

University of Information Technology

# Kiểm tra tính đúng đắn & hiệu năng của chương trình bằng bộ test

Discussion

Group 3  
CS112.N22.KHCL

# Các kỹ thuật kiểm thử

## Kiểm thử tĩnh (Static Verification):

- Thực hiện kiểm chứng mà không cần thực thi chương trình
- Kiểm tra tính đúng đắn của các tài liệu có liên quan
- Giúp đạt được sự nhất quán trong hệ thống
- Giúp phát hiện sớm lỗi

## Kiểm thử động (Dynamic Testing)

- Thực hiện kiểm thử dựa trên việc thực thi chương trình
- Giúp phát hiện các lỗi khó phát hiện bằng kiểm thử tĩnh
- Có thể áp dụng cho tất cả các chương trình

# Phân loại kiểm thử động

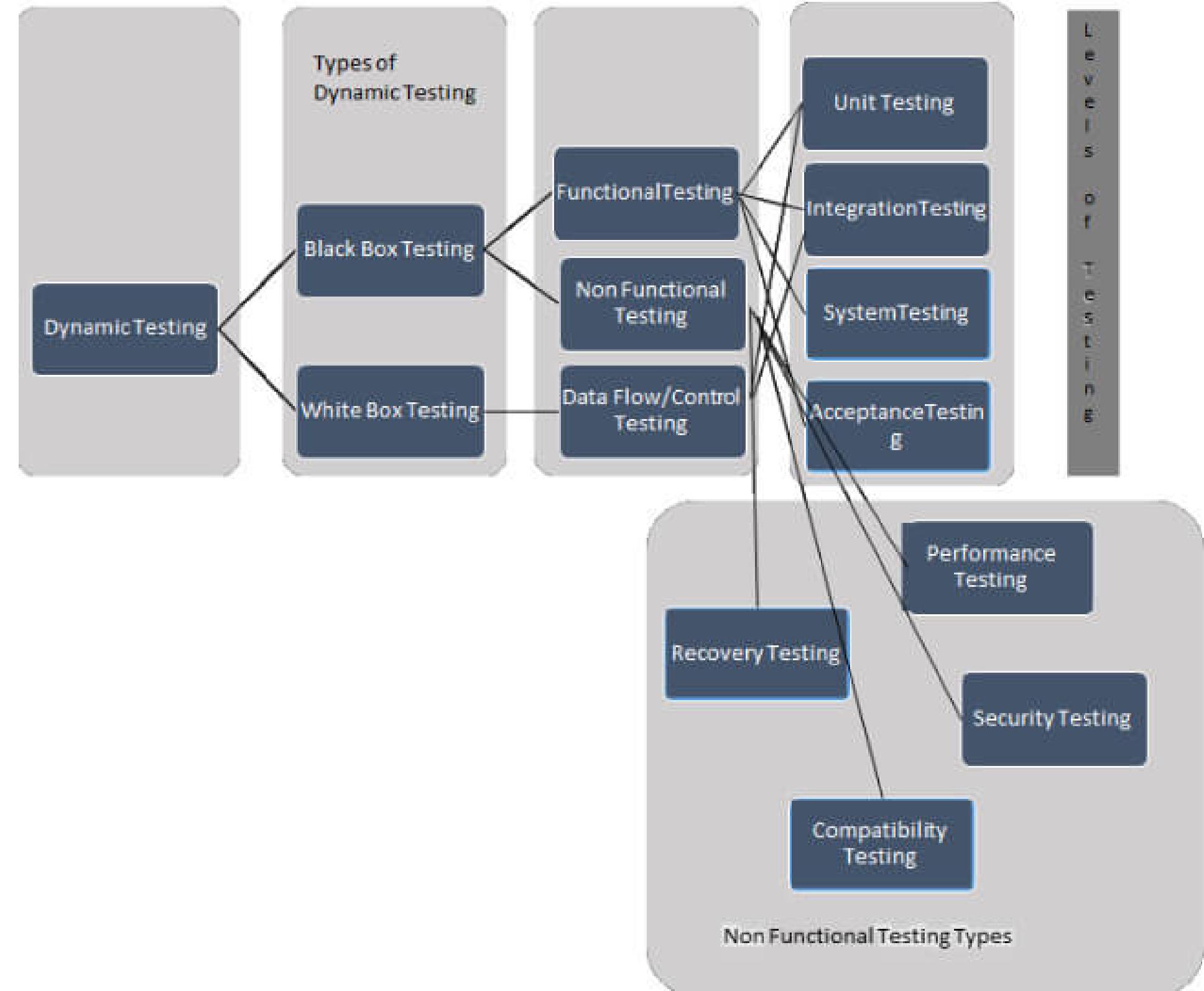
## Kiểm tra chức năng (Functional testing)

- Kiểm tra chức năng của chương trình dựa trên các đặc tả yêu cầu phần mềm
- Mỗi module sẽ được kiểm tra bằng các bộ input và output
- Bao gồm: Unit test, Integration test, Acceptance test

## Kiểm tra chất lượng phần mềm (Non-functional testing)

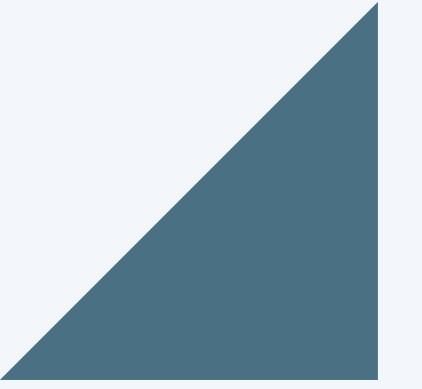
- Đảm bảo phần mềm đáp ứng các yêu cầu của người dùng cuối
- Giúp tăng cường hiệu năng, khả năng bảo trì, độ hiệu quả của sản phẩm
- Bao gồm performance testing (speed, load, stress test), security testing, usability testing

