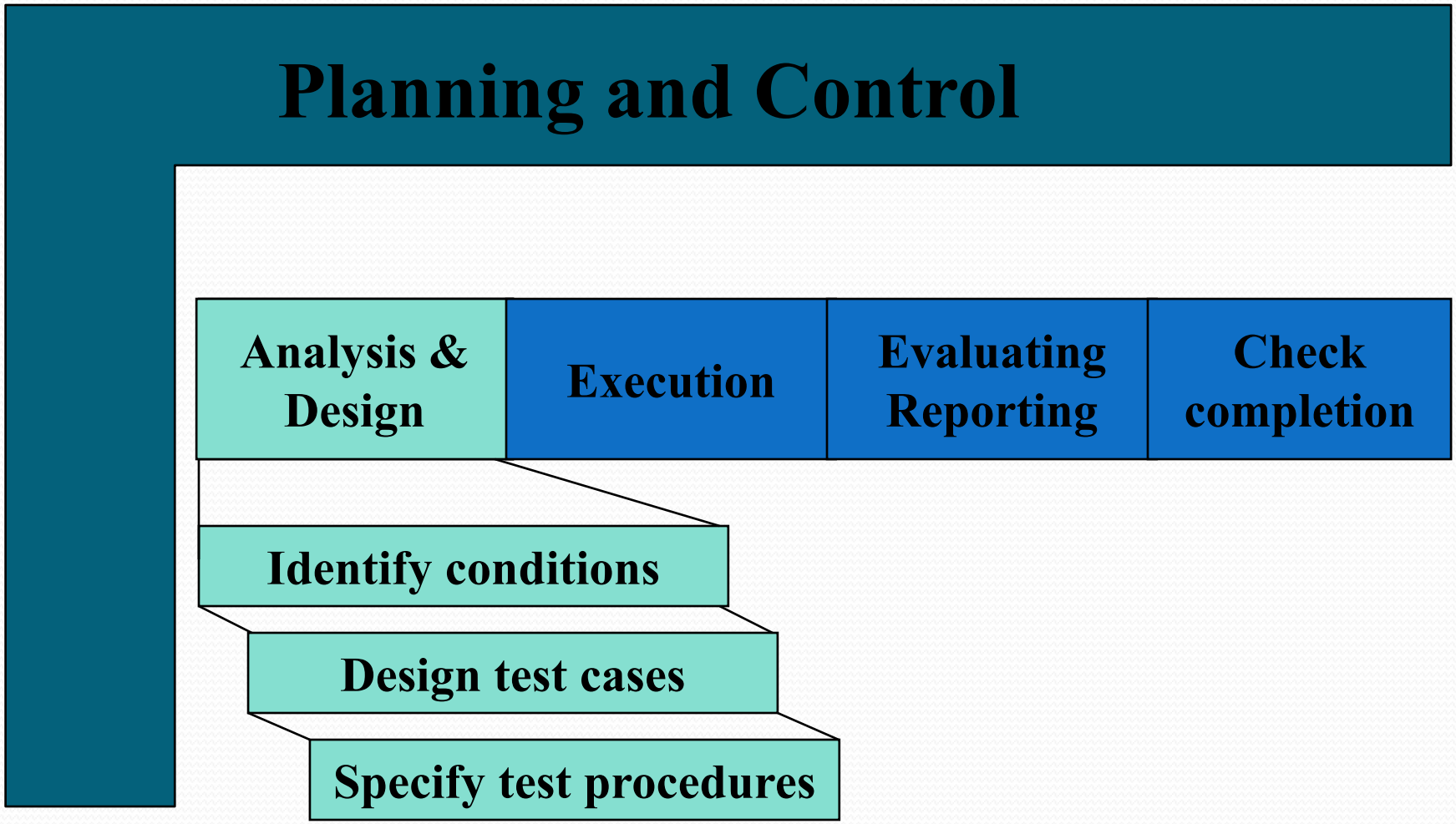
Black-box techniques

White-box techniques

Experience-based techniques

Choosing test techniques

Test process



Task 1: identify test conditions

- Test condition determine 'what' is to be tested, e.g.

  “number items ordered > 99”

  “the ID must be numeric”

- Based on (test basis):
  - system requirement
  - technical specification
  - code
  - business process
  - experienced user's knowledge of the system (sometimes)
- Prioritise the test conditions to ensure most important conditions are covered

Task 2: design test cases

- A set of **input values**, **expected results**, **pre-conditions**, **post-conditions**, developed for a *test condition*
  - input values: all inputs needed
  - expected result: predict the outcome of each test case
  - pre-condition: specifies things that must in place before the test can be run
  - post-condition: specifies anything that applies after the test case completes
- Prioritise the test cases

Task 3: specify test procedures

- Also referred to as a *test script*
- When to used:
  - describes the sequential steps to be taken in running a set of tests
  - some test cases may need to be run in a particular sequence

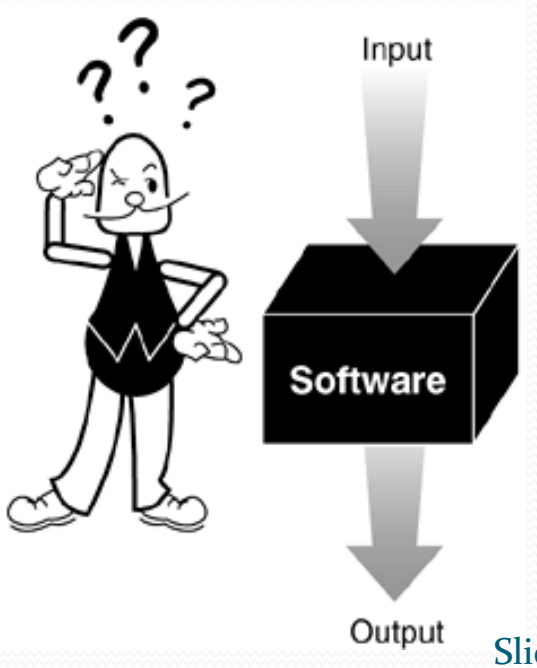
Test condition – Test case – Test procedure Example: Check Login functionality

Test Condition	Test Case Name	Pre-cond	Test Procedure	Input	Expected Results
Check Login functionality	Check valid User Name & Password		1) Launch application 2) Enter User Name 3) Enter Password 4) Click Login	User Name: admin Password: 123456	Login must be successfull

Contents

Black-box techniques

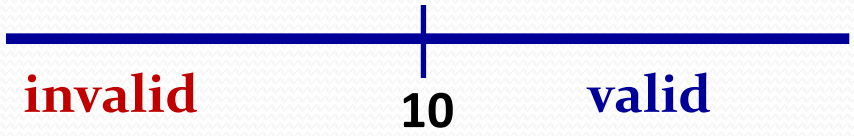
- Based on **specifications** (thông số kỹ thuật) or **models** of what the system should do
- Known as **specification-based techniques (kỹ thuật dựa trên đặc tả)**
- Including both functional and non-functional aspects
- Some techniques
  - equivalence partitioning
  - boundary value analysis
  - decision tables testing
  - state transition testing
  - use case testing



Equivalence partitioning (EP)

- **Divide (partition) the inputs, outputs,... into areas which makes the system behave "in the same manner"**
  - **we need test only one condition from each partition**
  - **if one element works correctly, all will work correctly**
- Rule: each input condition has at least two equivalence classes for it
  - one class that satisfies the condition – **valid** class
  - second class that doesn't satisfy the condition – **invalid** class

Ex1. Please input x>10



Ex2. Please input x in [a,b]

Equivalence partitioning Guidelines

- *Range* of values → one valid and two invalid classes

  "integer x shall be between 100 and 200" →

  {integer x | $100 \leq x \leq 200$},

  {integer x | $x < 100$},

  {integer x | $x > 200$}

- Specific *value* within a range → one valid and two invalid equivalence classes

  "value of integer x shall be 100" →

  {integer x | $x = 100$},

  {integer x | $x < 100$},

  {integer x | $x > 100$}

Equivalence partitioning Guidelines

- *Set* of values → one valid and one invalid equivalence class

  "weekday x shall be a working day" →

  x ∈ {Monday, Tuesday, Wednesday, Thursday, Friday},

  x ∈ {Saturday, Sunday}

- *Set* of values, and each case will be dealt with differently → a valid equivalence class for each element and only one invalid class for values outside the set

  "a discount code must be input as P for a preferred customer, R for a standard reduced rate, or N for none, and if each case is treated differently" →

  code=P, code=R, code=N, code=not one of P, R, N

Equivalence partitioning Guidelines

- *Boolean* ➔ one valid and one invalid equivalence class

  "condition x shall be true" →

  x = true, x = false

- One or several equivalence classes for *illegal* values, that is, for values that are *incompatible with the type* of the input parameter and therefore out of the parameter's domain

  "integer values x" ➔

  {real-number x}, {character-string x}

Equivalence partitioning Guidelines

- If an input condition specifies a '*must be*' situation

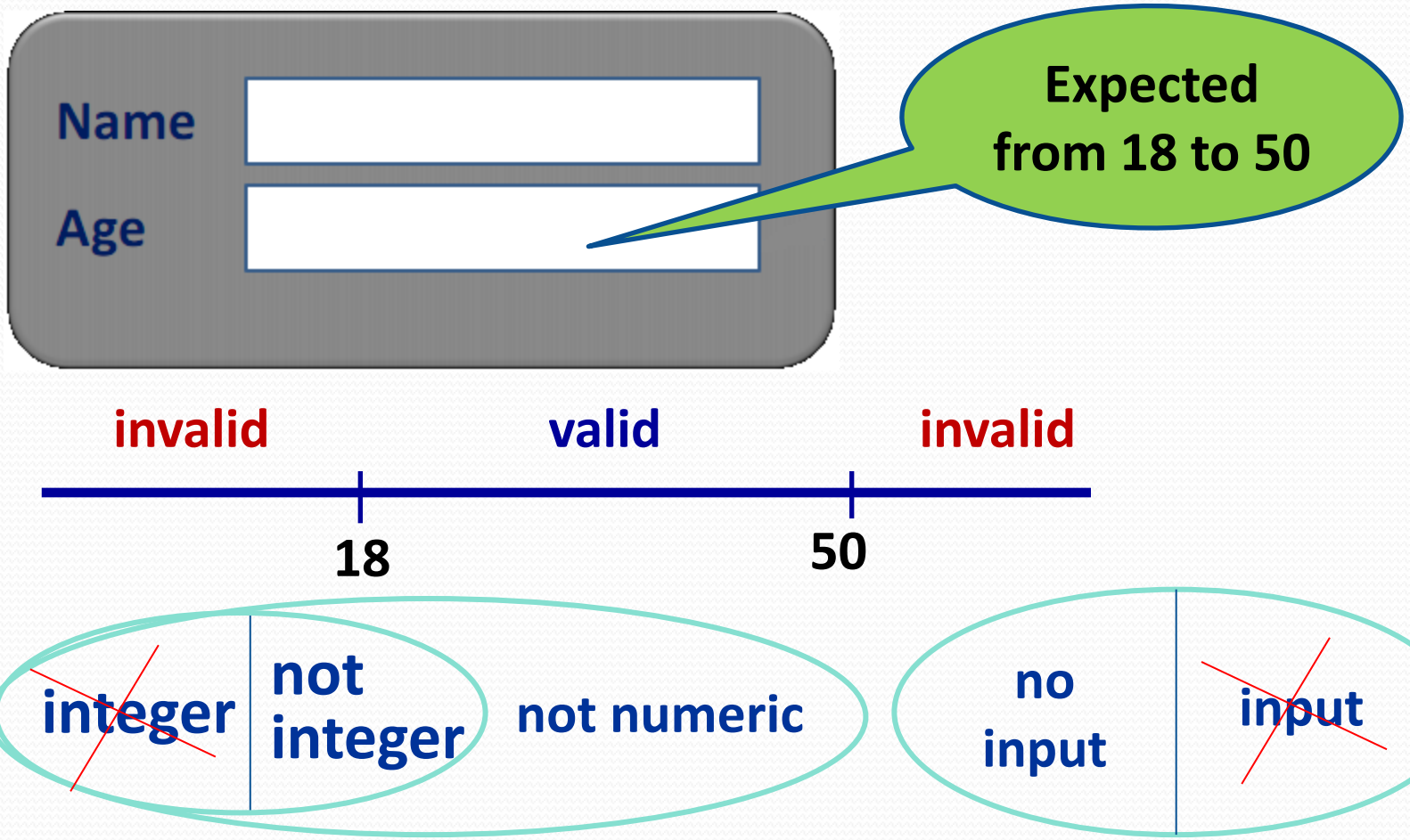
  "first character of the identifier must be a letter" ➔

  {first character is a letter}, {first character is not a letter}

- Equivalence classes can be of the output desired in the program

- If there is reason to believe that the system handles each valid/invalid/illegal input value differently, then each value shall generate an equivalence class

Example EP 1


Name
Age
Expected
from 18 to 50
invalid
valid
invalid
Age
18
50
integer
not
integer
not numeric
no
input
input

Example EP 1 (cont.)

