

CS 361 – Software Engineering

An Introduction to Software Testing

Kamonphop Srisopha
Churee Techawut



Faculty of Science, Chiang Mai University
คณะวิทยาศาสตร์ มหาวิทยาลัยเชียงใหม่

Agenda

- Why is Software Testing Important?
- Basic Terminology
- Why is Testing Hard?
- General Approach to Testing
- Levels of Testing

Why is Software Testing Important?

Windows 98 Keynote



Ariane 5 Rocket Flight V88 (1996)



Problem:

Rocket exploded **37 seconds** after launch, resulting in a loss of more than US\$370 million.

Cause:

Old software (Ariane 4) to a new hardware (Ariane 5), Can't handle faster velocity readings
(Conversion of 64-bit float to 16-bit integer)

```
int counter = MAX_INT;  
counter++;
```

Therac-25 (1985-1987)



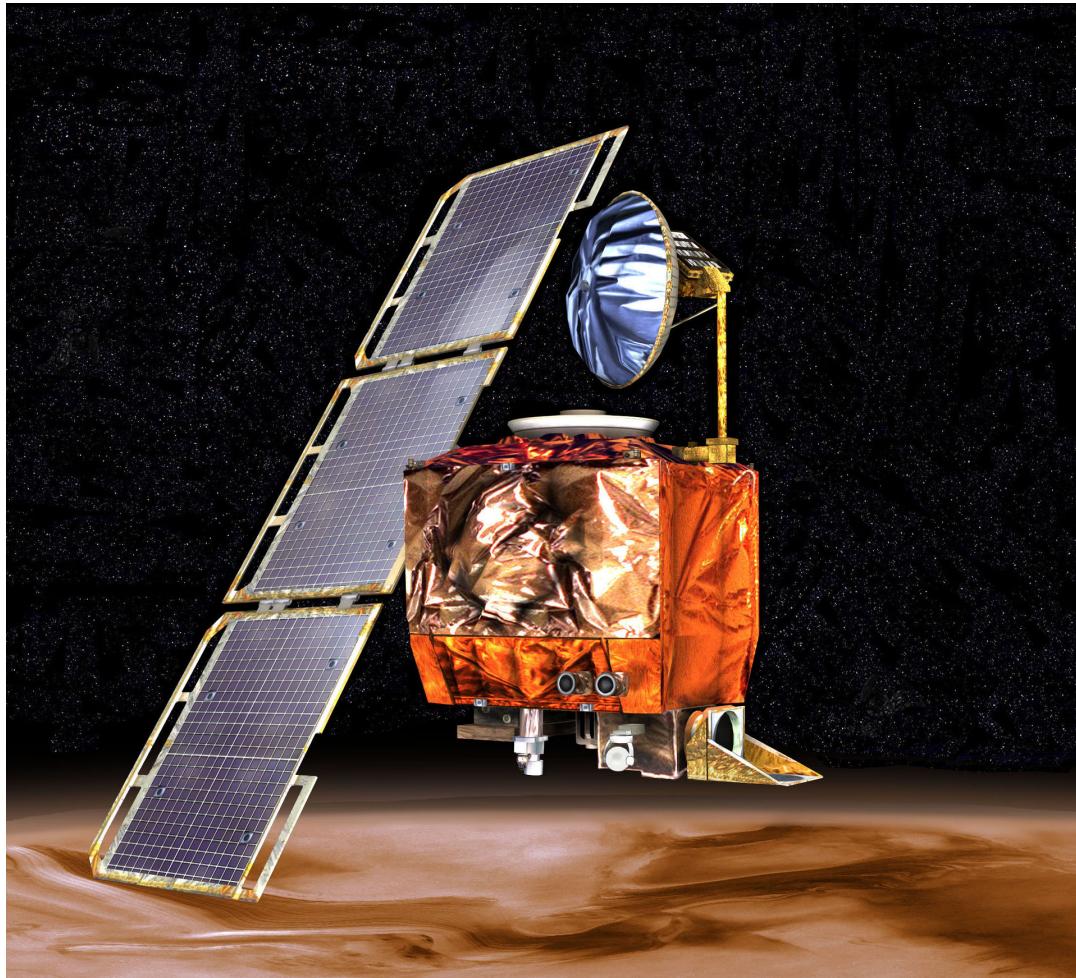
Two software faults cause the machine to deliver lethal doses of radiation to patients.

