

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

**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

1	2	3	4	5
6	7	8	9	

# Contents

## Dynamic techniques

Test condition – Test case – Test procedure

Black-box techniques

White-box techniques

Experience-based techniques

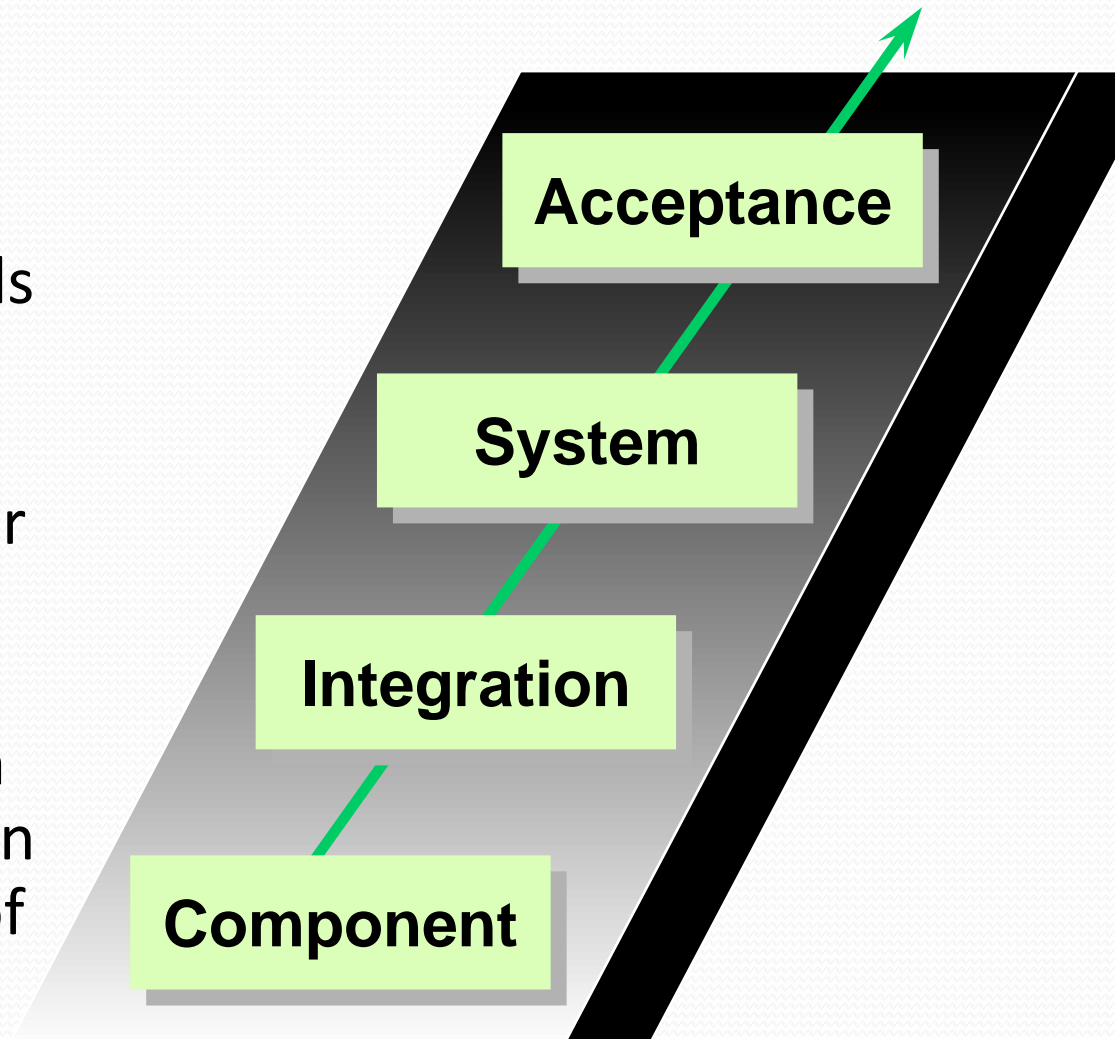
Choosing test techniques

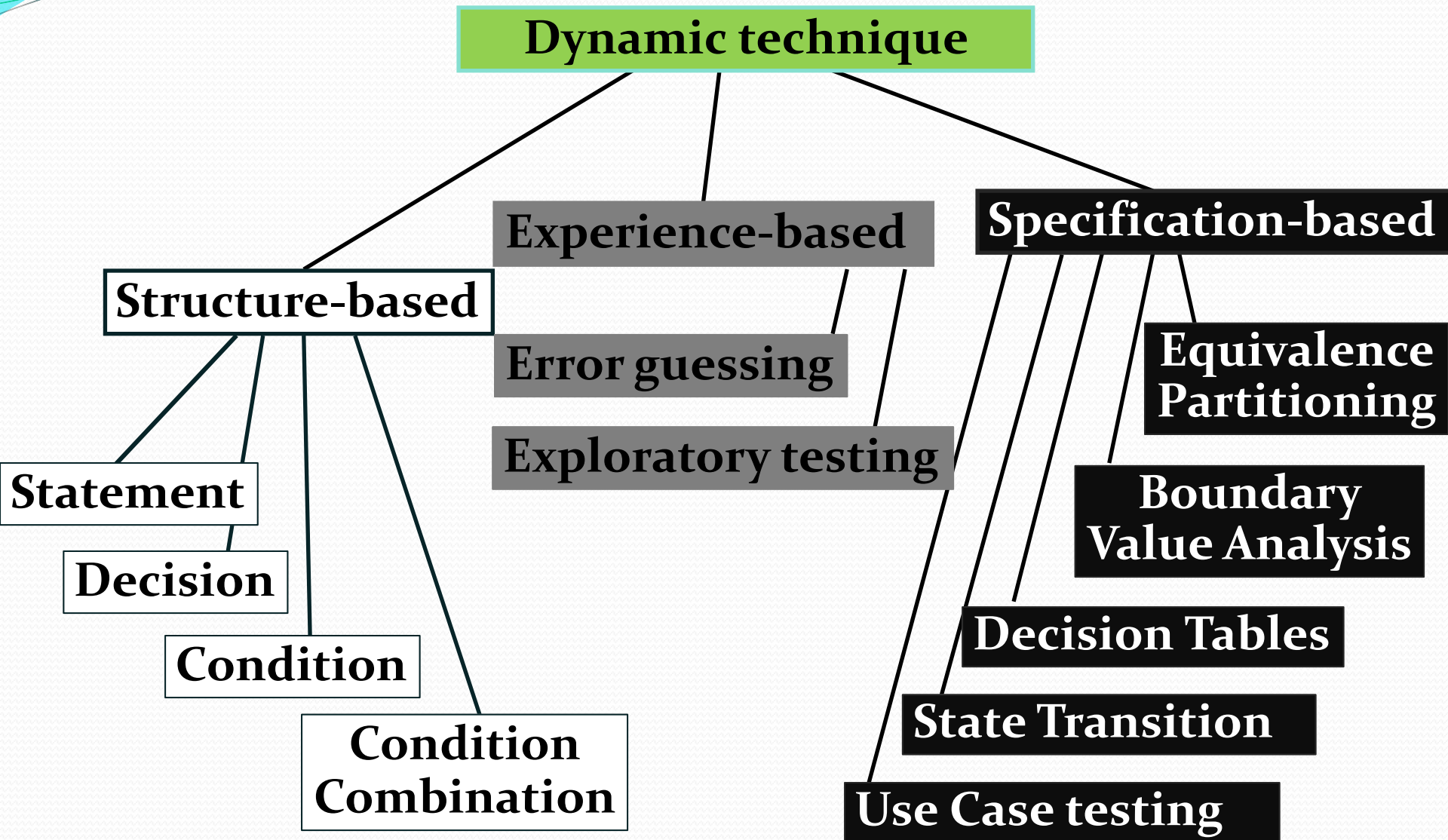
# Categories of dynamic techniques

- 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- **Experience-based techniques** used when there is no specification or inadequate or out of date





1	2	3	4	5
6	7	8	9	

# Contents

Dynamic techniques

Test condition – Test case – Test procedure

Black-box techniques

White-box techniques

Experience-based techniques

Choosing test techniques



# Test process

## Planning and Control

**Analysis &  
Design**

**Execution**

**Evaluating  
Reporting**

**Check  
completion**

**Identify conditions**

**Design test cases**

**Specify test procedures**

# 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
<b>Check Login functionality</b>	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