- Draw table of analysis

Condition	Valid partition	Invalid partition
Age	integer between 18 and 50	<18
		>50
		not integer
		not numeric
		no input

Design test case

- Write a new test case covering **as many** of the uncovered **valid equivalence classes** as possible, until all valid equivalence classes have been covered by test cases

- Write a test case that covers **one, and only one**, of the uncovered **invalid equivalence classes**, until all invalid equivalence classes have been covered by test cases

- Example (next slide)

Design test case for EP: Example

Table of analysis

Condition	Valid partition	Invalid partition
Age	integer between 18 and 50	<18
		>50
		not numeric
		not integer
		no input

Test case design

Condition	Test case name	Inputs	Expected results
Test on Age field	Test valid age	Abc;20	Ok…
	Test age<18	Abc;10	Message “Age must be >18”
	Test age>50	Abc;60	Message “Age must be <50”
	Test invalid characters	Abc;ab	Message “Age must be a numeric”
	Test not integer	Abc;21.5	Message “Age must be an integer”
	Test null value	Abc;	Message “Age not allow null”

Example EP 2

The Golden Splash Swimming Center's ticket price depends on four variables: day (weekday, weekend), visitor's status (OT = one time, M = member), entry hour (6.00–19.00, 19.01–24.00) and visitor's age (up to 16, 16.01–60, 60.01–120).

	Mon, Tue, Wed, Thurs, Fri				Sat, Sun			
Visitor's status	OT	OT	M	M	OT	OT	M	M
Entry hour	6.00–19.00	19.01–24.00	6.00–19.00	19.01–24.00	6.00–19.00	19.01–24.00	6.00–19.00	19.01–24.00
	Ticket prices – $							
Visitor's age								
0.0–16.00	**5.00**	**6.00**	**2.50**	**3.00**	**7.50**	**9.00**	**3.50**	**4.00**
16.01–60.00	**10.00**	**12.00**	**5.00**	**6.00**	**15.00**	**18.00**	**7.00**	**8.00**
60.01–120.00	**8.00**	**8.00**	**4.00**	**4.00**	**12.00**	**12.00**	**5.50**	**5.50**

Define valid and invalid equivalence classes and the corresponding test case values.

Example EP 2: Solution

Condition	Valid Par.	Invalid Par.
Day of week		
Visitor's status		
Entry hour		
Visitor's age		

Example EP 2: Solution (cont.)

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For valid partition	1	Mon	OT	7.55	8.4	$5.00
	2	Sat	M	20.44	42.7	$8.00
	3	Sat	M	22.44	65.0	$5.50

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For invalid partition	1	Mox	OT	7.55	8.4	Invalid day
	2	Mon	88	7.55	8.4	Invalid visitor status
	3	Mon	OT	4.40	8.4	Invalid entry hour
	4	Mon	OT	&@	8.4	Invalid entry hour
	5	Mon	OT	7.55	TTR	Invalid visitor age
	6	Mon	OT	7.55	150.1	Invalid visitor age
	⋮					

Example EP 3

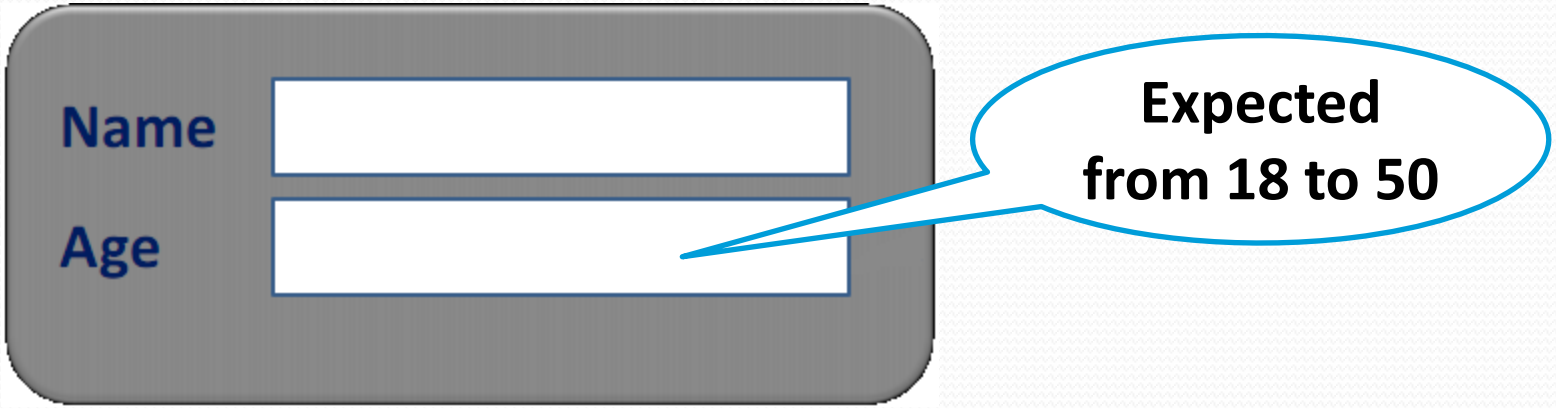
A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Boundary value analysis (BVA)

- Based on testing at the boundaries between partitions (the maximum and minimum values of partitions)
- Have both *valid boundaries* (in the valid partitions) and *invalid boundaries* (in the invalid partitions)
- For example, if a program should accept a sequence of numbers between 1 and 100

Boundary values: 0, 1, 100, 101

Example BVA 1



Condition	Valid Boundary	Invalid Boundary
Age	18	17
	50	51

Example BVA 2

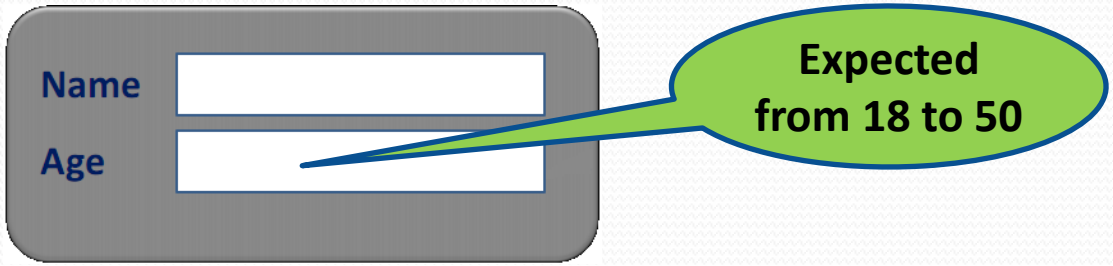
- The Golden Splash Swimming Center's ticket price

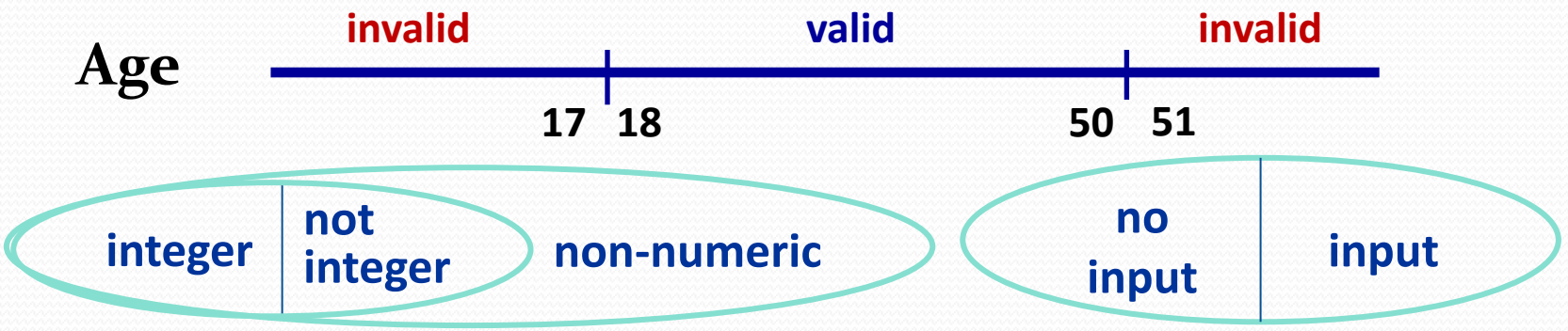
Condition	Valid Boundary	Invalid Boundary
Entry hour	6.00	5.59
	19.00	24.01
	19.01	
	24.00	
Visitor's age	0.0	-0.01
	16.00	120.01
	16.01	
	60.00	
	60.01	
	120.00	

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For valid boundary	1	Sat	M	6.00	0.0	$3.50
	2	Sat	M	19.00	16.00	$3.50
	3	Sat	M	19.01	16.01	$8.00
	4	Sat	M	19.01	60.00	$8.00
	5	Sat	M	24.00	60.01	$5.50
	6	Sat	M	24.00	120.0	$5.50

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For invalid boundary	1					
	2					
	3					
	4					

Example EP - BVA



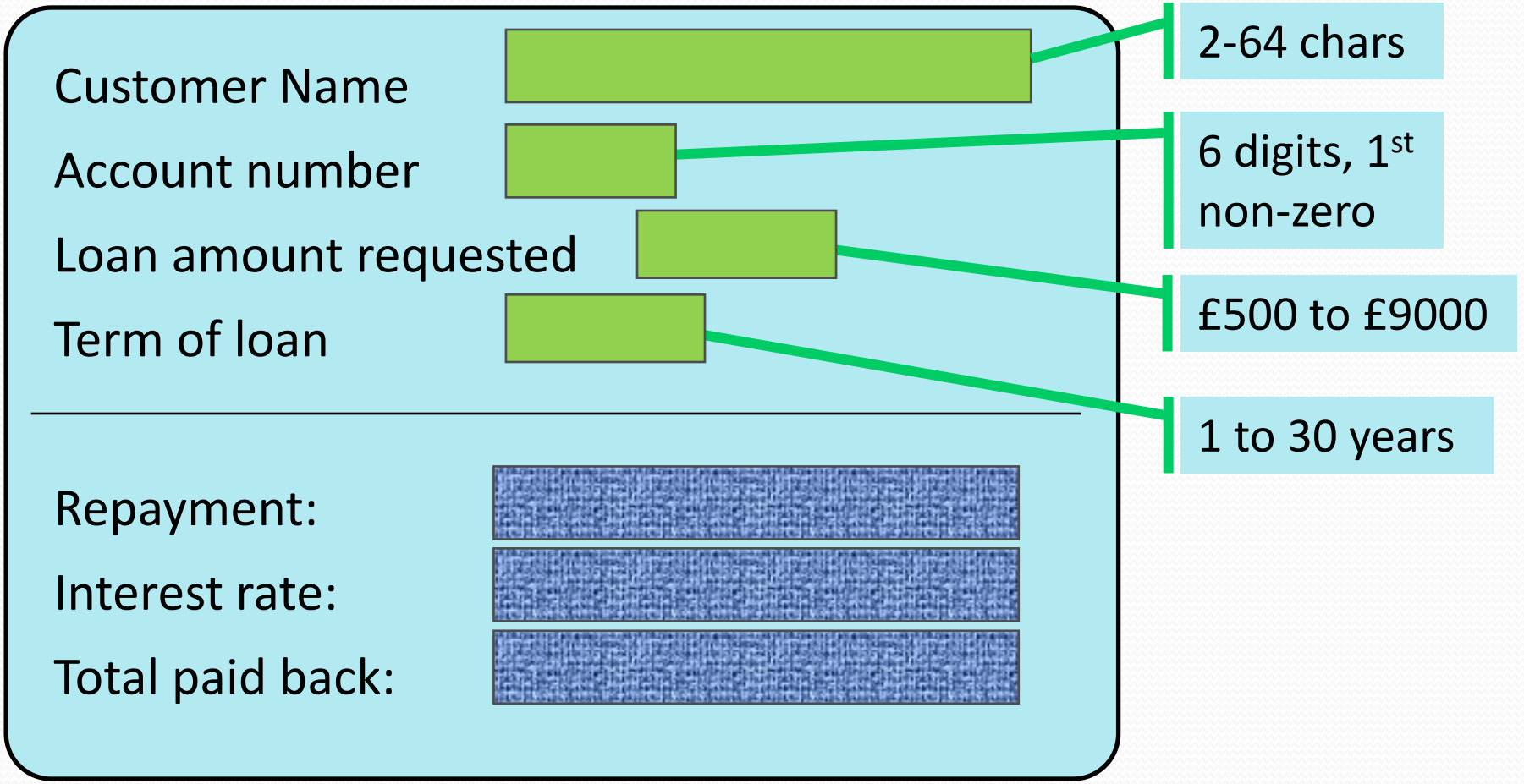


Condition	Valid Partition	Invalid Partition	Valid Boundary	Invalid Boundary
Age	18 - 50	< 18	18	17
		> 50	50	51
		Số thực		
		Không là kí tự số		
		Rỗng		

Design test case (for Age)