- Software fails to recognize the changes in mode when switching two types of therapy from x-ray and electron too quickly (needs ≥ 8 mins to successfully change mode).
- Software can report incorrect “turntable” position (0 correct position, other number = not correct → problem occurs when the position is 255 and $+1 = 0$ (like your car odometer) – arithmetic overflow)

Therac-25: Selecting Beam Type

PATIENT NAME: John	BEAM TYPE: E	ENERGY (KeV) :	10
TREATMENT MODE: FIX			
	ACTUAL	PRESCRIBED	
UNIT RATE/MINUTE	0.000000	0.000000	
MONITOR UNITS	200.000000	200.000000	
TIME (MIN)	0.270000	0.270000	
GANTRY ROTATION (DEG)	0.000000	0.000000	VERIFIED
COLLIMATOR ROTATION (DEG)	359.200000	359.200000	VERIFIED
COLLIMATOR X (CM)	14.200000	14.200000	VERIFIED
COLLIMATOR Y (CM)	27.200000	27.200000	VERIFIED
WEDGE NUMBER	1.000000	1.000000	VERIFIED
ACCESSORY NUMBER	0.000000	0.000000	VERIFIED
DATE: 2012-04-16	SYSTEM: BEAM READY	OP.MODE: TREAT	AUTO
TIME: 11:48:58	TREAT: TREAT PAUSE	X-RAY	173777
OPR ID: 033-tfs3p	REASON: OPERATOR	COMMAND: █	

Mars Climate Orbiter (1999)



NASA lost a \$125 million Mars orbiter because a Lockheed Martin engineering team used **English Standard units of measurements** while NASA used the **metric system** for a key spacecraft operation

The wrong navigation information was sent to the Orbiter. The spacecraft went in lower altitude than expected. – Burned and broken into pieces

The calculation is done with **Pound-force** seconds, not **Newton** seconds

Why is Software Testing Important?

The image is a collage of news headlines from different sources, each highlighting a significant failure due to software bugs or testing issues:

- British Airways flight delayed affecting 100,000 passengers** (British Airways, 13 October)
- Tesla recalls 362,000 vehicles over crash** (Reuters)
- Up to 300,000 heart patients may have been given wrong drugs or advice due to major NHS IT blunder** (BGR)
- James Webb Space Telescope just hit a software glitch** (BGR)
- PayPal \$92 quadrillion in error** (BGR, 17 July 2013)

Each headline includes a snippet of the original news article, such as the date and author's name.

The Economic Impact of Software Failures

- Financial (including loss of sales)
- Loss of life
- Loss of equipment
- Inconvenience
- Confusion and Chaos
- Loss of trust



(2022) The cost of poor software quality in the US has grown to at least **\$2.41 trillion**. Operational problems contributed the largest share, with **\$1.56 trillion** in losses. The cost of cybersecurity failures has grown significantly