1	2	3	4	5
6	7	8	9	

# Contents

Dynamic techniques

Test condition – Test case – Test procedure

Black-box techniques

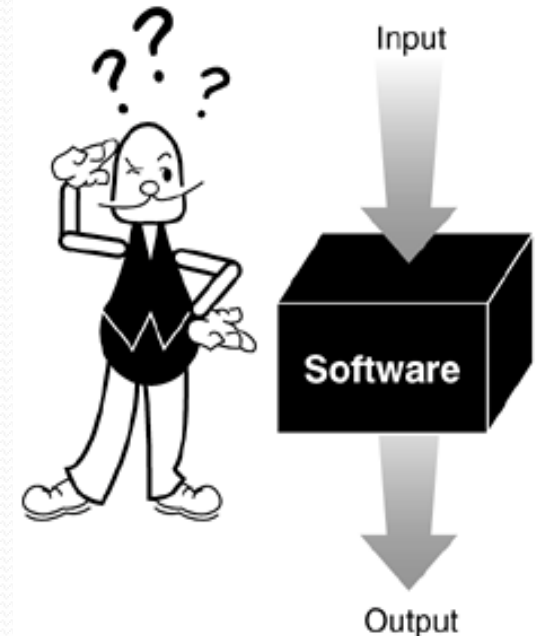
White-box techniques

Experience-based techniques

Choosing test techniques

# Black-box techniques

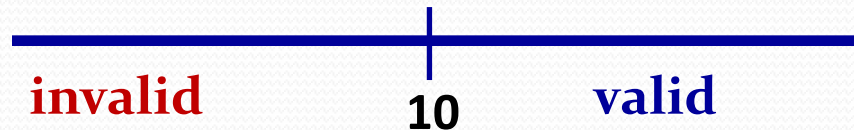
- Based on **specifications** or **models** of what the system should do
- Known as **specification-based techniques**
- Including both functional and non-functional aspects
- Some techniques
  - equivalence partitioning
  - boundary value analysis
  - decision tables testing
  - state transition testing
  - 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” →  
 $\{\text{integer } x \mid 100 \leq x \leq 200\}$ ,  
 $\{\text{integer } x \mid x < 100\}$ ,  
 $\{\text{integer } x \mid x > 200\}$
- Specific *value* within a range → one valid and two invalid equivalence classes  
“value of integer x shall be 100” →  
 $\{\text{integer } x \mid x = 100\}$ ,  
 $\{\text{integer } x \mid x < 100\}$ ,  
 $\{\text{integer } x \mid x > 100\}$

# Equivalence partitioning

## Guidelines

- Set of values → one valid and one invalid equivalence class  
“weekday x shall be a working day” →  
 $x \in \{\text{Monday, Tuesday, Wednesday, Thursday, Friday}\}$ ,  
 $x \in \{\text{Saturday, Sunday}\}$
- 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

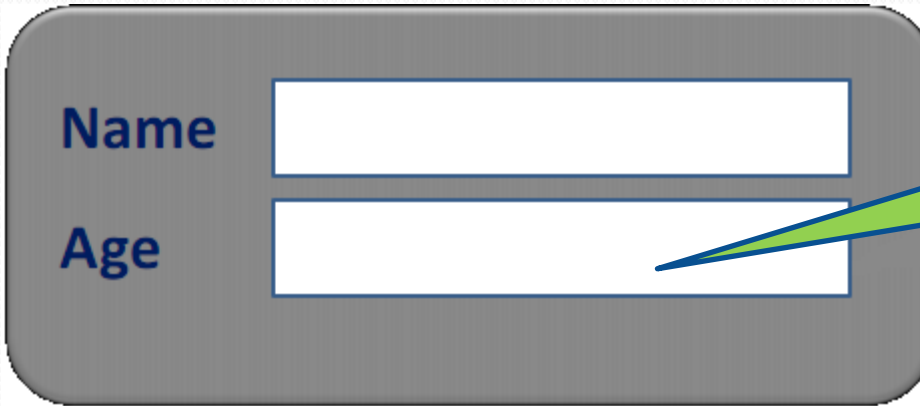
- *Boolean* → one valid and one invalid equivalence class  
“condition x shall be true” →  
x = true, x = false
- One or several equivalence classes for *illegal* values, that is, for values that are *incompatible with the type* of the input parameter and therefore out of the parameter's domain  
“integer values x” →  
{real-number x}, {character-string x}