#	Test case type	Input		Expected result
		Name	Age	
1	Valid partition cho Age	Nguyen An	30	(tìm trong đặc tả)
2	Invalid partition cho Age	Nguyen An	15	"Lỗi: nhập Age <18"
3		Nguyen An	60	"Lỗi: nhập Age >50"
4		Nguyen An	20.5	"Lỗi: Age là số thực"
5		Nguyen An	ab	"Lỗi: Age không là số"
6		Nguyen An		"Lỗi: Age rỗng"
7	Valid boundary cho Age	Nguyen An	18	(tìm trong đặc tả)
8		Nguyen An	50	(tìm trong đặc tả)
9	Invalid boundary cho Age	Nguyen An	17	"Lỗi: nhập Age <18"
10		Nguyen An	51	"Lỗi: nhập Age >50"

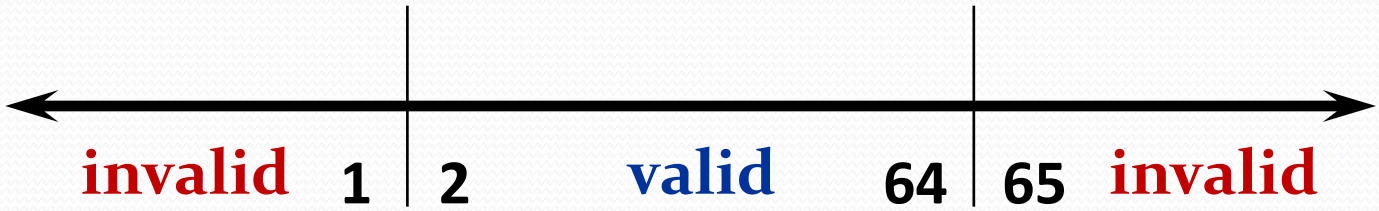
Exercise 1: Loan application



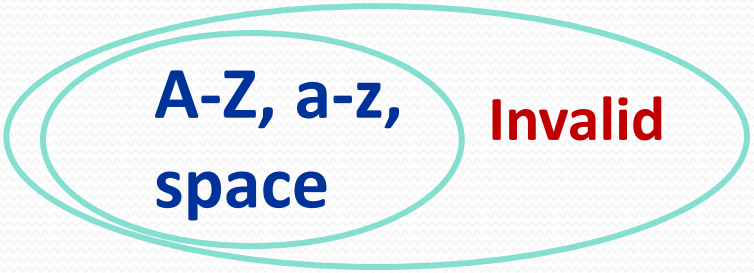
Customer name

2-64 chars

Number of characters:



Valid characters

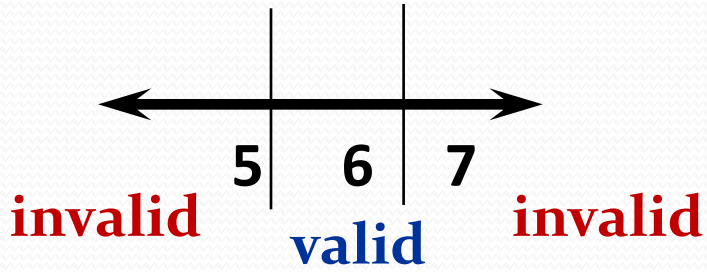


Condition	Valid Partition	Invalid Partition	Valid Boundary	Invalid Boundary
Customer name	2 to 64 chars	~~< 2 chars~~	2 chars	1 chars
		> 64 chars	64 chars	65 chars
		not a name		
		null		

Account number

6 digits, 1st non-zero

Number of digits:



First character:

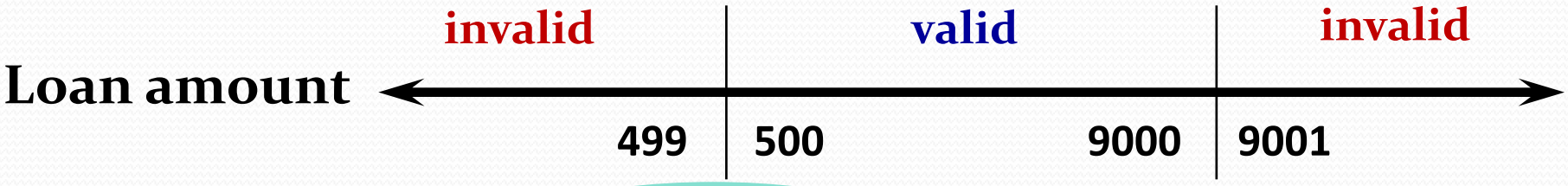
- valid: non-zero
- invalid: zero

Valid characters:

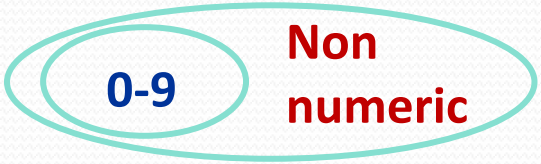
Condition	Valid Partition	Invalid Partition	Valid Boundary	Invalid Boundary
Account number	6 digits and 1st non-zero	< 6 digits		5 digits
		> 6 digits		7 digits
		6 digits and 1st digit = 0		
		non-digit		
		null		

Loan amount

£500 to £9000



Valid characters:

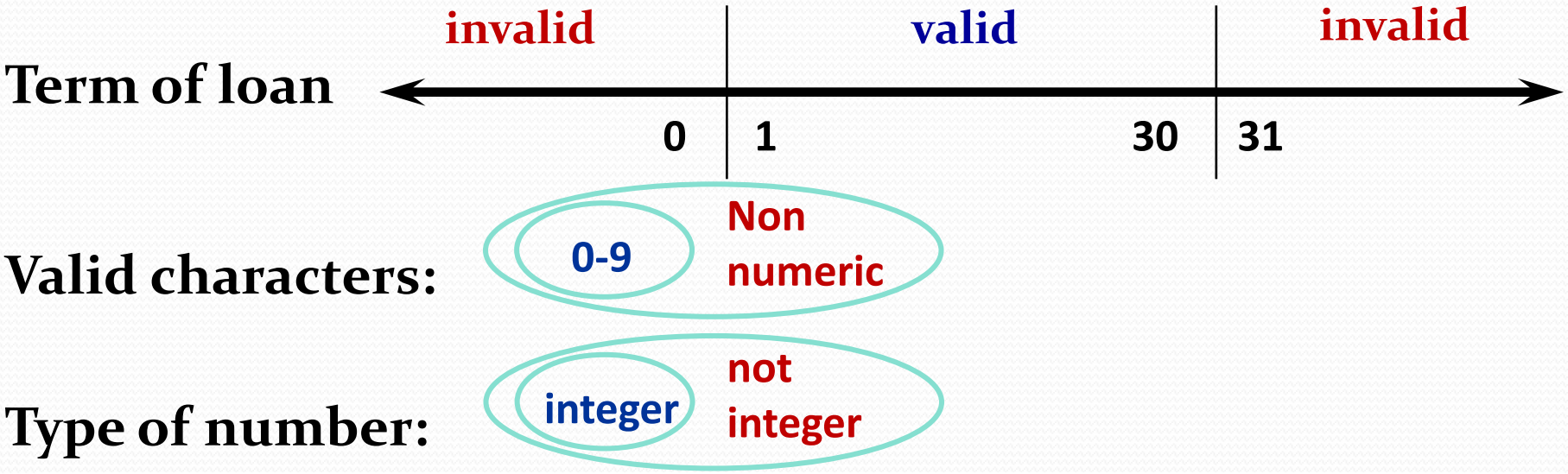


Type of number:

Condition	Valid Partition	Invalid Partition	Valid Boundary	Invalid Boundary
Loan amount	500 – 9000	< 500	500	499
		>9000	9000	9001
		non numeric		
		not integer		
		null		

Term of loan

1 to 30 years



Condition	Valid Partition	Invalid Partition	Valid Boundary	Invalid Boundary
Term of loan	1 – 30	< 1	1	0
		>30	30	31
		non numeric		
		not integer		
		null		

Condition template

Conditions	Valid Partitions	Invalid Partitions	Valid Boundaries	Invalid Boundaries
Customer name	2 - 64 chars	> 64 chars	2 chars	1 char
		not a name	64 chars	65 chars
		null		
Account number	6 digits and 1st non-zero	< 6 digits		5 digits
		> 6 digits		7 digits
		6 digits and 1st digit = 0		0 digits
		non-digit		
Loan amount	500 - 9000	< 500	500	499
		>9000	9000	9001
		non numeric		
		not integer		
		null		
Term of loan	1-30	<1	1	0
		>30	30	31
		non numeric		
		not integer		
		null		

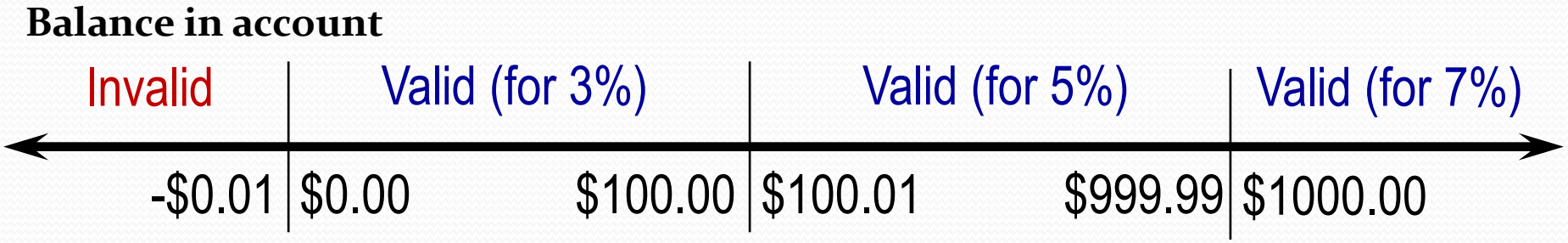
Design test case for loan application

#	Test case name	Input	Expected result
1	Test valid partition	Customer name= John H Account numer= 123456 Loan amount= 600 Term of loan= 2	Repayment= Interest rate= Total paid back=
2	Customer name: Test invalid partition number of chars (> 64 chars)	…	Error message: …
…	…	…	…

Exercise 2: Bank account

Suppose you have a bank account that the rate of interest depending on the balance in the account: a balance in the range $0 up to $100.00 has a 3%, a balance over $100.00 and up to $1000.00 has a 5%, and balances of $1000.00 and over have a 7%. **What valid partition, invalid partition, valid boundary and invalid boundary might you use? What test cases we design?**

Solution: Bank account



Test conditions	Valid partitions	Invalid partitions	Valid boundaries	Invalid boundaries
Balance in account	$0.00 - $100.00	< $0.00	$0.00	**-$0.01**
	$100.01 - $999.99	>$**Max**	$100.00	**$Max**+0.01
	$1000.00 - $Max	non-digit (if balance is an input field)	$100.01	
			$999.99	
		null	$1000.00	
			$**Max**	

BVA with ‘open boundary’

- One of the sides of the partition is not defined
- How to test?
  - Go back to the specification (đặc tả)
  - Investigate (xem xét) other related areas of the system
  - Probably need to use an intuitive (trực quan) or experience-based (dựa trên kinh nghiệm) approach to probe various large values trying to make it fail

Exercise 3

- A mail-order company selling flower seeds charges
  - £3.95 for postage and packing on all orders up to £20.00 value and
  - £4.95 for orders above £20.00 value and up to £40.00 value.
  - For orders above £40.00 value there is no charge for postage and packing.
- If you were using equivalence partitioning to prepare test cases for the postage and packing charges what valid partitions would you define?
- What about non-valid partitions? What boundary values? Design test cases.

Exercise 3: Solution

Orders

invalid	valid	valid (for £3.95)	valid (for £4.95)	valid (for £0)
-£0.01	£0.00	£0.01 £20.00	£20.01 £40.00	£40.01

Test conditions	Valid partitions	Invalid partitions	Valid boundaries	Invalid boundaries
Orders	£0.01 - £20.00	< £0.00	£0.00	-£0.01
	£20.01 - £40.00	> £Max	£0.01	£Max+0.01
	£40.01- £Max	non-digit	£20.00	
		null	£20.01	
			£40.00	
			£40.01	
			£Max	

Exercise 3: Solution - Design test cases

Condition	#	Inputs	Expected results
Orders	1	£10.00	Postage and packing = 3.95
	2	£30.00	Postage and packing = 4.95
	3	£50.00	Postage and packing = 0
	4	-£1.00	Invalid orders…
	5	a1	Invalid character…
	6		Not allow empty
	7	£0.00	Postage and packing = 0
	8	£0.01	Postage and packing = 3.95
	9	£20.00	Postage and packing = 3.95
	10	£20.01	Postage and packing = 4.95
	11	£40.00	Postage and packing = 4.95
	12	£40.01	Postage and packing = 0
	13	-0.01	Invalid orders…

Applicability and Limitations

- **Equivalence class** and **boundary value testing** are most suited to systems in which much of the input data takes on values within **ranges** or within **sets**
- Applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system's requirements

Decision tables testing

- A good way to deal with combination of inputs, which produce different results
- **Decision table**
  - Known as a 'cause-effect' table
  - A table showing **combinations of input** with their associated **output** or **action**

	Business Rule 1	Business Rule 2	Business Rule 3	Business Rule 4
Condition 1	T	T	F	F
Condition 2	T	F	T	T
Condition 3	T	-	F	T
Action 1	Y	N	N	N
Action 2	N	Y	Y	N

Decision tables testing

A	T	T	F	F
B	T	F	T	F
R1	N	N	Y	Y

- Design and using decision table
  1. identify **conditions** (which need to be combined)
  2. put them into a **table**
  3. identify all of the **combinations** of true and false
  4. identify the correct **outcome** for each combination
     - rationalise input combinations
       - some combinations may be impossible or not of interest
       - use a hyphen to denote “don’t care”
  5. write **test cases** for each of the rule in the table
     - each column is one test case (at least), the Conditions specify the *inputs* and the Actions specify the *expected results*

Decision tables testing example 1

- Car rental example:
  - The specification says: If Age is over 23 and the person has a clean driving record, supply rental car, else reject.