# Phân loại kiểm thử động



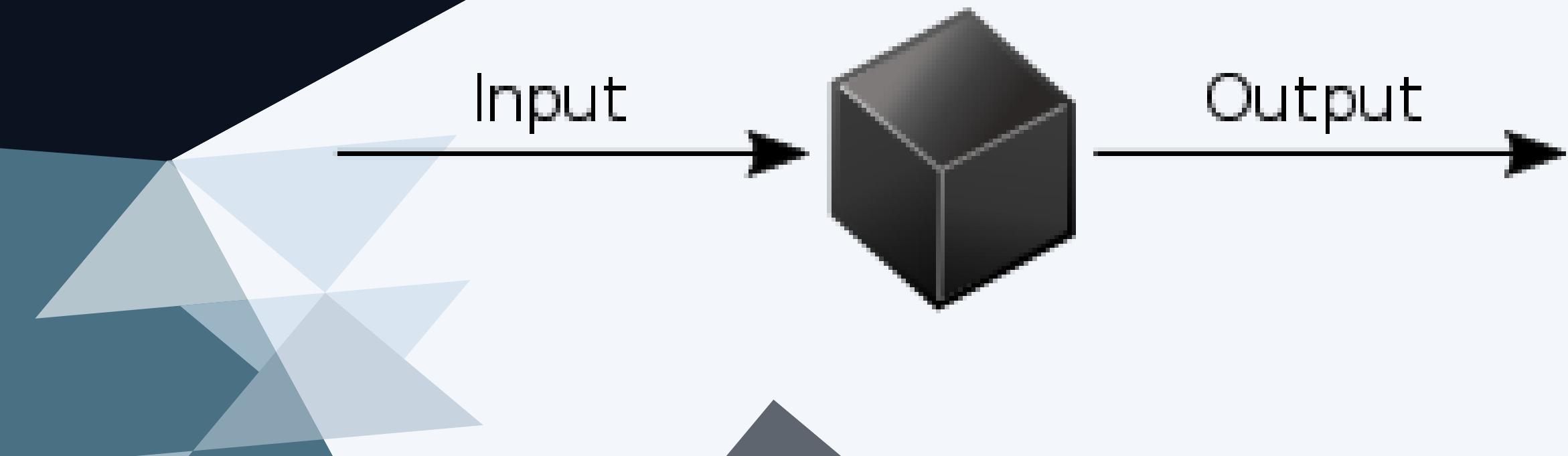
# Các phương pháp kiểm thử

- Kiểm thử hộp đen (Black Box Testing): kiểm thử dựa trên đặc tả yêu cầu phần mềm, xem xét phần mềm với các dữ liệu đầu vào và đầu ra mà không cần biết cấu trúc của phần mềm ra sao.
- Kiểm thử hộp trắng (White Box Testing): kiểm thử dựa trên cấu trúc phần mềm, tester có toàn bộ thông tin về cấu trúc của phần mềm, có quyền truy cập mã nguồn và các tài liệu kỹ thuật liên quan



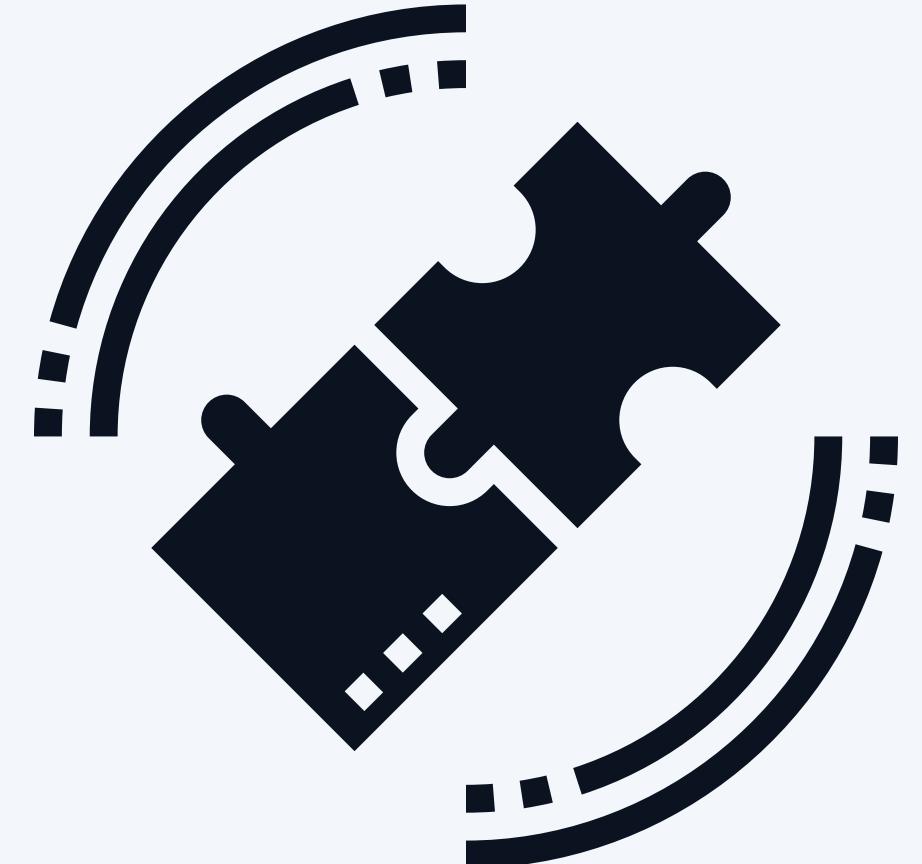
# Tổng quan về kiểm thử hộp đen

- Phương pháp kiểm thử hộp đen: coi hệ thống là một hộp đen, không thể thấy được cấu trúc logic bên trong. Người làm kiểm thử tập trung vào các yêu cầu chức năng của phần mềm dựa trên các dữ liệu lấy từ đặc tả yêu cầu phần mềm
- Có thể áp dụng gần như cho tất cả các cấp độ kiểm thử



# Tổng quan về kiểm thử hộp đen

- Kiểm thử hộp đen nhằm tìm ra các loại sai:
  - Chức năng thiếu hoặc không đúng đắn.
  - Sai về giao diện của chương trình
  - Sai trong cấu trúc dữ liệu hoặc trong truy cập dữ liệu bên ngoài.
  - Hành vi hoặc hiệu suất lỗi.