# Equivalence partitioning

## Guidelines

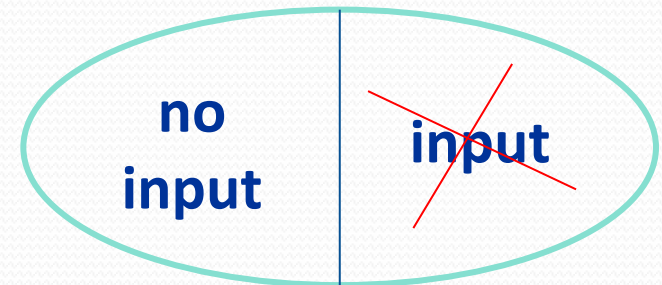
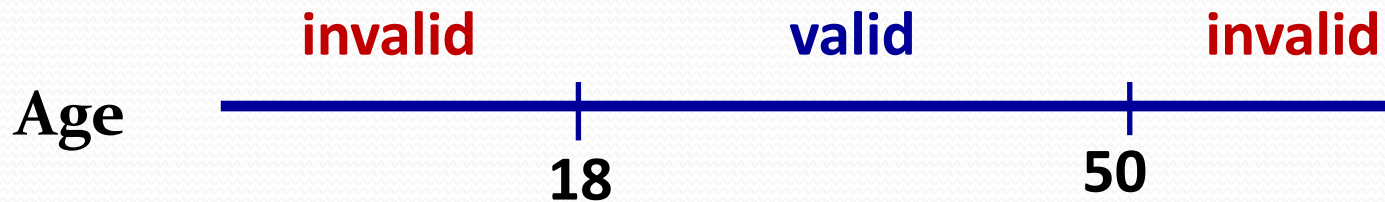
- If an input condition specifies a '*must be*' situation  
“first character of the identifier must be a letter” →  
{first character is a letter}, {first character is not a letter}
- Equivalence classes can be of the output desired in the program
- If there is reason to believe that the system handles each valid/invalid/illegal input value differently, then each value shall generate an equivalence class

# Example EP 1



A form with two input fields. The first field is labeled "Name" and the second is labeled "Age". Both fields are empty and have a blue border.

Expected  
from 18 to 50



# 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						

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For invalid partition						

# 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.

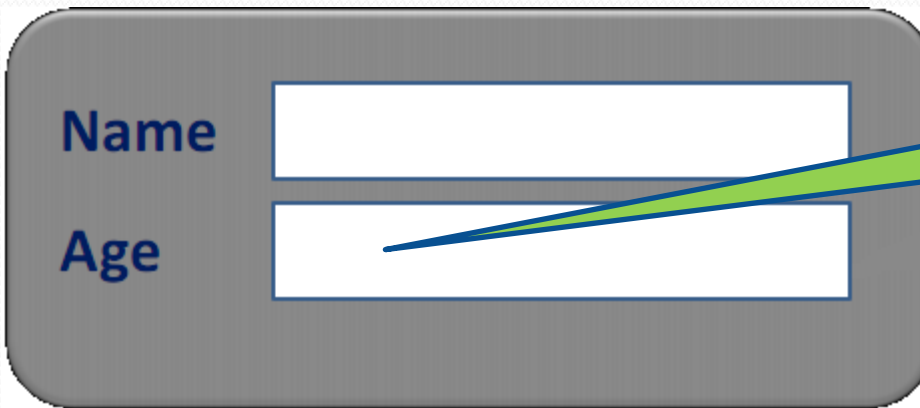
# Boundary value analysis (BVA)

- 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



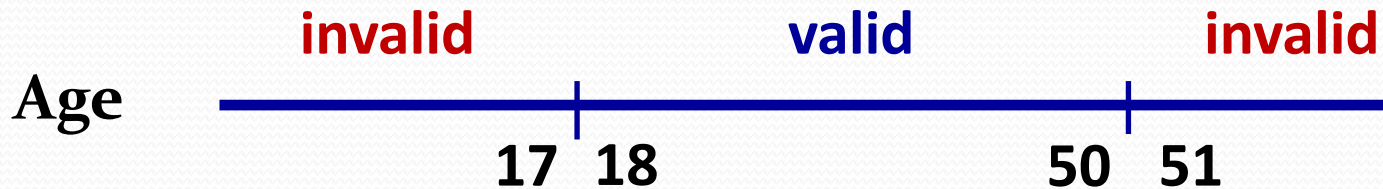
**Boundary values: 0, 1, 100, 101**

# Example BVA 1



A gray rounded rectangle containing two white input fields. The top field is labeled 'Name' and the bottom field is labeled 'Age' in blue text. A green callout bubble points to the 'Age' field.

**Expected  
from 18 to 50**



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		
Visitor's age		

Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For valid boundary	1					
	2					
	3					
	4					
	5					
	6					