Conditions/Input	Rule 1	Rule 2	Rule 3	Rule 4
Age > 23				
Clean driving record				
Action/Output				
Supply rental car				

#	Test case description	Expected result
1	Mr. A 30 years old, having clean driving record	Allow to rent car
2	Mr. B 30 years old, not having clean driving record	Not allow to rent car
3	…	…
4	…	…

Decision tables testing example 2

Credit card worked example.

If you are a **new customer** opening a credit card account, you will get a 15% discount on all your purchases today.

If you are an **existing customer** and you hold a **loyalty card (thẻ khách hàng thân thiết)**, you get a 10% discount.

If you have a **coupon (phiếu giảm giá)**, you can get 20% off today (but it can't be used with the 'new customer' discount).

Discount amounts are added, if applicable.

Decision tables testing example 2

Conditions	R1	R2	R3	R4	R5	R6	R7	R8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Action								
Discount (%)	X	X	20	15	30	10	20	0

X: Error message
R3: must review specification

Decision tables testing example 2

Conditions	R1	R3	R4	R5	R6	R7	R8
New customer (15%)	T	T	T	F	F	F	F
Loyalty card (10%)	T	F	F	T	T	F	F
Coupon (20%)	-	T	F	T	F	T	F
Action							
Discount (%)	**X**	**20**	**15**	**30**	**10**	**20**	**0**

Extending decision tables - 1

- Entries can be more than just ‘true’ or ‘false’
  - completing table needs to be done carefully
  - rationalising becomes more important
- Example

Code = 1, 2, or 3	1	1	1	1	2	2	2	2	3	3	3	3
Exp.date < now	T	T	F	F	T	T	F	F	T	T	F	F
Class A product	T	F	T	F	T	F	T	F	T	F	T	F

Extending decision tables - 2

- Decision table with **multiple actions**
  - Decision tables may specify more than one action for each rule
  - Example

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Condition-1	Yes	Yes	No	No
Condition-2	Yes	No	Yes	No
Actions				
Action-1	Do X	Do Y	Do X	Do Z
Action-2	Do A	Do B	Do B	Do B

Extending decision tables - 3

- A decision table with **non-binary conditions**

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Condition-1	0–1	1–10	10–100	100–1000
Condition-2	<5	5	6 or 7	>7
Actions				
Action-1	Do X	Do Y	Do X	Do Z
Action-2	Do A	Do B	Do B	Do B

Test Case ID	Condition-1	Condition-2	Expected Result
TC1	0	3	Do X / Do A
TC2	5	5	Do Y / Do B
TC3	50	7	Do X / Do B
TC4	500	10	Do Z / Do B

Decision tables testing exercise

Scenario:

If you hold an 'over 60s' rail card, you get a 34% discount on whatever ticket you buy.

If you are traveling with a child (under 16), you can get a 50% discount on any ticket if you hold a family rail card, otherwise you get a 10% discount.

You can only hold one type of rail card.

Produce a decision table showing all the combinations of fare types and resulting discounts and derive test cases from the decision table.

Solution - decision table

Conditions	R1	R2	R3	R4	R5	R6	R7	R8
'over 60s' rail card?	T	T	T	T	F	F	F	F
with a child?	T	T	F	F	T	T	F	F
family rail card?	T	F	T	F	T	F	T	F
Action								
discount	50%	34%	34%	34%	50%	10%	0%	0%

Conditions	R1	R2	R3	R5	R6	R7
'over 60s' rail card?	T	T	T	F	F	F
with a child?	T	-	F	T	T	F
family rail card?	T	F	T	T	F	-
Action						
discount	50%	34%	34%	50%	10%	0%

Solution – test cases

#	Input	Expected outcome
1	John, with 'over 60s' rail card and family rail card, traveling with his son (age 10)	50% discount for both tickets
2	Rogers, with 'over 60s', traveling with his wife	34% discount (for Rogers only)
3	Tom, with 'over 60s' rail card and family rail card	34% discount
4	Anna, with family rail card, traveling with her baby	50% for both tickets
5	Betty, no rail card, traveling with his 5-year-old niece	10% discount for both tickets
6	Henry, no rail card	No discount

Applicability and Limitations

- Decision table testing can be used whenever the system must implement **complex business rules** when these rules can be represented as a combination of conditions and when these conditions have discrete actions associated with them

State transition testing

- This technique is helpful where you need to test differrent system transitions
  - system where you **get a different output for the same input**, depending on current state and past state
- Based on **state transition diagram**

State transition diagram

- Called State Chart or Graph
- There are four main components of the graph

1) **states** software may occupy

2) **transitions** from one state to another

3) **events** that cause transition

PIN not OK

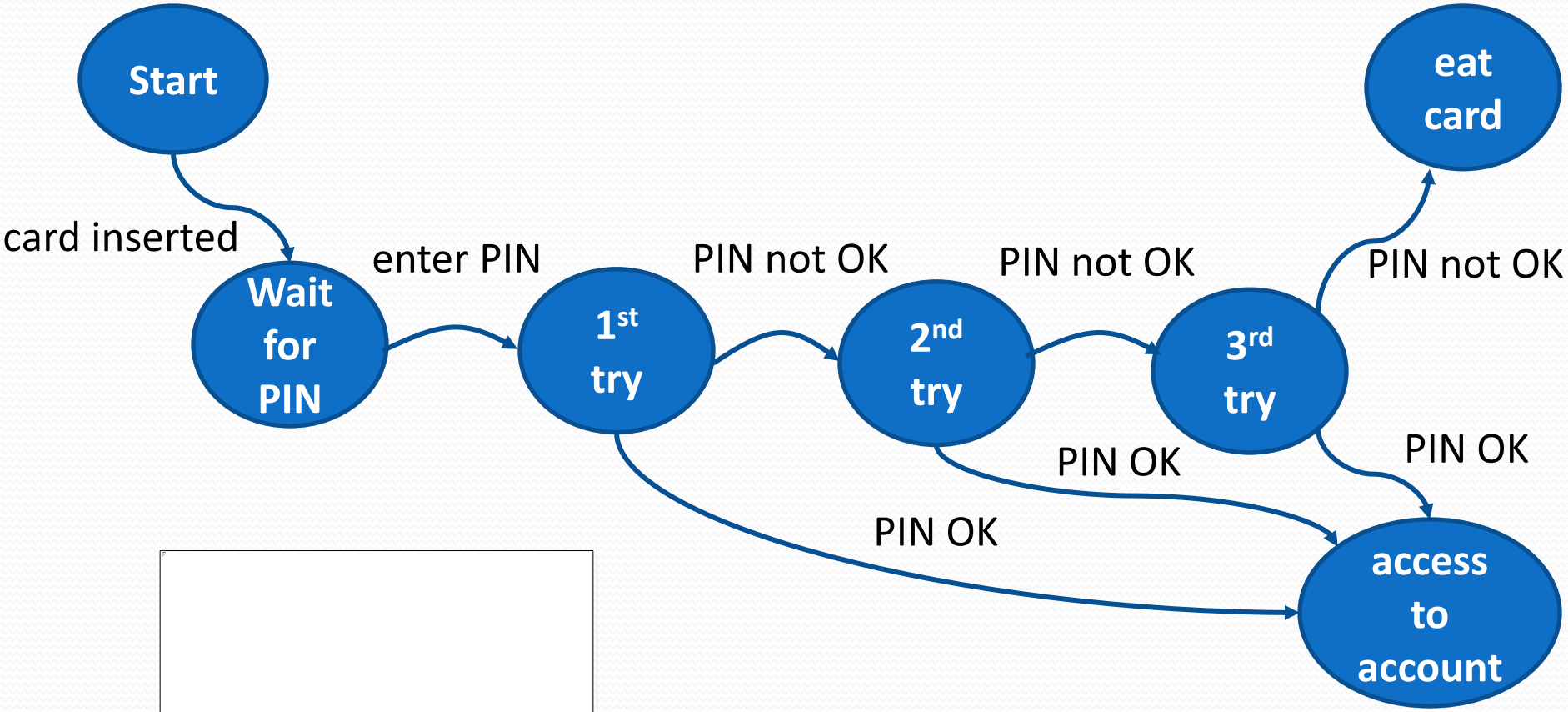
4) **actions** that result from transition

Example

- Entering a PIN to a bank account

State transition diagram

- Example State diagram for enter PIN

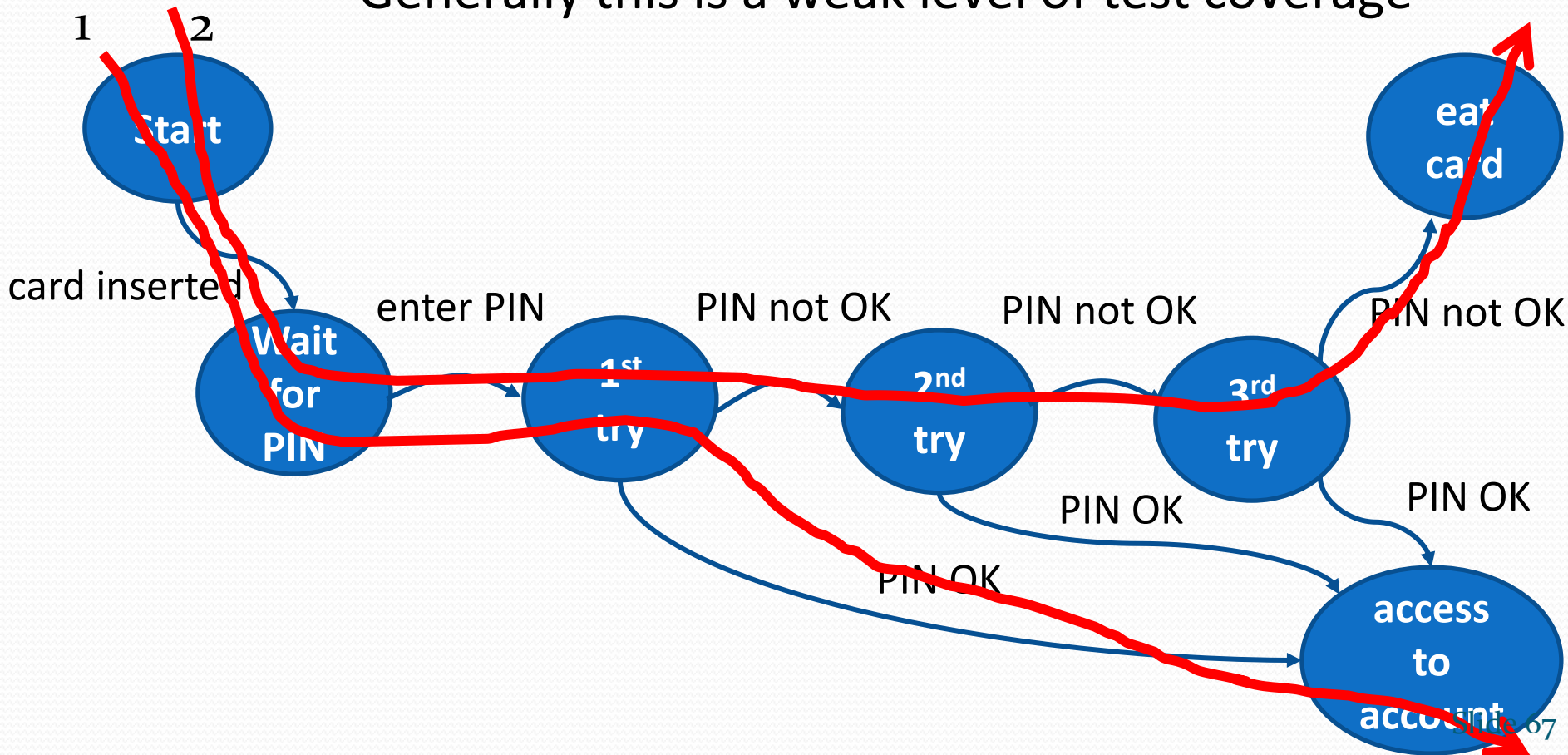


Creating test cases

- Test conditions can be derived from the state graph in various ways
- Four different levels of coverage
  - state coverage
  - event coverage
  - path coverage
  - transition coverage