Basic Terminology

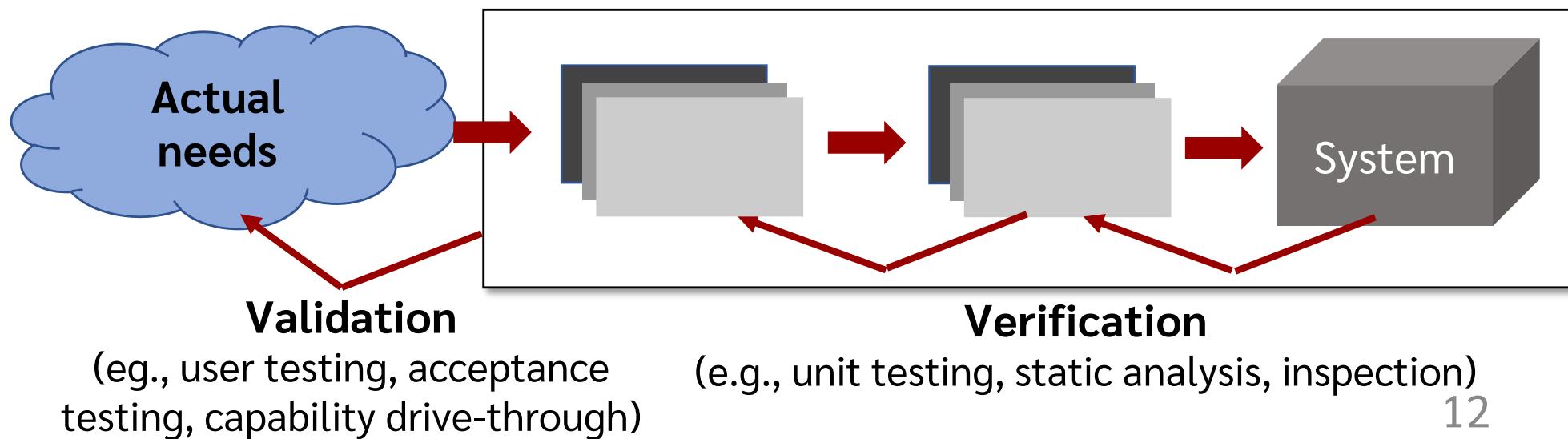
Basic Terminology

Verification: “Are we building the product right?”

การที่ต้องยืนยันว่าระบบถูกสร้างตาม specification ที่ได้ระบุไว้
ตลอดขั้นตอนการพัฒนาระบบทรีอิ่ม

Validation: “Are we building the right product?”

การที่ต้องยืนยันว่าระบบถูกสร้างขึ้นมาเพื่อตอบโจทย์ความต้องการ
ของผู้ใช้ หรือผู้ที่มีส่วนได้ส่วนเสียหรือไม่



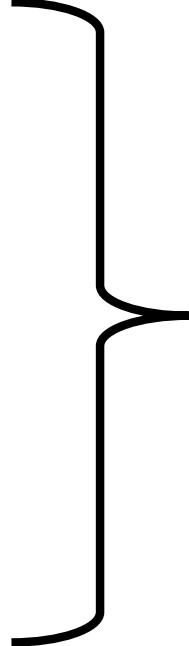
Validation

- เราจะรู้ได้อย่างไรว่าเรา built the right thing?
- ใครเป็นคนให้คำตอบว่า yes หรือ no?
- ให้นิยาม **Business Model** ที่ได้ทำ
 - ระบบที่พัฒนาให้คุณค่าแก่ Beneficiaries ที่ได้ระบุไว้หรือเปล่า
 - ระบบที่พัฒนาสามารถให้คุณค่าทางคุณภาพที่ระบุไว้ได้มั้ย (Value Proposition)
 - ผู้มีส่วนได้ส่วนเสียมีใครบ้าง (Stakeholders) และคนคิดเห็นอย่างไร

กระบวนการ **Validation** เกิดขึ้นได้เสมอ

- Validate กับผู้ว่าจ้าง เช่น การให้ผู้ว่าจ้างได้ลองใช้ increment หลังจบ Sprint (ในกิจกรรม Sprint Review)
- Meeting/Project Presentation กับผู้ว่าจ้าง หรือ กับคณาจารย์