# **Ưu nhược điểm của kiểm thử hộp đen**

## **Ưu điểm**

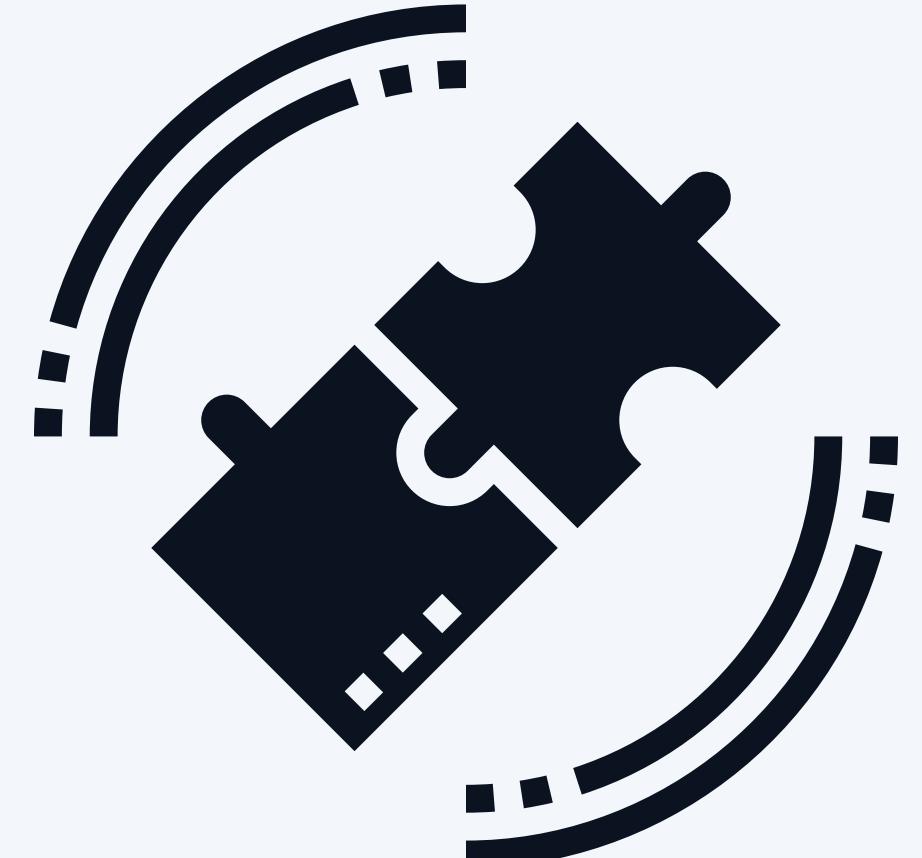
- Tester không cần phải nắm rõ cấu trúc chương trình
- Phát hiện được nhiều lỗi mà các lập trình viên khó tìm ra do phần mềm được nhìn với cái nhìn khác quan
- Thiết kế kịch bản kiểm thử nhanh
- Hệ thống thật sự với toàn bộ yêu cầu của nó được kiểm thử chính xác.

## **Nhược điểm**

- Đòi hỏi số lượng mẫu lớn
- Khả năng để bản thân kỹ sư lạc lối trong khi kiểm thử là khá cao.
- Chỉ có thể kiểm tra một phần nhỏ các trường hợp đầu vào

# Các kỹ thuật kiểm thử hộp đen

- Kỹ thuật phân lớp tương đương (Equivalence Class Testing)
- Kỹ thuật dựa trên giá trị biên (Boundary Value Testing)
- Kỹ thuật dựa trên bảng quyết định (Decision Table-Based Testing)
- Kỹ thuật dựa trên đồ thị nguyên nhân - kết quả (causes-effects)
- ...



# Kỹ thuật phân lớp tương đương (Equivalence Class Testing)

- Ý tưởng: Chia miền vào chương trình thành các lớp dữ liệu. Xác định đầu vào hợp lệ và không hợp lệ để lập các ca kiểm thử theo các lớp đó
- Mỗi lớp dùng để kiểm thử một chức năng, gọi là lớp tương đương.
- Thay vì kiểm tra tất cả các giá trị đầu vào, có thể lựa chọn từ đầu vào cho riêng từng lớp



# Kỹ thuật dựa trên giá trị biên (Boundary Value Testing)

- Phân tích giá trị biên - Boundary Value Analysis
- Thường được áp dụng đối với các đối số của một phương thức
- Tập trung vào việc kiểm thử các giá trị biên của miền giá trị inputs để thiết kế test case do “lỗi thường tiềm ẩn lại các ngõ ngách và tập hợp tại biên” ( Beizer )
- BVA hiệu quả nhất trong trường hợp “các đối số đầu vào (input variables) độc lập với nhau và mỗi đối số đều có một miền giá trị hữu hạn”



# Kỹ thuật dựa trên đồ thị nguyên nhân – kết quả (causes-effects)

- Là kỹ thuật thiết kế test case dựa trên đồ thị
- Tập trung vào việc xác định các mối kết hợp giữa các conditions và kết quả mà các mối kết hợp này mang lại



# Kỹ thuật dựa trên bảng quyết định

- Là một loại kiểm thử phần mềm kiểm tra cách một chương trình phản ứng với các sự kết hợp khác nhau của đầu vào
- Đây là một phương pháp trong đó các sự kết hợp khác nhau của đầu vào và hành vi hệ thống đi kèm (Đầu ra) được lập bảng



**The condition is simple** – The user will be routed to the homepage if they give the right username and password. An error warning will appear if any of the inputs are incorrect.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

### Legend

- T - Make sure your login and password are correct.
- F - Incorrect login or password
- E - An error message appears.
- H - The home screen appears.