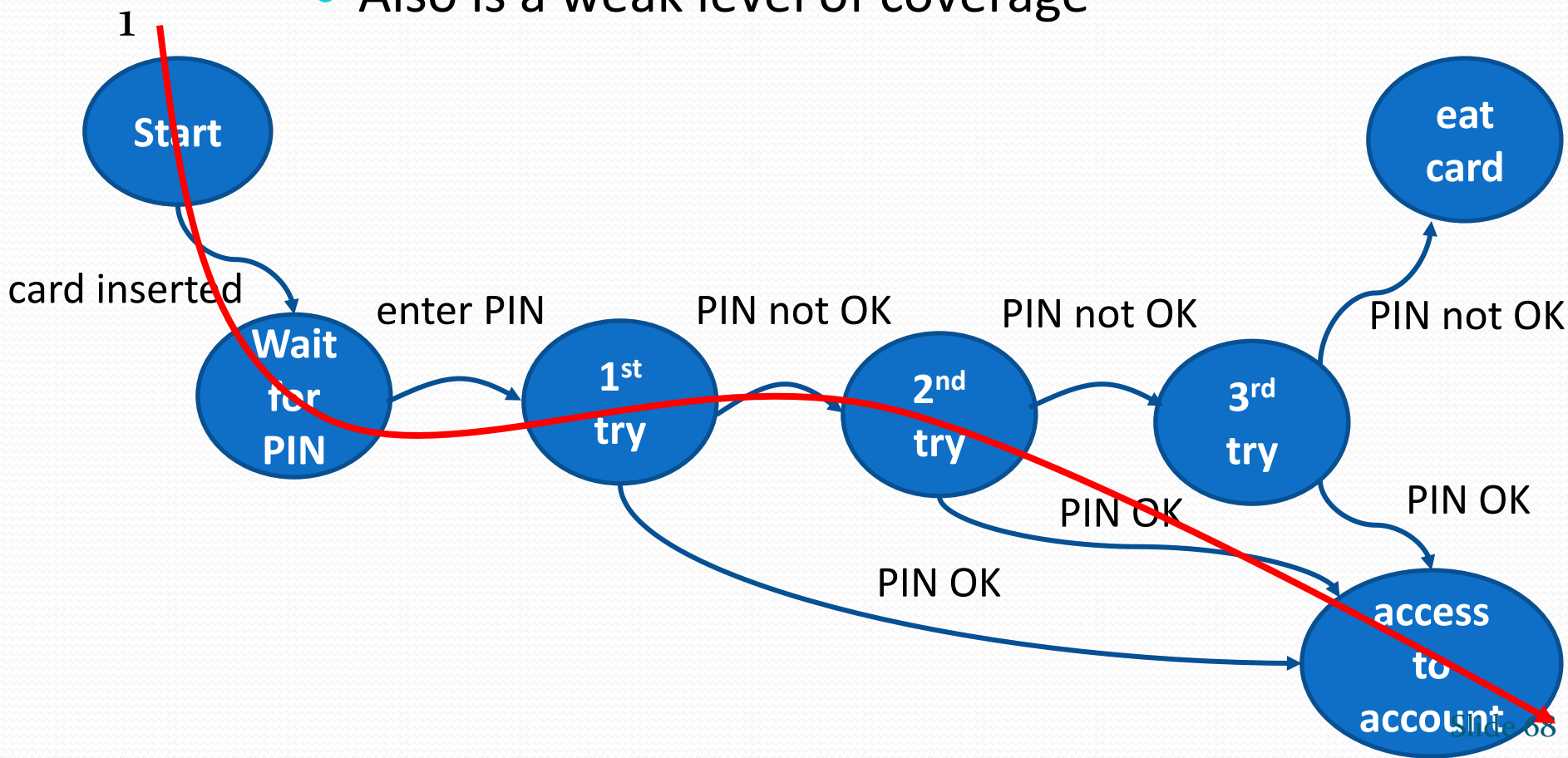
State coverage

- All states are visited at least once
- Generally this is a weak level of test coverage



Event coverage

- All events are triggered at least once
- Also is a weak level of coverage



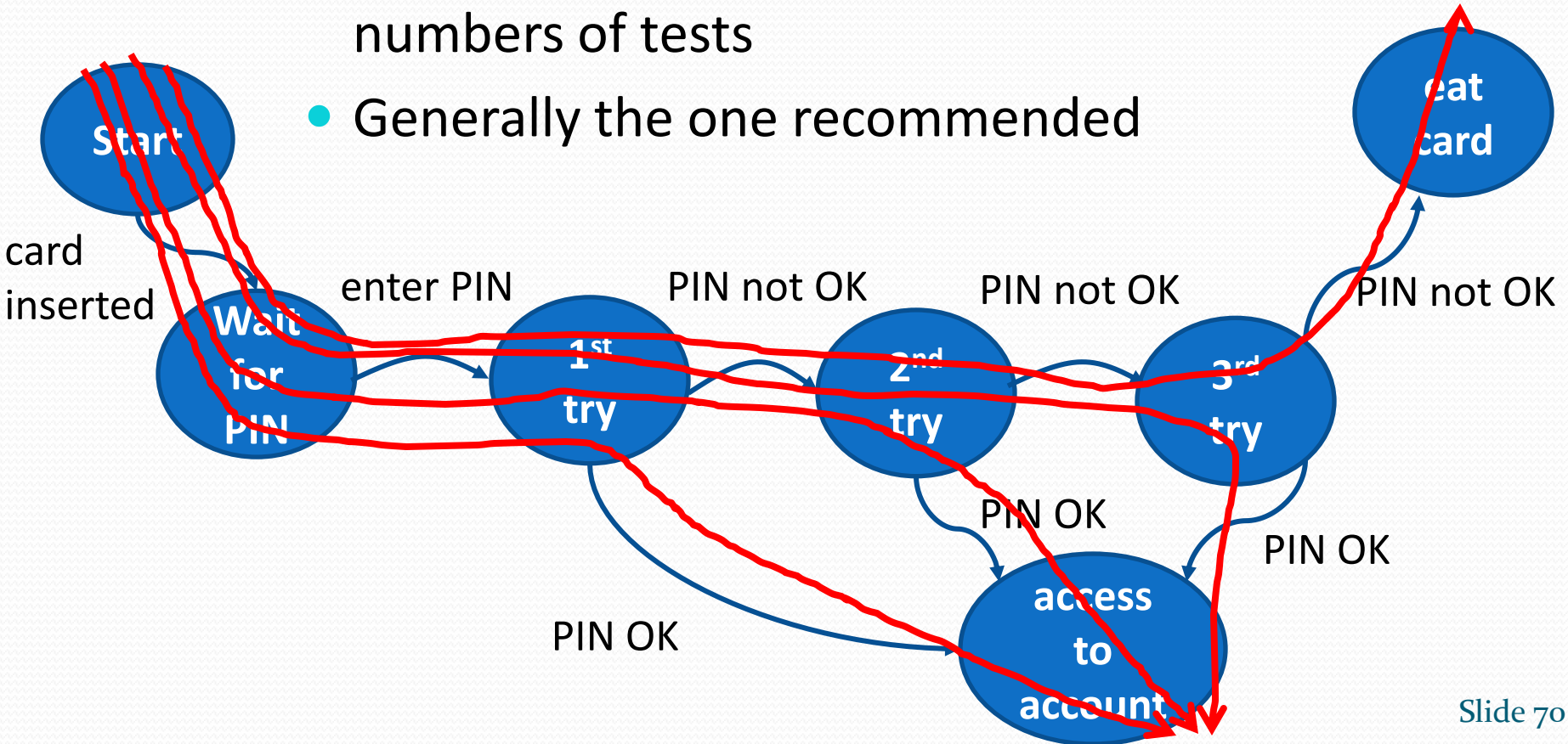
Path coverage

- All paths are executed at least once
- The strongest level of coverage but may not be feasible
- If the state transition diagram has loops, then the number of possible paths may be infinite
  - e.g. 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.

Transition coverage

- All transitions are exercised at least once
- Good level of coverage without generating large numbers of tests
- Generally the one recommended



Testing for invalid transitions

- Using **state table** as an intermediate step
  - list all the **possible states** and all the **possible events**: states are listed on the left side of the table, events that cause them on the top (or vice versa)
  - each cell represents the **state system will move to when the corresponding event occurs**
    - this will include events that are not expected to happen in certain states → **invalid transitions** from that state
- Test cases are usually derived from the state table
  - depending on the system risk, create test cases for some or all of the invalid state/event pairs to make sure the system has not implemented invalid paths

State table example

- Example of state table for the PIN entering

States \ Events	Insert card	Enter valid PIN	Enter invalid PIN
S1) Start	S2	-	-
S2) 1st try	-	S5	S3
S3) 2nd try	-	S5	S4
S4) 3rd try	-	S5	S6
S5) Access account	-	-	-
S6) Eat card	S1 (for new card)	-	-

Design test cases

#	Test Step/Substep	Expected Result
	In all cases, the ATM starts in the waiting for PIN	
1	1. Insert card	Prompts for PIN
	2. Enter valid PIN	Access account
2	1. Insert card	Prompts for PIN
	2. Enter invalid PIN (1st try)	Reprompts for PIN
	3. Enter valid PIN	Access account
3	…	…

State transition testing exercise

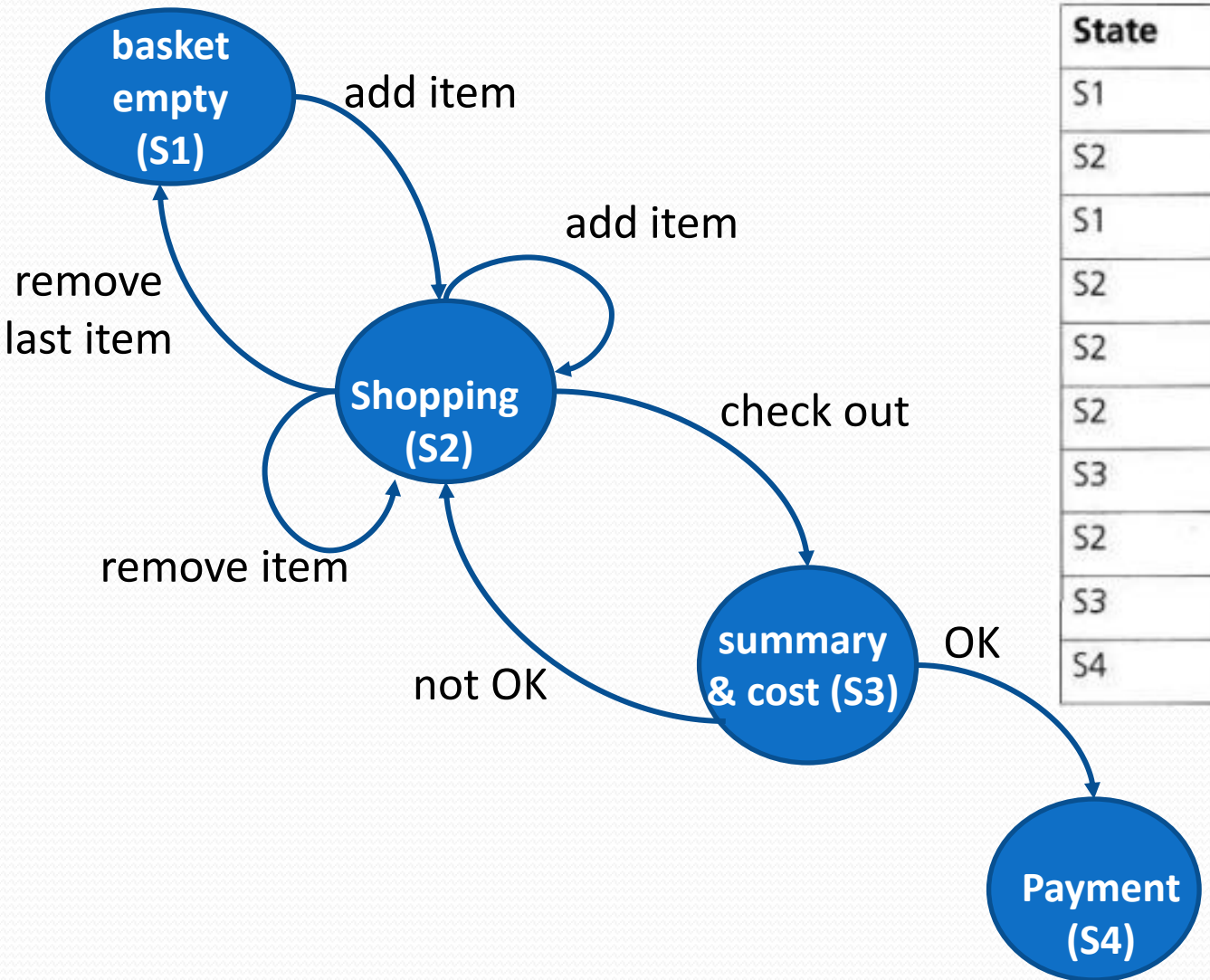
- Scenario: A website shopping basket starts out as empty. As purchases are selected, they are added to the shopping basket. Items can also be removed from the shopping basket. When the customer decides to check out, a summary of the items in the basket and the total cost are shown. If the contents and price are OK, then you leave the summary display and go to the payment system. Otherwise you go back to shopping (so you can remove items if you want).

  a. Produce a state diagram showing the different states and transitions. Define a test, in terms of the sequence of states, to cover all transitions.

  b. Produce a state table. Give an example test for an invalid transition.