Verification

- การที่ต้องยืนยันว่าระบบถูกสร้างตาม specification ที่ได้ระบุไว้ ตลอดขั้นตอนการพัฒนาระบบหรือไม่
 - Verification เกิดขึ้นได้ตลอดการพัฒนาซอฟต์แวร์
 - Software Testing
 - Static Analysis
 - Code Review
 - Code Inspection
 - Formal Proof
 - และอื่นๆ
- 
- แต่ละเทคนิค มีเป้าหมายที่ต่างกัน หรือเพ่งเป้าไปที่หน่วยที่ต่างกัน ใน การทดสอบระบบ

Fault, Failure, Error

- **Error**

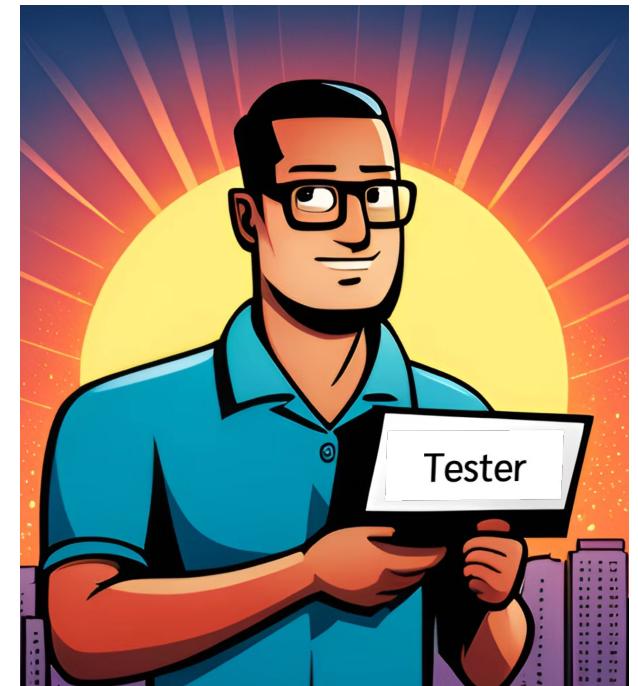
- สาเหตุของข้อผิดพลาด เกิดจากคนเป็นส่วนใหญ่ เช่น เรียกคำสั่งผิด พิมพ์คำสั่งผิด

- **Fault/Defect (bug)**

- ผลลัพธ์ของ Error หรือที่รู้จักกันดีว่า Bug. อาจก่อให้เกิด Failure แต่ก็ไม่เสมอไป

- **Failure**

- ข้อผิดพลาดของตัวโปรแกรมที่สังเกตเห็นได้ เมื่อโปรแกรมรันผ่าน Fault นั้นๆ (e.g., Crash, Freeze)



Example

```
1 def squareFunction(value):  
2     output = value * 2  
3     return output
```

print(squareFunction(2))