Ví dụ minh họa một bảng trong kỹ thuật bảng quyết định

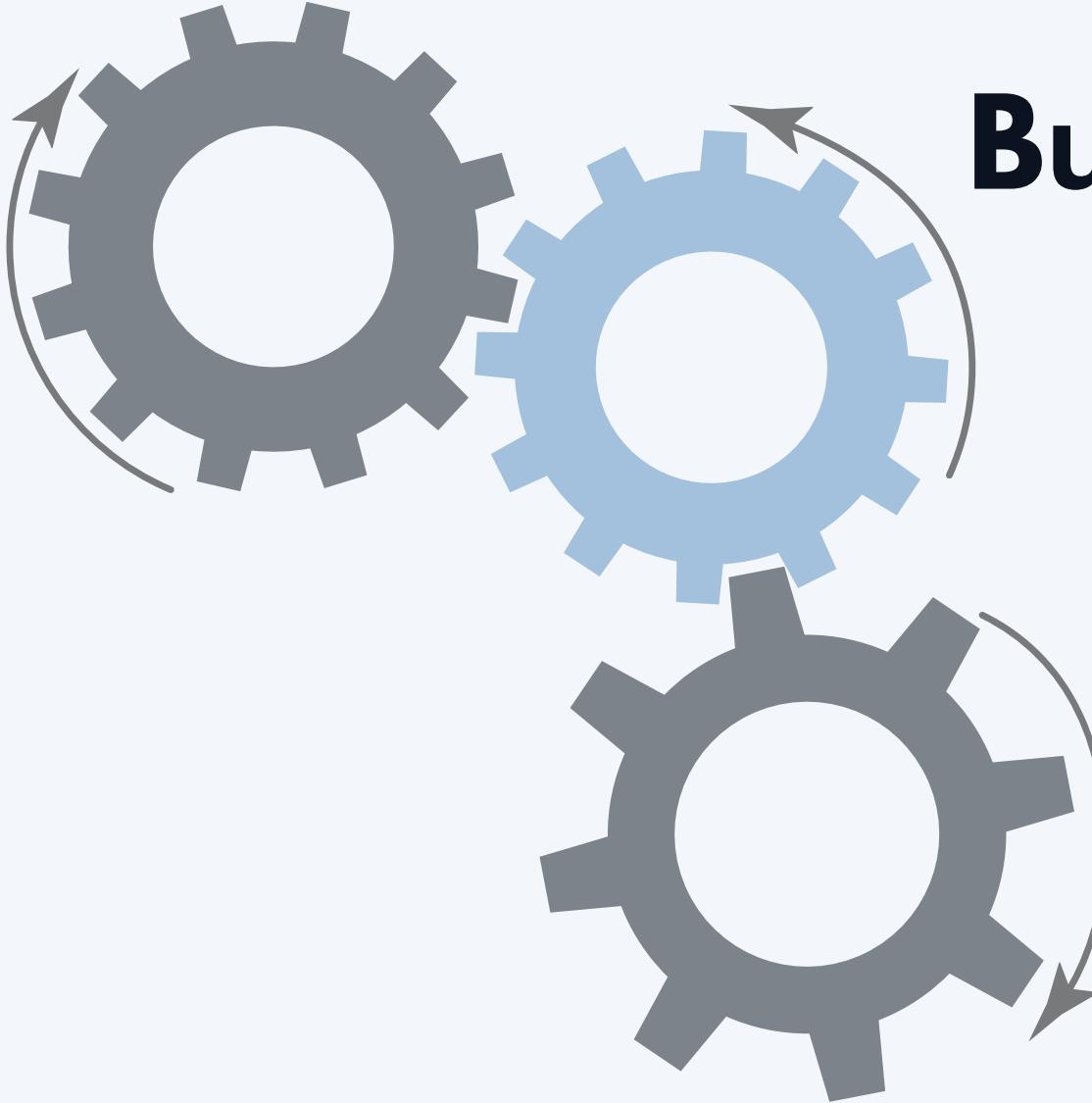
# Kỹ thuật dựa trên bảng quyết định

Liệt kê các nguyên nhân (cause) – kết quả (effect) trong 1 ma trận. Mỗi cột trong ma trận đại diện cho 1 phép kết hợp giữa các cause trong việc tạo ra 1 effect

		Combinations							
Causes	Values	1	2	3	4	5	6	7	8
Cause 1	Y, N	Y	Y	Y	Y	N	N	N	N
Cause 2	Y, N	Y	Y	N	N	Y	Y	N	N
Cause 3	Y, N	Y	N	Y	N	Y	N	Y	N
Effects									
Effect 1		X		X			X		X
Effect 2			X			X		X	X

Cause = Condition  
Effect = Actions = Expected Results

# Các bước tạo bảng quyết định

- 
- Bước 1** Liệt kê tất cả các nguyên nhân (causes) trong bảng quyết định
  - Bước 2** Tính tổng số lượng kết hợp giữa các cause
  - Bước 3** Điền vào các cột với tất cả các kết hợp có thể có
  - Bước 4** Rút bớt số lượng các phép kết hợp dư thừa
  - Bước 5** Kiểm tra các phép kết hợp có bao phủ hết mọi trường hợp hay không
  - Bước 6** Bổ sung kết quả (effects) vào bảng quyết định

# Bước 1: Liệt kê tất cả các nguyên nhân

Ví dụ: xét bài toán kiểm tra loại của 1 tam giác dựa vào chiều dài 3 cạnh a, b, c.

Causes	Values	Combinations							
		1	2	3	4	5	6	7	8
Cause 1	Y, N	Y	Y	Y	Y	N	N	N	N
Cause 2	Y, N	Y	Y	N	N	Y	Y	N	N
Cause 3	Y, N	Y	N	Y	N	Y	N	Y	N
Effects									
Effect 1		X		X				X	
Effect 2			X			X		X	

c1:  $a < b+c?$   
c2:  $b < a+c?$   
c3:  $c < a+b?$   
c4:  $a = b?$   
c5:  $a = c?$   
c6:  $b = c?$

- Điền vào các giá trị trong từng causes
- Gom nhóm các causes có liên quan với nhau
- Sắp xếp các cause theo thứ tự giảm dần theo độ ưu tiên

## Bước 2:

# Tính tổng số kết hợp giữa các causes

- Tổng số phép kết hợp = (số lượng values của cause 1) \*... \* (số lượng values của cause n)

c1: a < b + c?
c2: b < a + c?
c3: c < a + b?
c4: a = b?
c5: a = c?
c6: b = c?

- Mỗi cause có 2 giá trị true, false  
->Tổng số phép kết hợp =  $2^6 = 64$

# Bước 3: Điền giá trị các cột trong bảng

Causes	Values	Combinations							
		1	2	3	4	5	6	7	8
Cause 1	Y, N	Y	Y	Y	Y	N	N	N	N
Cause 2	Y, N	Y	Y	N	N	Y	Y	N	N
Cause 3	Y, N	Y	N	Y	N	Y	N	Y	N
Effects									
Effect 1		X		co	X				X
Effect 2			X			X		X	

- Thuật toán:
  - Xác định số lần lặp lại (RF) trong từng giá trị của cause bằng cách lấy tổng số phép kết hợp còn lại chia cho số values mà cause có thể nhận

# Bước 3: Điền giá trị các cột trong bảng

Causes	Values	Combinations							
		1	2	3	4	5	6	7	8
Cause 1	Y, N	Y	Y	Y	Y	N	N	N	N
Cause 2	Y, N	Y	Y	N	N	Y	Y	N	N
Cause 3	Y, N	Y	N	Y	N	Y	N	Y	N
Effects									
Effect 1		X		X		X		X	
Effect 2			X			X		X	

- Thuật toán:
  - Điền dữ liệu cho dòng thứ i: điền RF lần giá trị đầu tiên của cause i, tiếp theo RF lần giá trị tiếp theo của cause i... cho đến khi dòng đầy

# Bước 3: Điền giá trị các cột trong bảng

Causes	Values	Combinations							
		1	2	3	4	5	6	7	8
Cause 1	Y, N	Y	Y	Y	Y	N	N	N	N
Cause 2	Y, N	Y	Y	N	N	Y	Y	N	N
Cause 3	Y, N	Y	N	Y	N	Y	N	Y	N
Effects									
Effect 1		X		X		X		X	
Effect 2			X			X		X	

- Thuật toán:
  - Chuyển sang dòng kế tiếp, quay lại bước 1 và tiếp tục thực hiện