Solution - State diagram

Design a test cover all transitions?



State	Event (action)
S1	Add item
S2	Remove (last) item
S1	Add item
S2	Add item
S2	Remove item
S2	Check out
S3	Not OK
S2	Check out
S3	OK
S4	

Solution - State table

	Add item	Remove item	Remove last item	Check out	Not OK	OK
S1) Empty	S2	-	-	-	-	-
S2) Shopping	S2	S2	S1	S3	-	-
S3) Summary	-	-	-	-	S2	S4
S4) Payment	-	-	-	-	-	-

A test for an invalid transition?

- try to remove an item from the empty shopping basket
- attempt to add an item from the summary and cost state

Applicability and Limitations

- State-Transition diagrams are not applicable when the system has no state or does not need to respond to real-time events from outside of the system

Use case testing

- A technique that helps identify **test cases that cover the whole system**, on a transaction by transaction, from start to finish
- Use case is a sequence of steps that describe the interactions between the **actor** and the **system** in order to achieve a specific task
- At least one test case for the **main success scenario**
- At least one test case for each **extension**
- Used widely in developing tests at **system** or **acceptance level**

Use case testing

Use case component	Description	
Main success scenario **A: Actor** **S: System**	Step	Description
	1	A: Inserts card
	2	S: Validates card and ask for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extension	2a	Card not valid S: Displays message and rejects card
	4a	PIN not valid S: Displays message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eats card and exit

Black-box techniques - Advantages

- More effective on larger units of code than glass box testing
- Tester needs no knowledge of implementation, including specific programming languages
- Tester and programmer are independent of each other
- Tests are done from a user's point of view
- Will help to expose any ambiguities or inconsistencies in the specifications
- Test cases can be designed as soon as the specifications are complete

Black-box techniques - Disadvantages

- Only a small number of possible inputs can actually be tested
- May leave many program paths untested
- Without clear and concise specifications, test cases are hard to design
- There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried

Learn more

- Pairwise testing
- Domain analysis testing

1 Overview	2 Life cycle components	3 Infrastructure components	4 Management components	5 Standards and Organizing
6 Static tesing	7 Dynamic testing	8 Test management	9 Tools	

Dynamic techniques (cont.)

References

- Dorothy Grahamet, Erik van Veenendaal, Isabel Evans, Rex Black. *Foundations of software testing: ISTQB Certification*

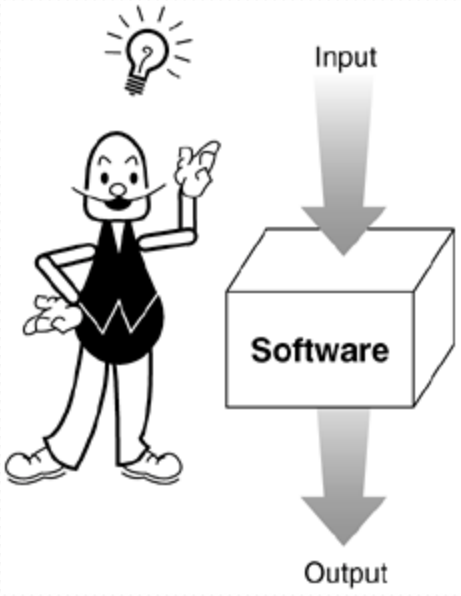
- Lee Copeland (2004). *A Practitioner's Guide to Software Test Design*. Artech House. ISBN:158053791x

Contents

White-box techniques

- Structure-based approach
  - based on the internal structure of a component or system
  - also called glass-box techniques

- What we may be interested in structures?
  - component level: program structures
  - integration level: the way components interact with others
  - system level: how user will interact with the system

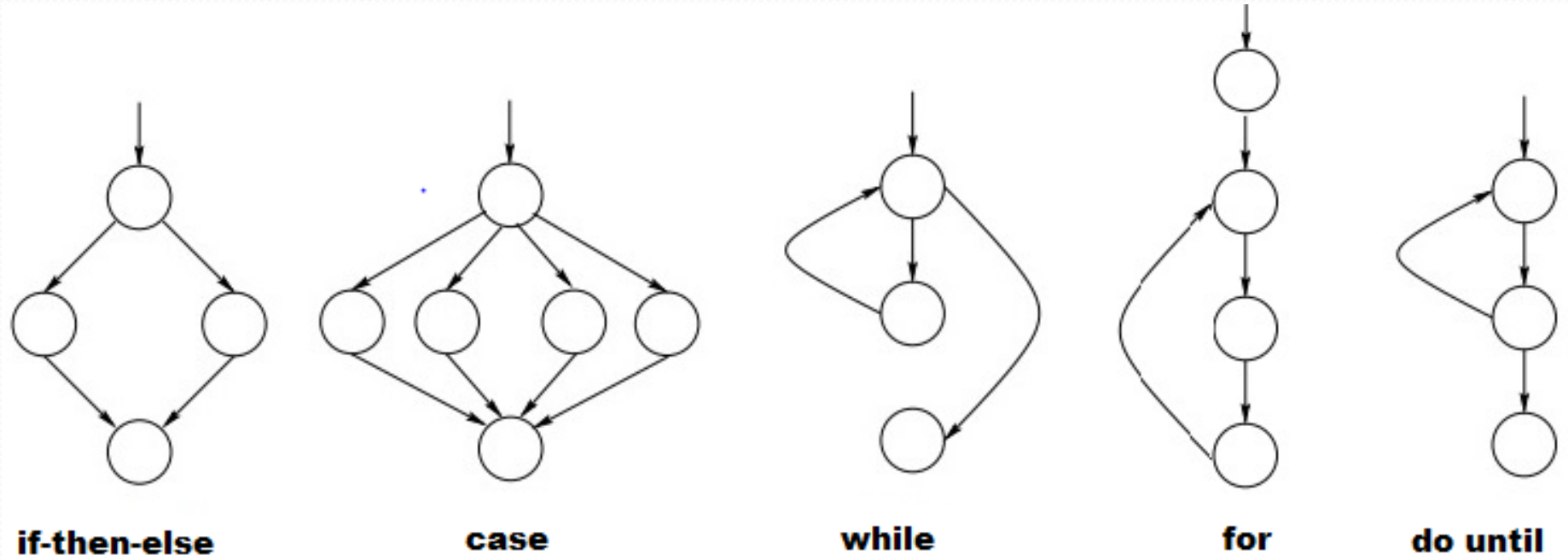


White-box techniques

- Some techniques
  - control flow testing
    - statement testing
    - decision testing
    - condition testing
    - decision/condition testing
    - multiple condition testing
    - path testing
  - data flow testing

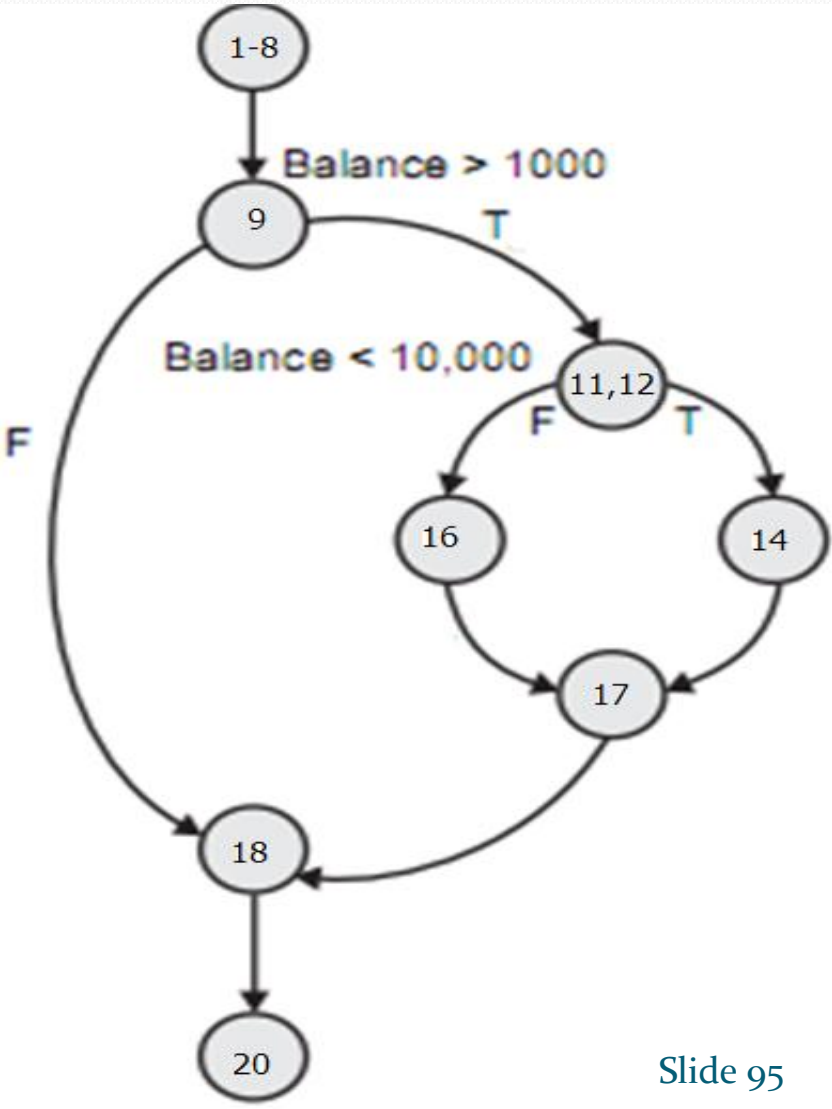
Control-flow graph

- Flow graphs for control structures



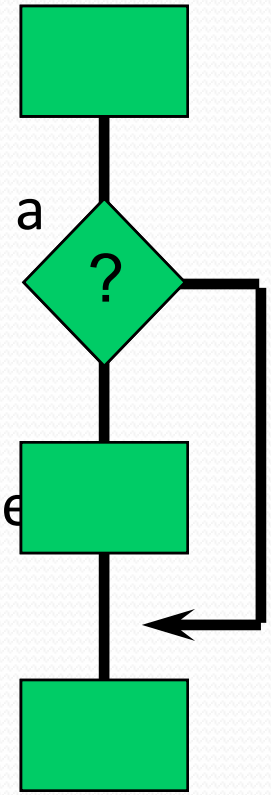
Control-flow graph - Example

```
1    Program BestInterest
2    Interest, Base Rate, Balance: Real
3
4    Begin
5    Base Rate = 0.035
6    Interest = Base Rate
7
8    Read (Balance)
9    If Balance > 1000
10   Then
11       Interest = Interest + 0.005
12       If Balance < 10000
13       Then
14           Interest = Interest + 0.005
15       Else
16           Interest = Interest + 0.010
17       Endif
18   Endif
19
20   Balance = Balance × (1 + Interest)
21
22   End
```



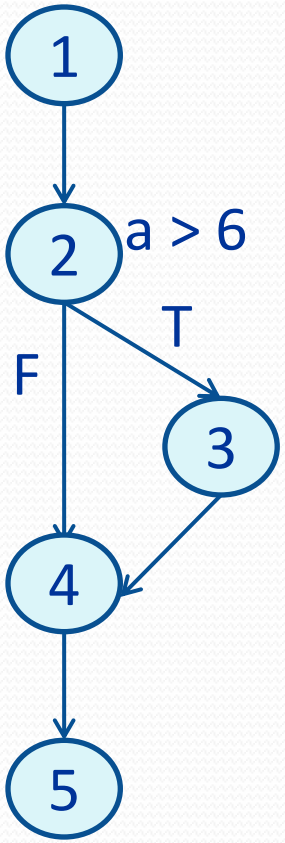
Statement testing (Level 1)

- *A test design technique in which test cases are designed to execute statements* [ISTQB Glossary]
- Statement coverage
  - the percentage of executable statements exercised by a test suite
    - $= \frac{Number\ of\ statements\ exercised}{Total\ number\ of\ statements} x100\%$
  - black-box testing: only 60% to 75% statement coverage
  - typical ad hoc testing achieves 30%
- How to get 100% statement coverage?
  - Find out the shortest number of paths which all the nodes will be covered



Statement coverage example 1

```
1  read(a)
2  IF a > 6 THEN
3     b = a
4  ENDIF
5  print b
```



How many test cases to get 100% statement coverage?