ผลลัพธ์เป็น 4

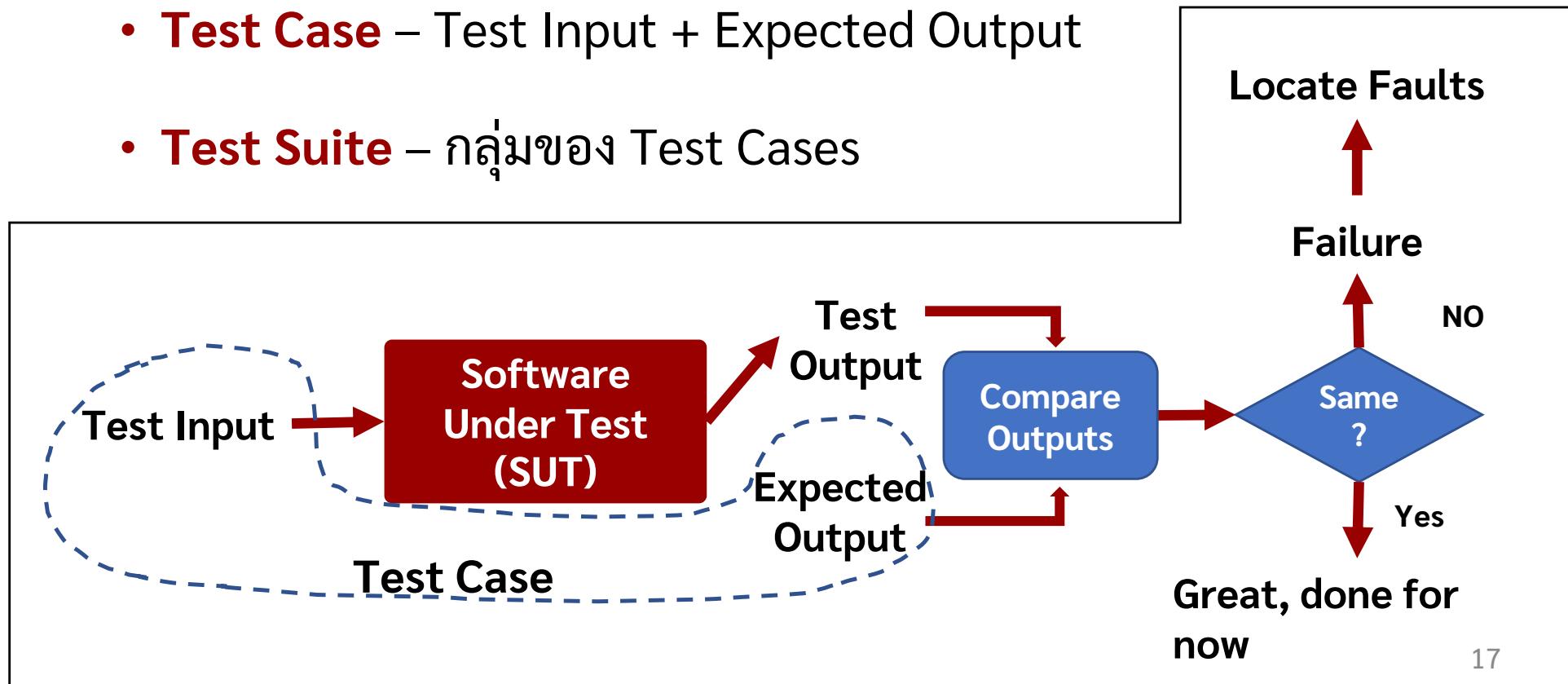
- Failure?

- Fault?

- Error?

Basic Terminology and Process

- **Test Input** – ข้อมูลที่จะนำไปใช้ในการทดสอบ
- **Expected Output** – ผลลัพธ์หรือพฤติกรรมของระบบที่คาดหวังจะเห็นหรือควรจะเป็น
- **Test Case** – Test Input + Expected Output
- **Test Suite** – กลุ่มของ Test Cases



Why is Testing Software Hard?

Coincidental Correctness

Program A

```
//This function returns  
//x squared  
  
1. def square(x):  
2.     result = x * 2;  
3.     return result;
```

Program B

```
//This function returns  
//the absolute value of x  
  
1. def absolute(x):  
2.     if(x < -2):  
3.         return -x  
4.     else:  
5.         return x
```

Test input {x = 2}

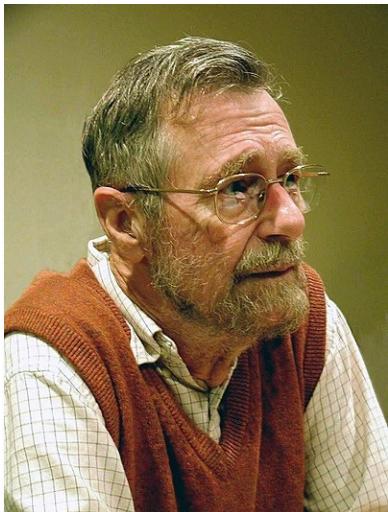
Test input {x = -3, -100, 0, 1, 2, 3}

A program can be *coincidentally correct* if
it executes a fault but does not fail