# Bước 3: Điền giá trị các cột trong bảng

- Ví dụ:

c1: a<b+c ?	32		32	
	F		T	
c2: b<a+c ?	16	16	16	16
	F	T	F	T
c3: c<a+b ?	8	8	8	8
	F	T	F	T
c4: a=b ?	....			

$$\text{RF} = 64 / 2 = 32$$

$$\text{RF} = 32 / 2 = 16$$

$$\text{RF} = 16 / 2 = 8$$

# Bước 4: Giảm số phép kết hợp

- Duyệt qua tất cả các ô trong từng cột, ô nào mà kết quả của nó không ảnh hưởng đến effect thì đặt giá trị trên ô này là “-” (don’t care entry)
  - Ghép các cột với nội dung giống nhau thành 1 cột

# Bước 5: Kiểm tra độ bao phủ các phép kết hợp

- Tính rule-count trên từng cột (số lượng phép kết hợp) mà cột này có thể thực hiện
  - Với các dòng có giá trị là '-' thì luỹ thừa 2
  - Nếu tổng của các rule-count bằng với tổng số kết hợp giữa các cause trong bước 2 thì bảng quyết định là đầy đủ

# Bước 6: Bổ sung kết quả (effect) vào trong bảng

- Duyệt qua từng cột và check vào kết quả (effect)
- Nhiều cột khác nhau có thể cho ra cùng 1 kết quả giống nhau

c1: $a < b+c?$	F	T	T	T	T	T	T	T	T	T	T
c2: $b < a+c?$	-	F	T	T	T	T	T	T	T	T	T
c3: $c < a+b?$	-	-	F	T	T	T	T	T	T	T	T
c4: $a = b?$	-	-	-	T	T	T	T	F	F	F	F
c5: $a = c?$	-	-	-	T	T	F	F	T	T	F	F
c6: $b = c?$	-	-	-	T	F	T	F	T	F	T	F
Rule Count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a Triangle	X	X	X	X	X	X	X	X	X	X	X
a2: Scalene											X
a3: Isosceles							X	X	X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

# Ví dụ 1

- Công ty Honda trao học bổng cho những bạn sinh viên thỏa mãn ít nhất 1 trong 2 điều kiện sau: Là sinh viên giỏi , là cán bộ lớp.
- Nếu thỏa mãn cả 2 điều kiện sẽ được học bổng 600\$, nếu thỏa mãn là sinh viên giỏi được học bổng 400\$, nếu là cán bộ lớp được học bổng là 300\$.
- Lập bảng hỗ trợ quyết định để thiết kế các ca kiểm thử

# VD1: Bảng hỗ trợ quyết định

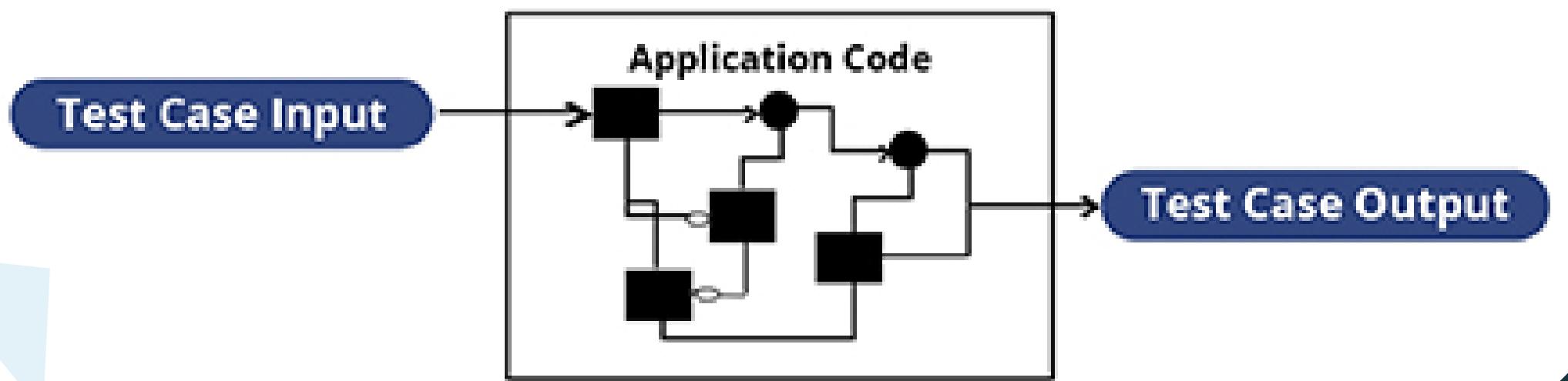
		Luật 1	Luật 2	Luật 3	Luật 4
Điều kiện	Là cán bộ lớp	Y	Y	N	N
	Là học sinh giỏi	Y	N	Y	N
Hành động	Học bổng	600\$	300\$	400\$	0\$

Các ca kiểm thử

Ca	Đầu vào	Đầu ra mong đợi
1	Là cán bộ lớp, là sv giỏi	600\$
2	Là cán bộ lớp, ko fai là sv giỏi	300\$
3	Ko phải cán bộ lớp, là sv giỏi	400\$
4	Ko phải cán bộ lớp, ko phải sv giỏi	0\$

# Tổng quan kiểm thử hộp trắng

- Thiết kế test case dựa vào cấu trúc nội tại bên trong của đối tượng cần kiểm thử
- Các kiến thức về cấu trúc bên trong của hệ thống được sử dụng để thiết kế các test case



# Tổng quan kiểm thử hộp trắng

- Các tên gọi khác: kiểm thử cấu trúc (structural testing), kiểm thử hộp kính (glass box), kiểm thử rõ ràng (clear box testing).
- Đối tượng chính của kiểm thử hộp trắng là tập trung vào cấu trúc bên trong chương trình và tìm ra tất cả những lỗi bên trong chương trình.



# Tổng quan kiểm thử hộp trắng

- Việc kiểm tra tập trung chủ yếu vào:
  - Cấu trúc chương trình: Những câu lệnh và các nhánh, các loại đường dẫn chương trình.
  - Logic bên trong chương trình và cấu trúc dữ liệu.
  - Những hành động và trạng thái bên trong chương trình.