#	Condition	Input	Line number excuted	Expected result
1	a>6	7	1,2,3,4,5	7 (not from code)

Statement coverage example 2

```
1. public int MaxAndMean(int A, int B, int C, out double Mean)
2. {
3.     Mean = (A + B + C) / 3.0;
4.     int Maximum;
5.     if (A > B)
6.         if (A > C)
7.             Maximum = A;
8.         else
9.             Maximum = B;
10.    else
11.        if (B > C)
12.            Maximum = B;
13.        else
14.            Maximum = C;
15.  return Maximum;
16. }
```

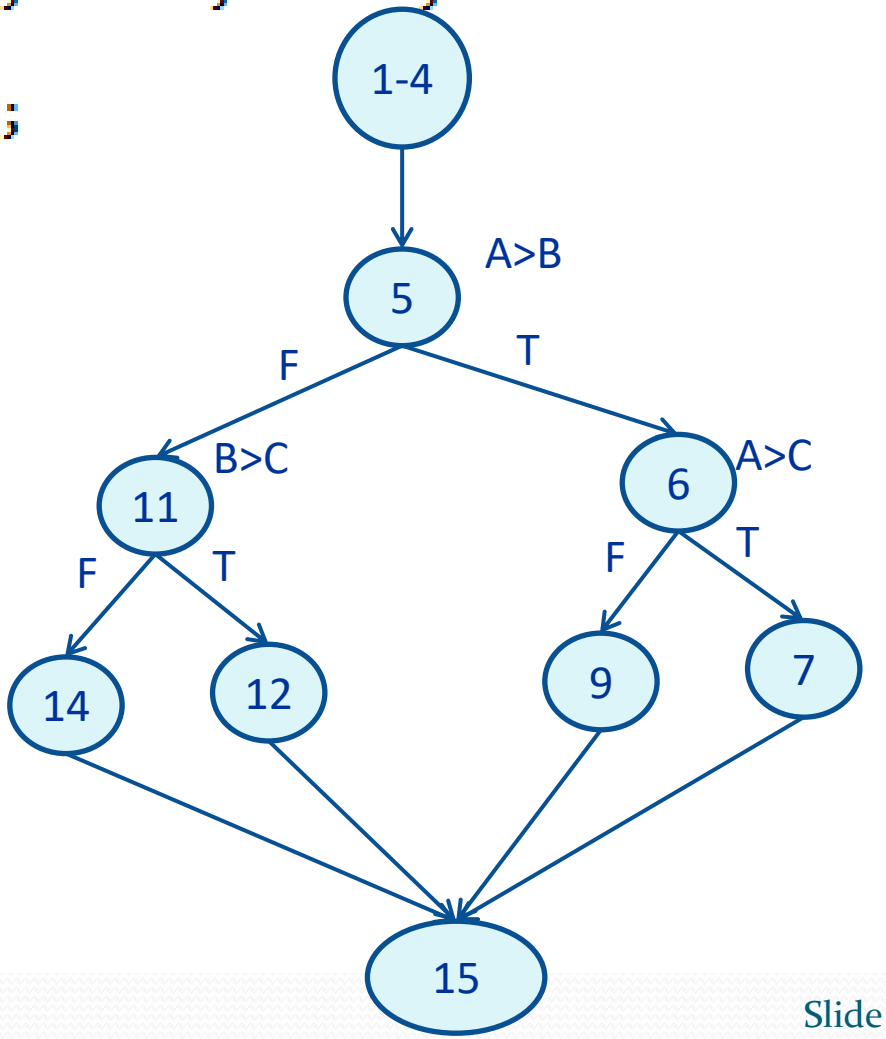
A program for calculating the mean and maximum of three integers.

a. How many test cases will you need to achieve 100% statement coverage?

b. What will the test cases be?

Statement coverage example 2 - Solution

```
1. public int MaxAndMean(int A, int B, int C, out double Mean)
2. {
3.     Mean = (A + B + C) / 3.0;
4.     int Maximum;
5.     if (A > B)
6.         if (A > C)
7.             Maximum = A;
8.         else
9.             Maximum = B;
10.    else
11.        if (B > C)
12.            Maximum = B;
13.        else
14.            Maximum = C;
15.  return Maximum;
16. }
```



Statement coverage example 2 - Solution

- Design test case

#	Test condition	Input			Line number executed	Expected result	
		A	B	C		Max	Mean
1	A>B and A>C	5	2	3	1-4,5,6,7,15	5	3.33
2	A>B and A<=C	5	2	7	1-4,5,6,9,15	7	4.66
3	A<=B and B>C	5	7	4	1-4,5,11,12,15	7	5.33
4	A<=B and B<=C	5	6	8	1-4,5,11,14,15	8	6.33

Statement coverage exercise

```
1    Program Grading
2
3    StudentScore: Integer
4    Result: String
5
6    Begin
7
8    Read StudentScore
9
10   If StudentScore > 79
11   Then Result = "Distinction"
12   Else
13        If StudentScore > 59
14        Then Result = "Merit"
15        Else
16             If StudentScore > 39
17             Then Result = "Pass"
18             Else Result = "Fail"
19             Endif
20        Endif
21   Endif
22   Print ("Your result is", Result)
23   End
```

A program evaluates grades for students:

- **If StudentScore<=39, print "Fail".**
- **If StudentScore between 39 and 60, print "Pass"**
- **If StudentScore between 59 and 80, print "Merit"**
- **If StudentScore>79, print "Distinction"**

a. How many test cases will you need to achieve 100% statement coverage? Design test case.

b. Suppose we ran two test cases: StudentScore = 50 and StudentScore = 30, which lines of pseudo code will not be exercised? How much is statement coverage?

Statement coverage exercise: Solution

Statement coverage problems

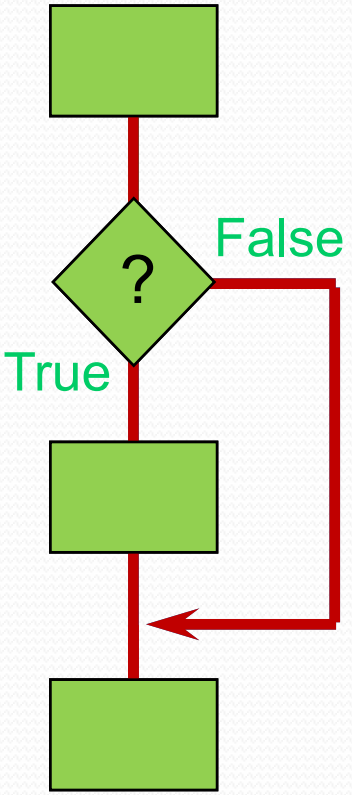
- Statement coverage can be achieved without branch coverage
  - important cases may be missed

Decision testing (Level 2)

- *A test design technique in which test cases are designed to execute **decision outcomes** [ISTQB Glossary]*
  - decision: a logical expression which can be composed of several logical operators like "or", "and", "xor"
  - decision outcome: each exit from a decision
    - two or more possible decision outcomes

IF A > 1 ← **decision**

IF A > 1 AND X = 2



- Known as 'Branch testing', 'Basis path testing'

Decision testing (cont'd)

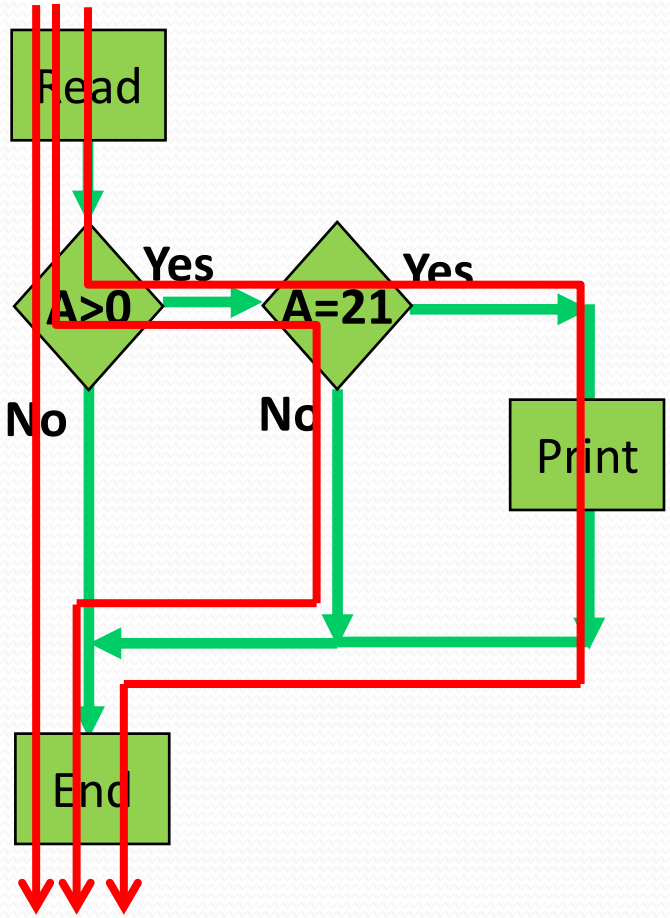
- Decision coverage
  - percentage of decision outcomes exercised

$$= \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decision\ outcomes} x100\%$$

  - specification-based may achieve only 40% to 60% decision coverage
  - typical ad hoc testing achieves 20%
  - 100% decision coverage guarantees 100% statement coverage, but not vice versa
- How to get 100% decision corverage?
  - Find out the minimum number of paths which will ensure covering of all the edges

Decision testing example 1

```
Read A
IF A > 0 THEN
    IF A  = 21 THEN
        Print "Key"
    ENDIF
ENDIF
```



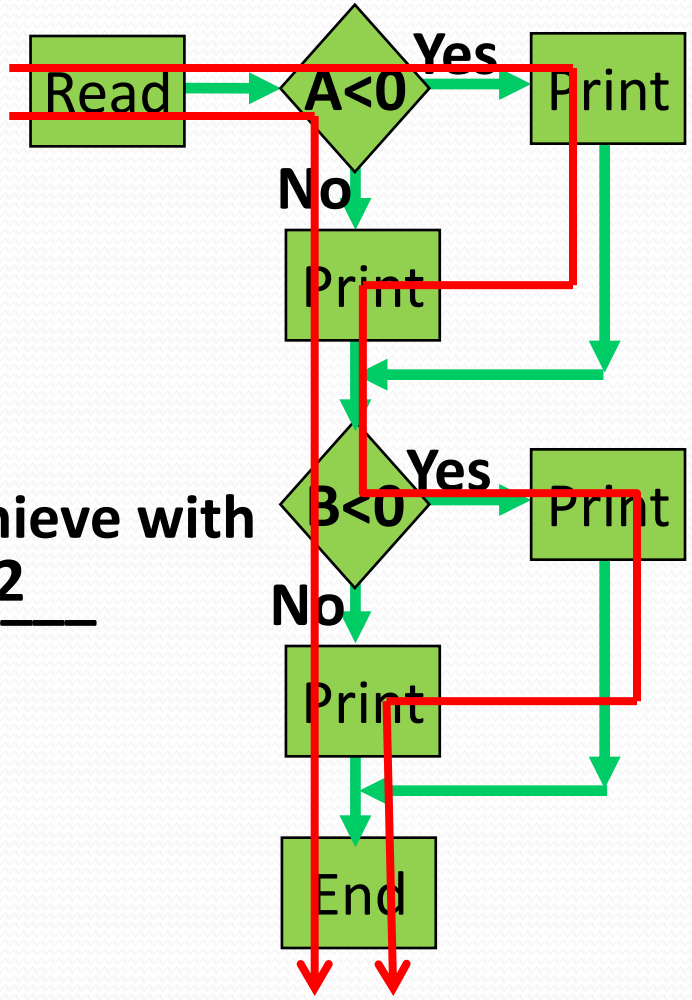
- **Minimum tests to achieve with decision coverage: 3**

#	Cases	Inputs	Expected result
1	A>0(F)	A=-5	No message
2	A>0(T) and A=21(F)	A=10	No message
3	A>0(T) and A=21(T)	A=21	Message "Key"

Decision testing example 2

```
Read A
Read B
IF A < 0 THEN
    Print  "A negative"
ELSE
    Print  "A positive"
ENDIF
IF B < 0 THEN
    Print  "B negative"
ELSE
    Print  "B positive"
ENDIF
```

- **Minimum tests to achieve with decision coverage: 2**



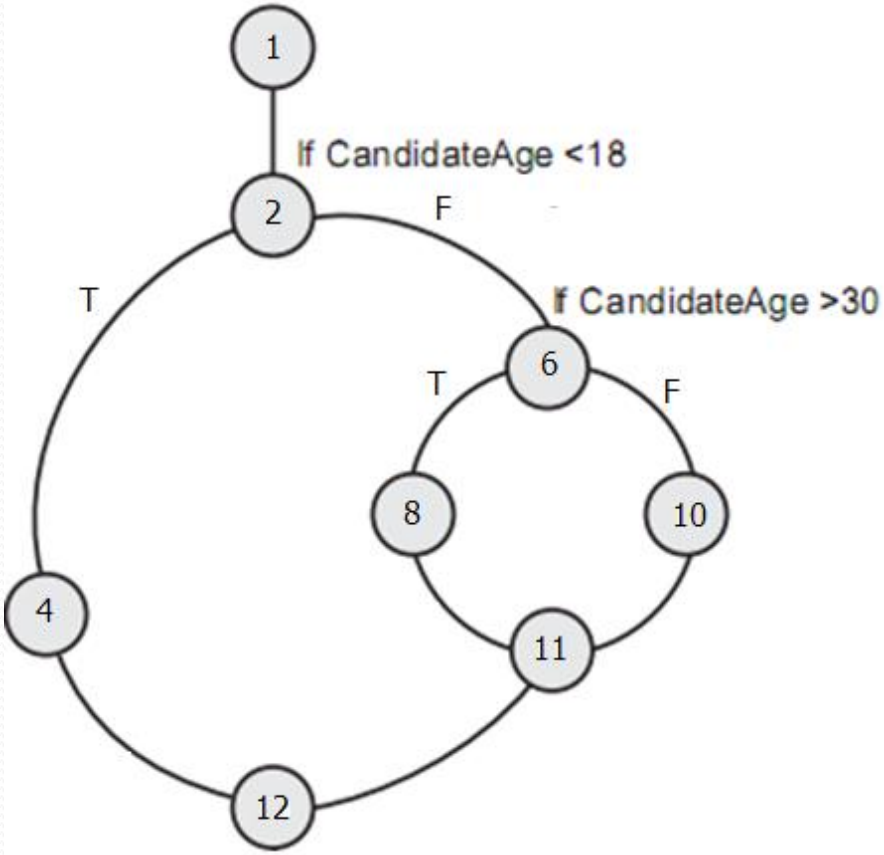
Decision testing exercise 1

```
1   Read(CandidateAge)
2   If  CandidateAge < 18
3   Then
4      Print (“Candidate is too young”)
5   Else
6    If  CandidateAge > 30
7    Then
8      Print (“Candidate is too old”)
9    Else
10     Print(“Candidate may join Club 18–30”)
11   Endif
12  Endif
```

How many test cases for 100% decision coverage?