Coincidental Correctness



Just because tests do not reveal a failure does not mean there are no faults



“Testing shows the presence, not the absence of bugs”

การทดสอบแสดงถึงการมีอยู่ของ bug
ไม่ได้แสดงถึงการไม่มีอยู่ของ bug

- Edsger W. Dijkstra

Why not Just Test Everything – a Naïve Approach

```
//x and y are 32-bit integers  
  
1. bool prog1(int x, int y) {  
2.     //do something to x,y  
3.     //return something  
4. }
```

Q1

Test Input ทั้งหมดที่เป็นไปได้?

Q2

ถ้าจะเราจะสามารถทดสอบ 10^9 test inputs ใน 1 วินาที เราจะต้องใช้เวลาเท่าไร?



การทดสอบระบบด้วย test input ทั้งหมดนั้นทำได้ยากมาก และยิ่งแทบจะเป็นไม่ได้เลย สำหรับโปรแกรมที่ซับซ้อน

Generating Test Cases Randomly – a Naïve Approach

```
//x and y are 32-bit integers
1. int isEqual(int x, int y) {
2.     if (x == y)
3.         return false;
4.     else
5.         return false;
6. }
```

Q1

Test Input ทั้งหมดที่เป็นไปได้?

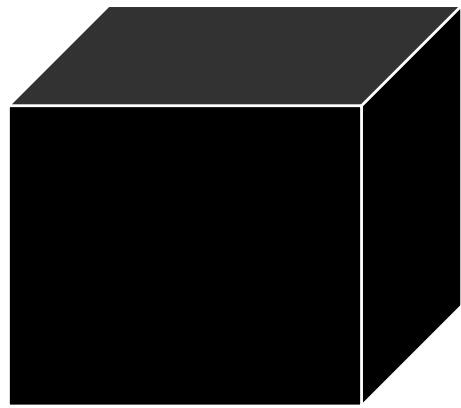
Q2

มีกี่ test input ที่ค่า x เท่ากับค่า y?

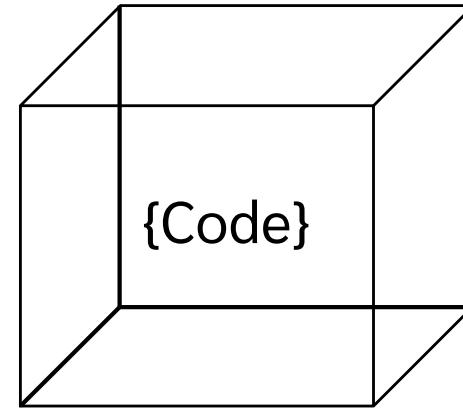
ค่าความน่าจะเป็น (probability) เมื่อ $x = y$?

General Approach To Testing

How Should Test Cases Be Selected?