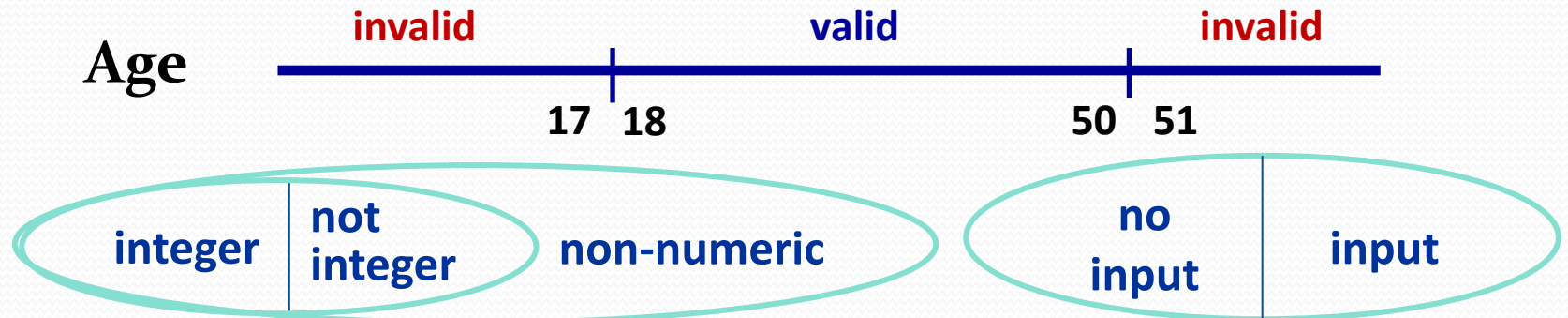
Test case type	Test case no.	Day of week	Visitor's status	Entry hour	Visitor's age	Test case results
For invalid boundary	1					
	2					
	3					
	4					

# Example EP - BVA

Name

Age

Expected  
from 18 to 50



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	<input type="text"/>	2-64 chars
Account number	<input type="text"/>	6 digits, 1 <sup>st</sup> non-zero
Loan amount requested	<input type="text"/>	£500 to £9000
Term of loan	<input type="text"/>	1 to 30 years
<hr/>		
Repayment:	<input type="text"/>	
Interest rate:	<input type="text"/>	
Total paid back:	<input type="text"/>	



# Customer name



2-64 chars

# Account number

6 digits, 1<sup>st</sup>  
non-zero

# Loan amount

£500 to £9000

# Term of loan

1 to 30 years

# Condition template

# 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

# BVA with 'open boundary'

- One of the sides of the partition is not defined
- How to test?
  - go back to the specification
  - investigate other related areas of the system
  - probably need to use an intuitive or experience-based 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

# Exercise 3: Solution - Design test cases

# 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
R <sub>1</sub>	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**, you get a 10% discount.

If you have a **coupon**, 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

# Decision tables testing example 2

# 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
<b>Conditions</b>				
Condition-1	0–1	1–10	10–100	100–1000
Condition-2	<5	5	6 or 7	>7
<b>Actions</b>				
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

# Solution – test cases

# 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 different 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**