# Ưu nhược điểm của kiểm thử hộp trắng

## Ưu điểm

- Các kiến thức về cấu trúc chương trình giúp ích nhiều trong quá trình kiểm thử
- Dễ dàng tự động hóa
- Dễ dàng bắt kịp các thay đổi trong mã nguồn
- Giúp kiểm tra chương trình bao quát nhiều trường hợp hơn
- Dễ dàng phát hiện lỗi

## Nhược điểm

- Tốn nhiều công sức hơn trong việc cập nhật test case khi chương trình thay đổi
- Làm cho quá trình kiểm thử phức tạp, đặt ra yêu cầu cao hơn cho tester
- Việc bỏ sót chức năng, sai chức năng sẽ không bị phát hiện

# Các kỹ thuật kiểm thử hộp trắng

- Basis Path Testing
- Control-flow/Coverage Testing
- Data-flow Testing



# Control-flow/Coverage Testing



## Khái niệm

Là kỹ thuật thiết kế test case đảm bảo “cover” được tất cả các câu lệnh, biểu thức điều kiện trong code module cần test



## Các tiêu chí đánh giá độ bao phủ

- Method Coverage
- Statement Coverage
- Decision/Branch Coverage
- Condition Coverage

# Method Coverage

- Tỷ lệ phần trăm các phương thức trong chương trình được gọi thực hiện bởi các test case
- Test case cần phải đạt được 100% method coverage

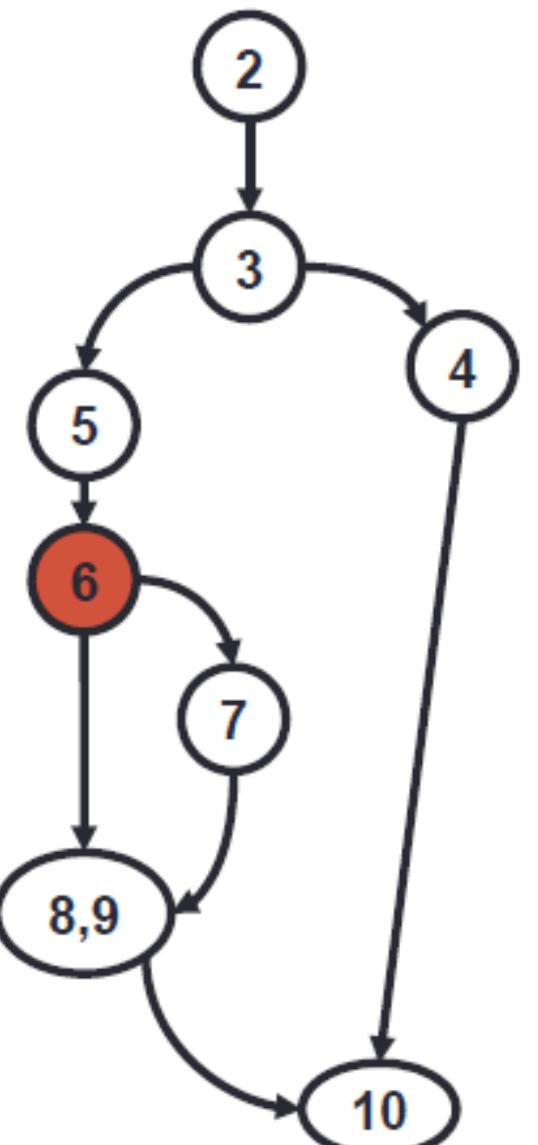
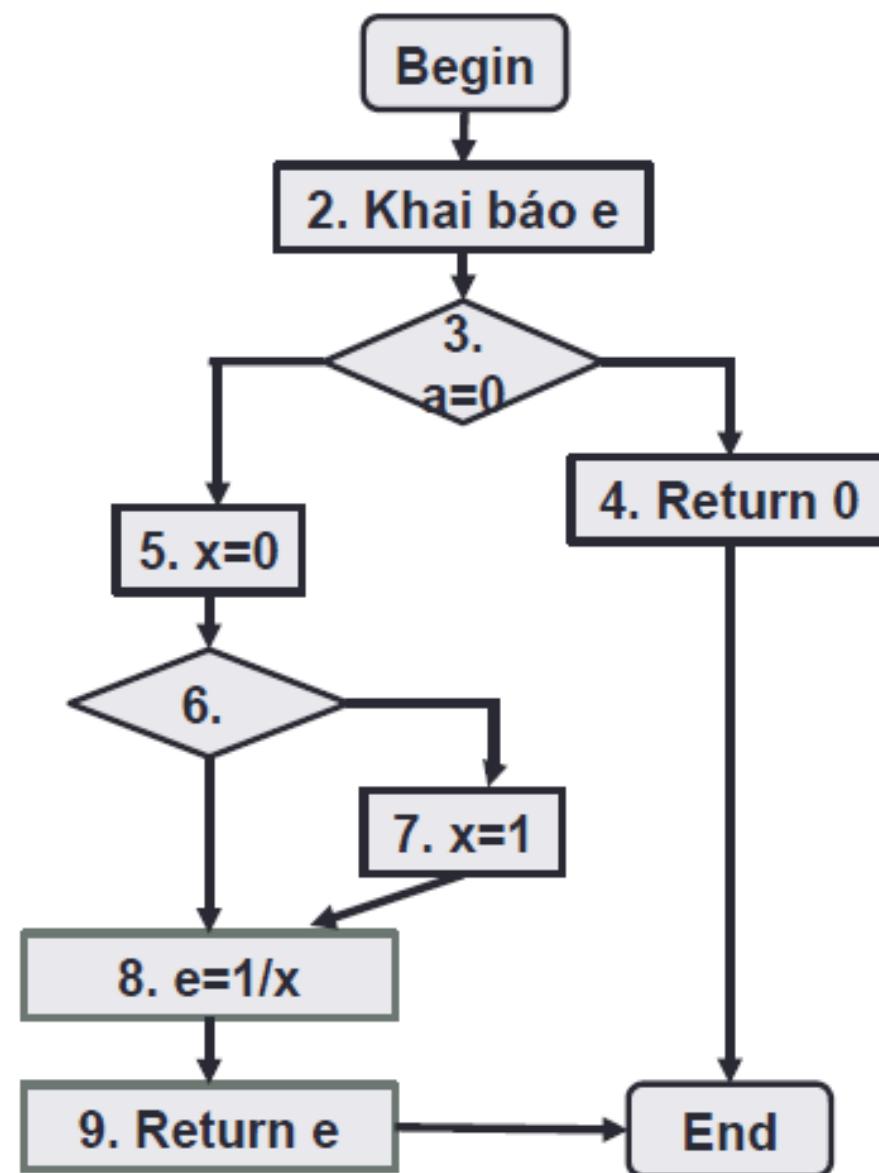
# Ví dụ - Method Coverage



```
1 def foo(a, b, c, d):  
2     if a == 0:  
3         return 0  
4     x = 0  
5     if a == b or (c == d and a == 3):  
6         x = 1  
7     e = 1 / x  
8     return e  
9
```