Solution exercise 1

```
1   Read(CandidateAge)
2   If  CandidateAge < 18
3   Then
4       Print ("Candidate is too young")
5   Else
6     If  CandidateAge > 30
7     Then
8       Print ("Candidate is too old")
9     Else
10      Print("Candidate may join Club 18–30")
11    Endif
12  Endif
```



CandidateAge<18

CandidateAge>=18 and CandidateAge>30

CandidateAge>=18 and CandidateAge<=30

Decision testing exercise 2

```
1    Begin
2    Read Time
3    If Time < 12 Then
4        Print(Time, "am")
5    Endif
6    If Time > 12 Then
7        Print(Time -12, "pm")
8    Endif
9    If Time = 12 Then
10       Print (Time, "noon")
11   Endif
12   End
```

a. How many test cases are needed to achieve 100% decision coverage?

b. If the test cases Time = 11 and Time = 15 were input, what level of decision coverage would be achieved?

a. 3 test cases

b. 83%

Solution exercise 2

Decision testing exercise 3

```
7    Index = 0
8    Sum = 0
9    Read (Count)
10   Read (New)
11
12   While Index <= Count
13   Do
14       If New < 0
15       Then
16           Sum = Sum + 1
17       Endif
18       Index = Index + 1
19       Read (New)
20   Enddo
21
22   Print ("There were", Sum, "negative numbers in the input
     stream")
23
24   End
```

What test case for 100% decision coverage?

Solution exercise 3

Test

Suppose you are developing a software unit that will convert a non-signed 16 bit binary number (in string format) to a decimal integer. For example, BinaryToDecimal("0000000000001111") = 15. Draw CFG for this function.

```
1. public long BinaryToDecimal(string sbin)
2. {
3.     int sum = 0;
4.     int tpow = 1;
5.     const int MAX_BITS = 16;
6.     if (sbin.Length > MAX_BITS)
7.         throw new OverflowException("The number is too big.");
8.     for (int i = sbin.Length - 1; i >= 0; i++)
9.     {
10.        if (sbin[i] == '1')
11.            sum = sum + tpow;
12.        else if (sbin[i] != '0')
13.            throw new FormatException("Invalid binary format.");
14.        tpow = 2 * tpow;
15.    }
16.    return sum;
17. }
```

Decision testing problems

- Some branching decisions in programs are made not based on a single condition but on **multiple conditions**
  - Decision coverage does not ensure that all entry-exit paths are executed
- A compound predicate is treated as a single statement
  - If n clauses, 2^n combinations, but only 2 are tested
  - Example

```
if (condition1 && (condition2 || function1()))
        statement1;
else
        statement2;
```

Condition testing (Level 3)

- Design test cases based on Boolean sub-expression (BsE): **each condition** to be evaluated as **true and false at least once**
- Condition coverage
  - $= \frac{Number\ of\ BsE\ outcomes\ exercised}{Total\ number\ of\ BsE\ outcomes} x100\%$

Decision/Condition testing (Level 4)

- Full condition coverage does not guarantee full decision coverage
  - Example: if (x&&y) {conditionedStatement;}
  - Using condition coverage, if we choose two test cases (**x=TRUE, y=FALSE** and **x=FALSE, y=TRUE**), the conditionedStatement will never be executed

- Decision/Condition testing: Test cases are created for **every condition** and **every decision**

Multiple condition testing (Level 5)

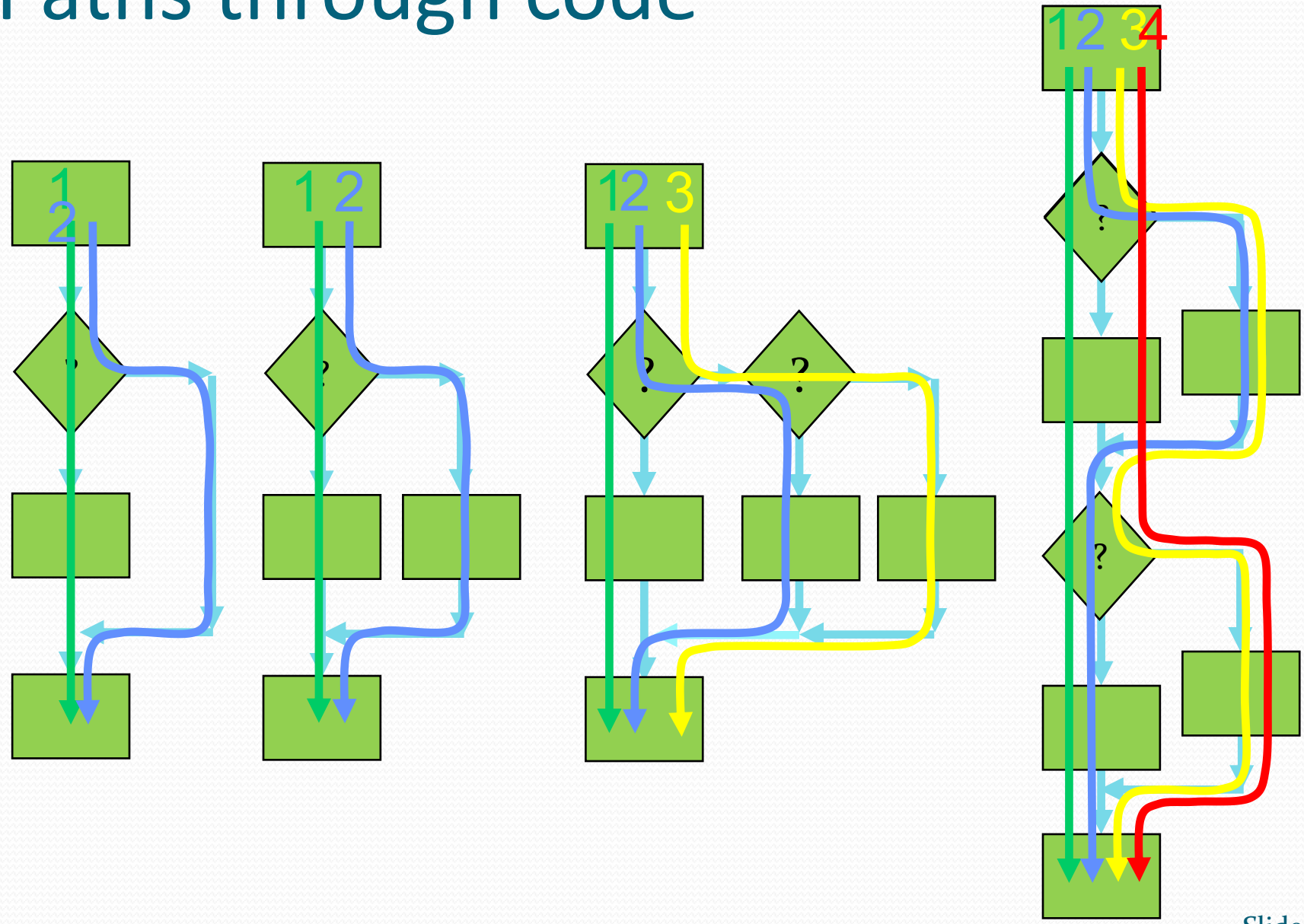
- Known as ‘condition combination testing'
- Requiring 2^n test cases to achieve 100% coverage of a decision containing n boolean operands
- Example: A or (B and C)

Case	A	B	C
1	FALSE	FALSE	FALSE
2	FALSE	FALSE	TRUE
3	FALSE	TRUE	FALSE
4	FALSE	TRUE	TRUE
5	TRUE	FALSE	FALSE
6	TRUE	FALSE	TRUE
7	TRUE	TRUE	FALSE
8	TRUE	TRUE	TRUE

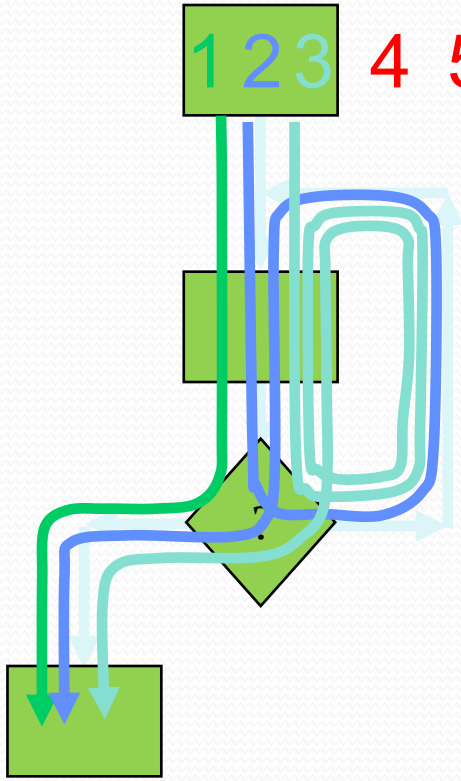
Path testing (Level 6)

- Design test cases based on what **paths** which should be exercised
- Path coverage
  - $= \frac{number\ of\ paths\ exercised}{total\ number\ of\ paths} x100\%$

Paths through code



Paths through code with loops



for as many times as it is possible to go round the loop (this can be unlimited, i.e. infinite)

White box techniques - Advantages

- It permits direct checking of processing paths and algorithms
- It provides line coverage follow-up that delivers lists of lines of code that have not yet been executed
- It is capable of testing the quality of coding work

White box techniques - Disadvantages

- It requires vast resources, much above those required for black box testing
- It cannot test the performance of software in terms of availability, reliability, stress, etc.
- The tester must have sufficient programming skill to understand the code and its control flow
- Control flow testing can be very time consuming because of all the modules and basis paths that comprise a system

Unit Testing

- **In software engineering**, **unit testing** is a test (often automated) that validates that individual units of source code are working properly.
- **A unit** is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a method, which may belong to a base / super class, abstract class or derived / child class.

Unit Testing

Contents

Non-systematic test techniques

- Based on a person's knowledge, experience, imagination and intuition
- Some techniques
  - error guessing
  - exploratory testing

It is true that testing should be rigorous, thorough and systematic

This is not all there is to testing

Error guessing

- No rules, no script
- Think of situations in which the software may not be able to cope
  - division by zero
  - blank (or no) input
  - empty files
  - wrong kind of data (e.g. alphabetic characters where numeric are required)...
- After systematic techniques have been used
- Supplements systematic techniques

Not a good approach to start testing with

Exploratory testing

This testing helps improving quality

"A style of testing in which you **explore the software** while simultaneously designing and executing tests, using feedbacks from the last test to inform the next." (Elisabeth Hendrickson)

Exploratory testing as 'an interactive process of simultaneous learning, test design, and test execution' (James Bach)

Contents

Dynamic techniques

Test condition – Test case – Test procedure

Black-box techniques

White-box techniques

Experience-based techniques

Choosing test techniques

Choosing test techniques

- Each technique is good for certain things, and not as good for other things
- The best testing technique is no single testing technique: each testing technique is good at finding one specific class of defect
- How can we choose the most appropriate testing techniques to use? - Depend on internal and external factors

Choosing test techniques (cont'd)

- Internal factors
  - models used
  - tester knowledge or experience
  - test objective
  - documentation
  - life cycle model

Choosing test techniques (cont'd)

- External factors
  - risk
  - customer or contractual requirements
  - type of system
  - time and budget
  - regulatory requirements