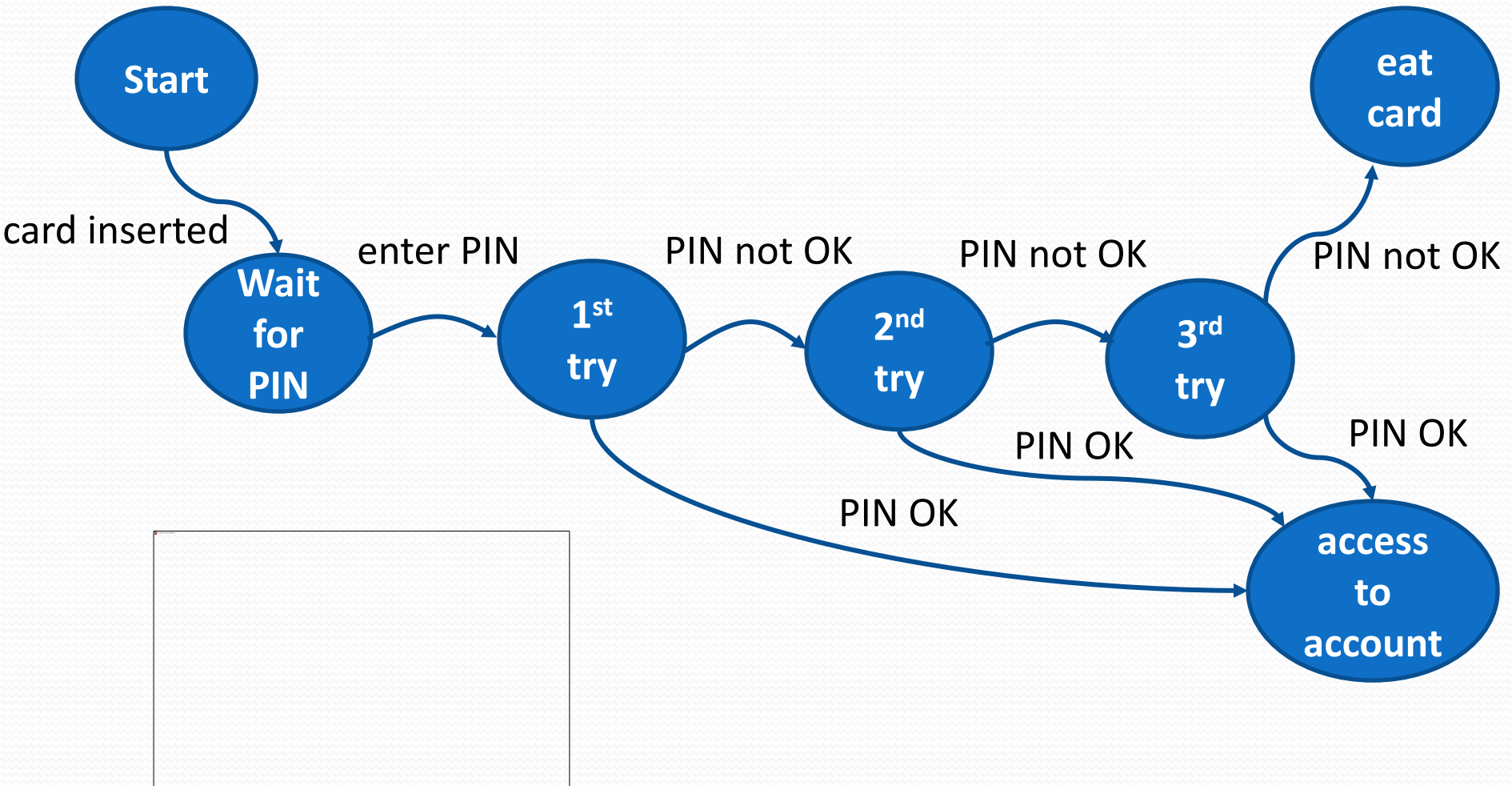
# Example

- Entering a PIN to a bank account



# State transition diagram

- Example State diagram for enter PIN



# State transition diagram

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

1) **states** software may occupy



Start

A blue oval containing the word "Start" in white text.

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



3) **events** that cause transition

PIN not OK

4) **actions** that result from transition



eat  
card

A blue oval containing the text "eat" and "card" in white text, stacked vertically.