- Test case 1: foo (0,0,0,0)
- 100% method coverage

# Vd: Đồ thị dòng



# Statement Coverage



```
1 def foo(a, b, c, d):  
2     if a == 0:  
3         return 0  
4     x = 0  
5     if a == b or (c == d and a == 3):  
6         x = 1  
7     e = 1 / x  
8     return e  
9
```

- Tỷ lệ phần trăm các câu lệnh trong chương trình được gọi thực hiện bởi các test case

- Test case 1 thực hiện các lệnh từ 1 đến 3 trong 8 câu lệnh đạt 37.5% Statement Coverage

- Để đạt 100% Statement Coverage -> Test case 2: foo (1,1,1,1)

# Decision/Branch Coverage

- Tỷ lệ phần trăm các **biểu thức điều kiện** trong chương trình được ước lượng giá trị trả về (true, false) khi thực thi các test case
- Một biểu thức điều kiện (**cho dù là single hay complex**) phải được kiểm tra trong cả hai trường hợp giá trị của biểu thức là true hay false
- Đối với các hệ thống lớn, thường chỉ đạt từ 75% -> 85% độ bao phủ

# Decision/Branch Coverage

- Đạt 75% coverage

Test case 3: foo (1,2,1,2) 100% coverage

Line	Predicate	True	False
2	$a == 0$	Test case 1 foo(0, 0, 0, 0)	Test case 2 foo(1, 1, 1, 1)
4	$a == b \text{ or } (c == d \text{ and } a == 3)$	Test case 2 foo(1, 1, 1, 1)	

# Condition Coverage

- Tỷ lệ phần trăm các biểu thức điều kiện đơn trong biểu thức điều kiện phức của chương trình được ước lượng giá trị trả về (true, false) khi thực thi các test case

# Condition Coverage

- Đạt 75% coverage

Test case 4: foo (3,2,1, 1) 100% coverage

Predicate	True	False
$a == 0$	<b>Test case 1</b> foo(0, 0, 0, 0)	<b>Test case 2</b> foo(1, 1, 1, 1)
$a == b$	<b>Test case 2</b> foo(1, 1, 1, 1)	<b>Test case 3</b> foo(1, 2, 1, 2)
$c == d$	<b>Test case 2</b> foo(1, 1, 1, 1)	
$a == 3$	<b>Test case 2</b> foo(1, 1, 1, 1)	

# Quiz

Viết số lượng test case ít nhất nhưng đạt được 100% condition coverage



```
1 def is_prime(number):
2     """Returns True if the given number is prime, False otherwise."""
3     if number <= 1:
4         return False
5     elif number == 2:
6         return True
7     elif number % 2 == 0:
8         return False
9     else:
10        # check odd numbers from 3 up to the square root of the number
11        for i in range(3, int(number**0.5) + 1, 2):
12            if number % i == 0:
13                return False
14        return True
```

# Tổng quan về UnitTest

- Unit test là các đoạn mã có cấu trúc giống như các đối tượng được xây dựng để kiểm tra từng bộ phận trong hệ thống.
- Mỗi unit test sẽ gửi đi một thông điệp và kiểm tra câu trả lời nhận được đúng hay không, bao gồm:
  - Các kết quả trả về mong muốn
  - Các lỗi ngoại lệ mong muốn

# Vòng đời của UnitTest

UnitTest có 3 trạng thái cơ bản:

- Fail
- Ignore (Skip)
- Pass

# Ứng dụng của UnitTest

- Kiểm tra mọi đơn vị nhỏ nhất là các thuộc tính, sự kiện, thủ tục và hàm.
- Kiểm tra các trạng thái và ràng buộc của đối tượng ở các mức sâu hơn mà thông thường chúng ta không thể truy cập được.
- Kiểm tra các quy trình (process) và mở rộng hơn là các khung làm việc(workflow - tập hợp của nhiều quy trình).

# Lợi ích của UnitTest

- Tạo ra môi trường lý tưởng để kiểm tra bất kỳ đoạn mã nào
- Phát hiện các thuật toán thực thi không hiệu quả
- Phát hiện các vấn đề về thiết kế, xử lý hệ thống
- Phát hiện các lỗi nghiêm trọng khó xảy ra
- Tạo hàng rào an toàn cho các khối mã
- Là môi trường lý tưởng để tiếp cận các thư viện API bên ngoài một cách tốt nhất
- Giải phóng các chuyên viên Q/A khỏi các công việc kiểm tra phức tạp
- Tăng sự tự tin khi hoàn thành một công việc

# Thiết kế test case

Hai tài nguyên thiết yếu sau sẽ cần thiết cho việc thiết kế các testcase:

- **Đặc tả chức năng module :** nêu rõ các thông số đầu vào, đầu ra và các chức năng cụ thể chi tiết của module.
- **Mã nguồn của module.**

Tính chất các testcase là dựa chủ yếu vào kỹ thuật kiểm thử hợp trắcng

- Khi kiểm thử phần tử ngày càng lớn hơn thì kỹ thuật kiểm thử hộp trắcng ít khả thi hơn
- Việc kiểm thử sau đó thường hướng đến việc tìm ra các kiểu lỗi (lỗi phân tích, lỗi nắm bắt yêu cầu phần mềm)

# Unit test với Pytest

Pytest là một công cụ phục vụ kiểm thử chức năng cho chương trình, với các ưu điểm như nhỏ gọn, dễ đọc, có thể kiểm thử những chương trình phức tạp