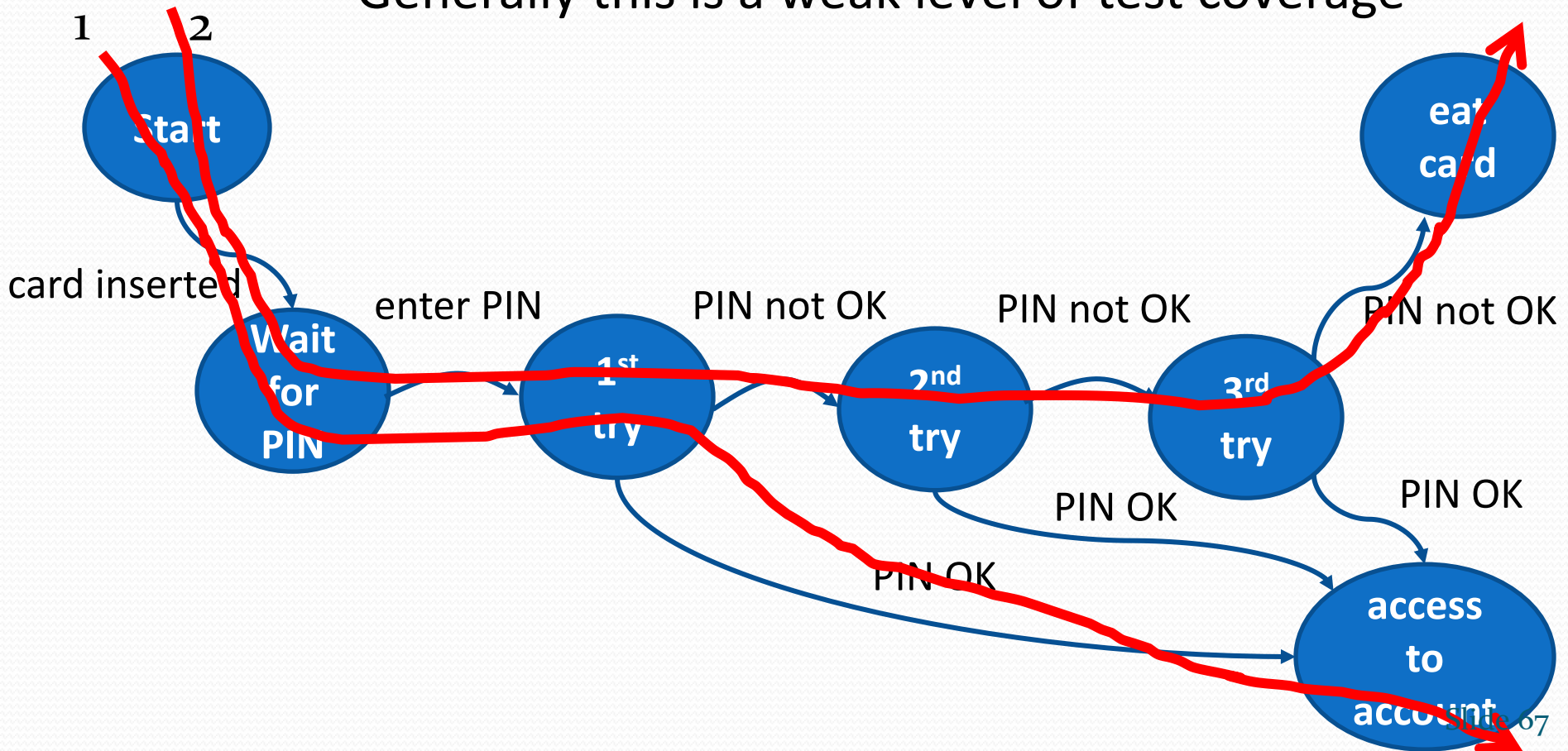


# Creating test cases

- 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

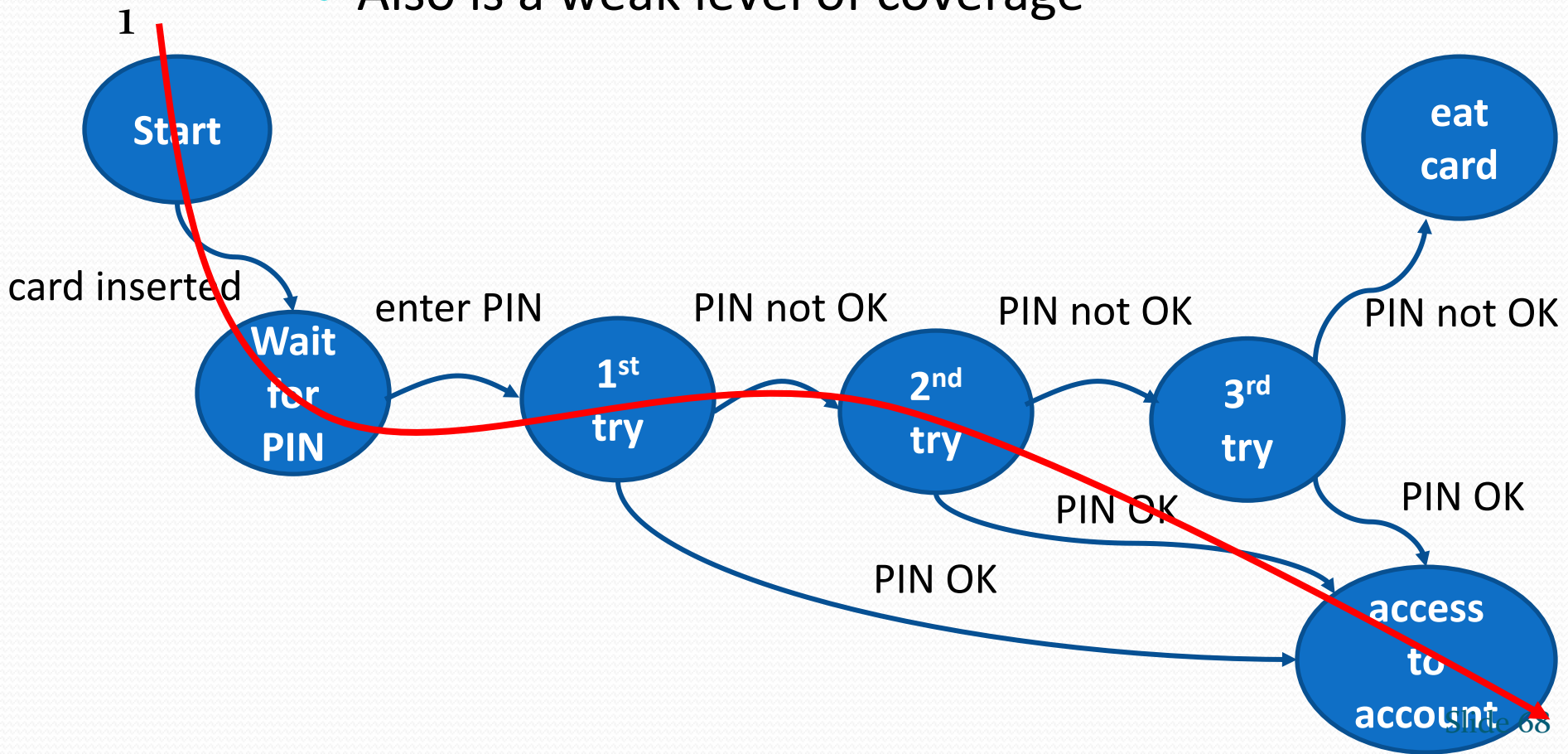
# State coverage

- 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:

$A \rightarrow B$

$A \rightarrow B \rightarrow A$

$A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \rightarrow B$

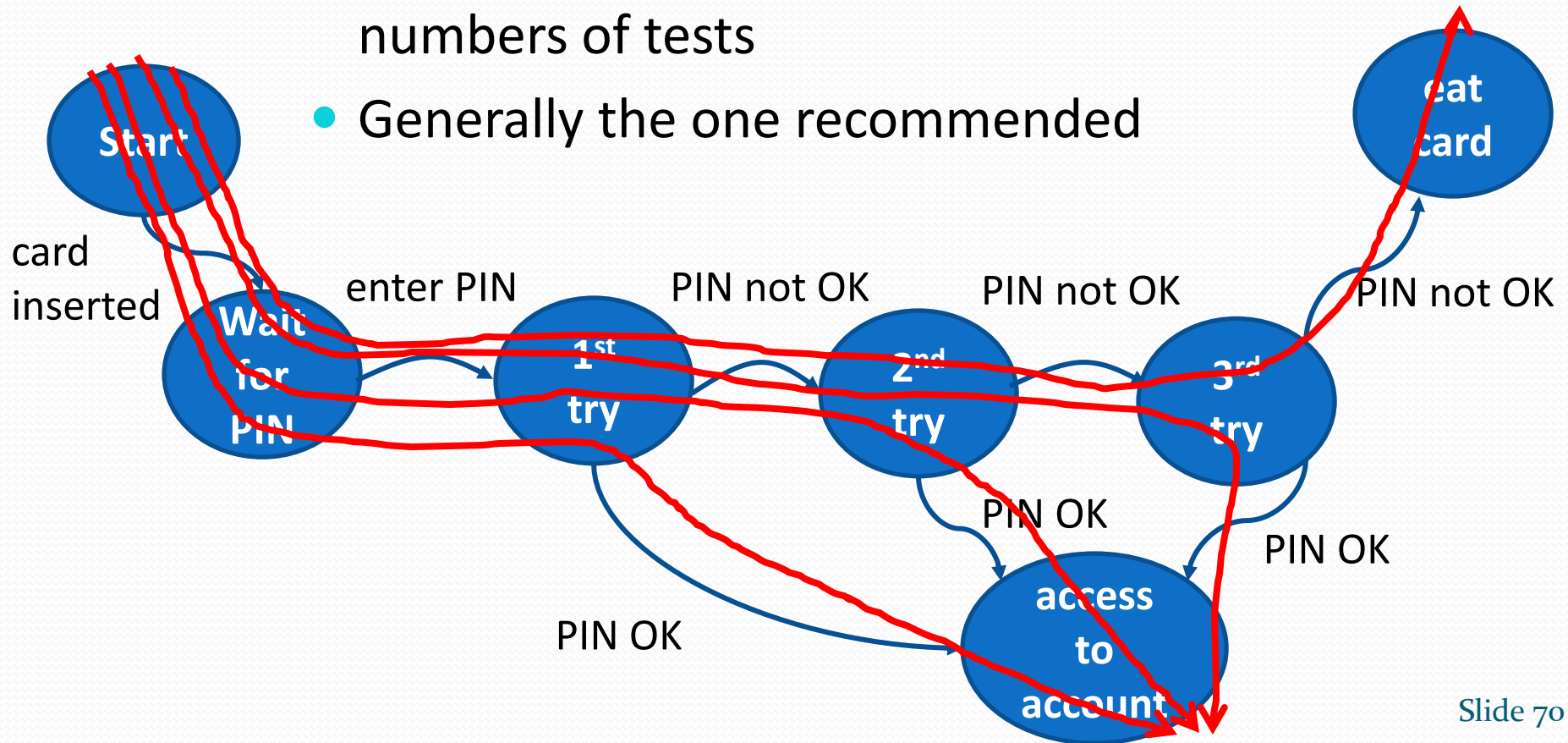
$A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow A$

...

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

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

# Solution - State table

# 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 <b>A: Actor</b> <b>S: System</b>	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