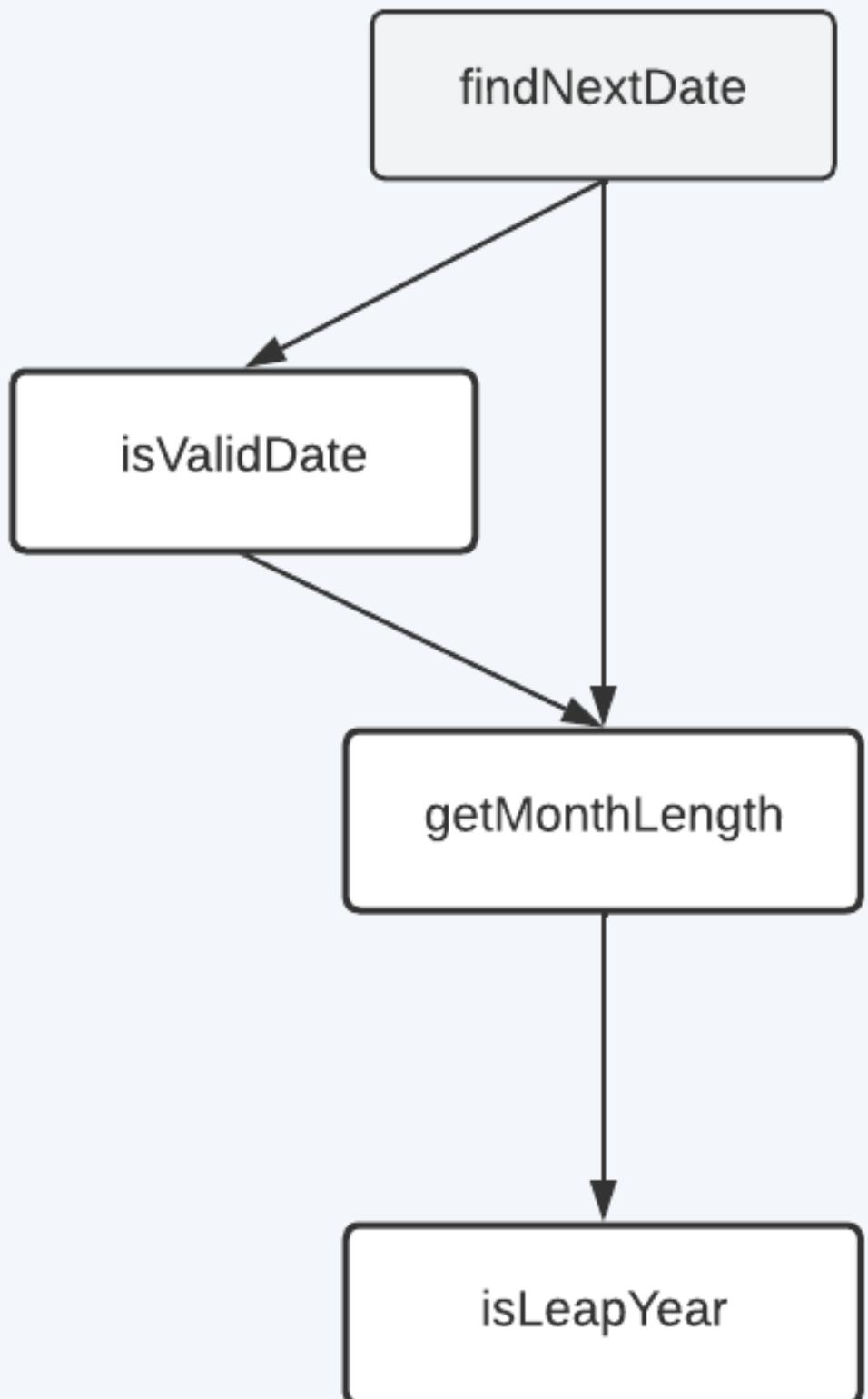
Trang chủ: [Pytest homepage](#)

# Ví dụ: Bài toán tìm ngày kế tiếp

Link ví dụ: [Find next date](#)

Yêu cầu bài toán:

Tìm ngày tiếp theo của ngày được nhập vào,  
nếu ngày được nhập vào không hợp lệ thì sẽ  
báo lỗi



# Kiểm thử hàm getMonthLength



```
1 def getMonthLength(month: int, year: int) -> int:
2     if month in (1, 3, 5, 7, 8, 10, 12):
3         return 31
4     elif month == 2:
5         if isLeapYear(year):
6             return 29
7         else:
8             return 28
9     else:
10        return 30
```

# Kiểm thử hàm getMonthLength

1. Mock isLeapYear  
Giá trị isLeapYear sẽ được  
truyền trực tiếp vào từ  
testcase



```
1 @pytest.fixture
2 def mockIsLeapYear(mocker, isLeapYear):
3     mocker.patch("src.findNextDate.isLeapYear", return_value=isLeapYear)
```



```
1 @pytest.mark.parametrize(
2     "month, year, isLeapYear, monthLength",
3     [(2, 2023, False, 28), (2, 2024, True, 29),
4      (3, 2023, False, 31), (4, 2023, False, 30)],
5 )
6 @pytest.mark.usefixtures("mockIsLeapYear")
7 def test_get_month_length(month, year, isLeapYear, monthLength):
8     assert getMonthLength(month, year) == monthLength
```

## 2. Viết test case cho getMonthLength

- Truyền các test case qua `@pytest.mark.parametrize`
- Sử dụng `mockIsLeapYear` để thay thế cho hàm `isLeapYear`

# Kiểm thử hàm getMonthLength

## 3. Chạy test bằng lệnh

```
python -m pytest --cov=src/ tests/ --cov-branch --cov-report html
```

```
tests\test_find_next_day.py sssss...ssssssss [100%]

----- coverage: platform win32, python 3.11.1-final-0 -----
Name           Stmts  Miss Branch BrPart  Cover
src\findNextDate.py      33     22      22      0    31%
TOTAL          33     22      22      0    31%

===== 4 passed, 13 skipped in 0.13s =====
```

# Quiz

```
def isValidDate(day: int, month: int, year: int) → bool:  
    if month < 1 or month > 12:  
        return False  
    if day < 0 or day > getMonthLength(month, year):  
        return False  
    return True
```

Bộ test trên đạt tỉ lệ condition coverage là bao nhiêu phần trăm

```
@pytest.mark.parametrize(  
    "day, month, year, monthLength, isValid",  
    [  
        (1, 2, 2023, 28, True),  
        (30, 2, 2024, 29, False),  
        (20, 0, 2023, 0, False)  
    ],  
)  
  
@ pytest . mark . usefixtures ("mockGetMonthLength")  
def test_is_valid_date (day, month, year, monthLength, isValid):  
    assert isValidDate (day, month, year) == isValid
```

# Chiến lược viết mã hiệu quả

- Phân tích các tình huống có thể xảy ra với mã
- Mọi UT phải bắt đầu với trạng thái fail và chuyển sang pass sau một số thay đổi hợp lý với mã
- Mỗi khi viết một đoạn mã quan trọng, hãy viết UT tương ứng cho các trường hợp bạn nghĩ ra
- Nhập số lượng đủ lớn các testcase
- Sớm nhận biết các đoạn mã không ổn định, có nguy cơ lỗi

# Chiến lược viết mã hiệu quả

- Tạo UT cho mỗi đối tượng nghiệp vụ và đối tượng truy cập dữ liệu
- Thực thi lại tất cả UnitTest mỗi khi có sự thay đổi quan trọng
- Sử dụng nhiều phương thức kiểm tra khác nhau
- UnitTest đòi hỏi sự nỗ lực, kinh nghiệm và sáng tạo như viết phần mềm

University of Information Technology

**Thank You**  
For Your Attention

Group 3  
CS112.N22.KHCL