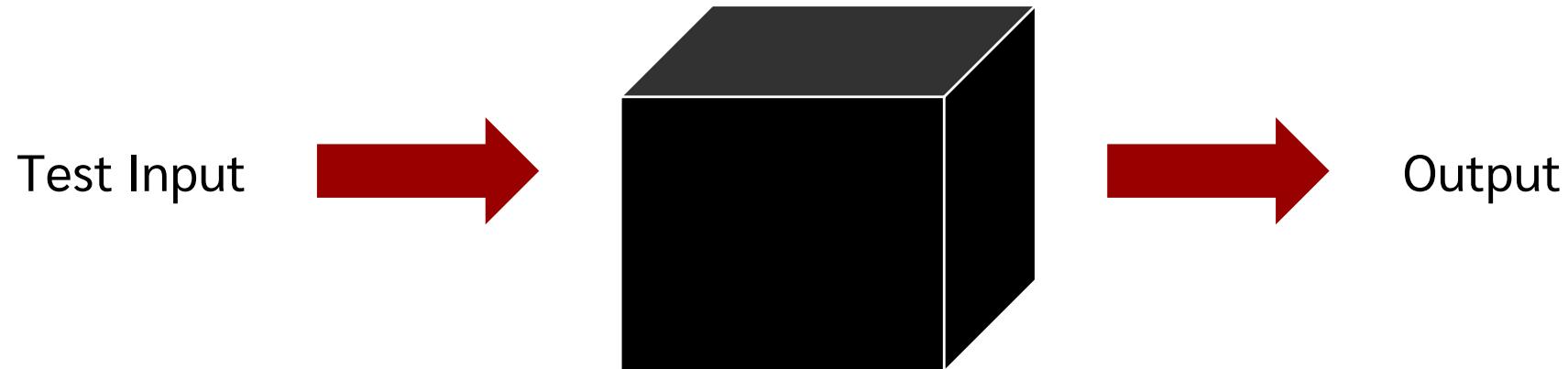


Black-box Testing
(Test Case สร้างขึ้นมาโดยอ้างอิง
specification)



White-box Testing
(Test Case สร้างขึ้นมาโดยอ้างอิง จาก
โครงสร้างภายในของ code
(structure elements of the code))

Black-box Testing



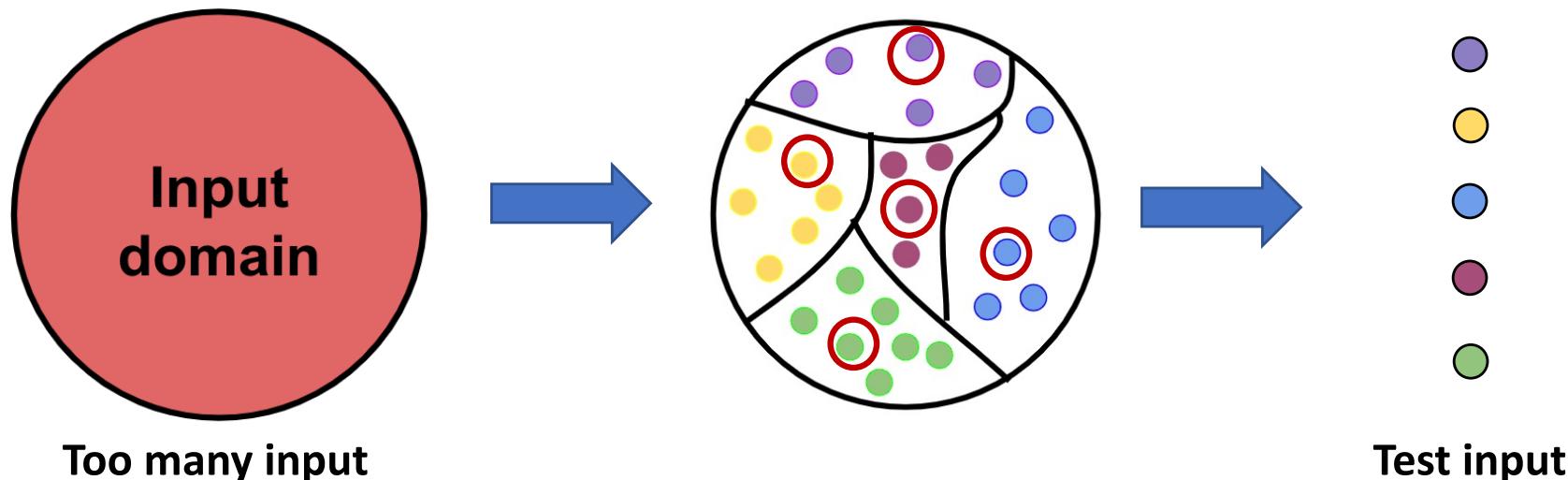
Black-box Testing
(Based on a **specification**)

- The program is considered a “black-box”
- Test cases are based on the **system specification**

Black-box Testing – An Introduction

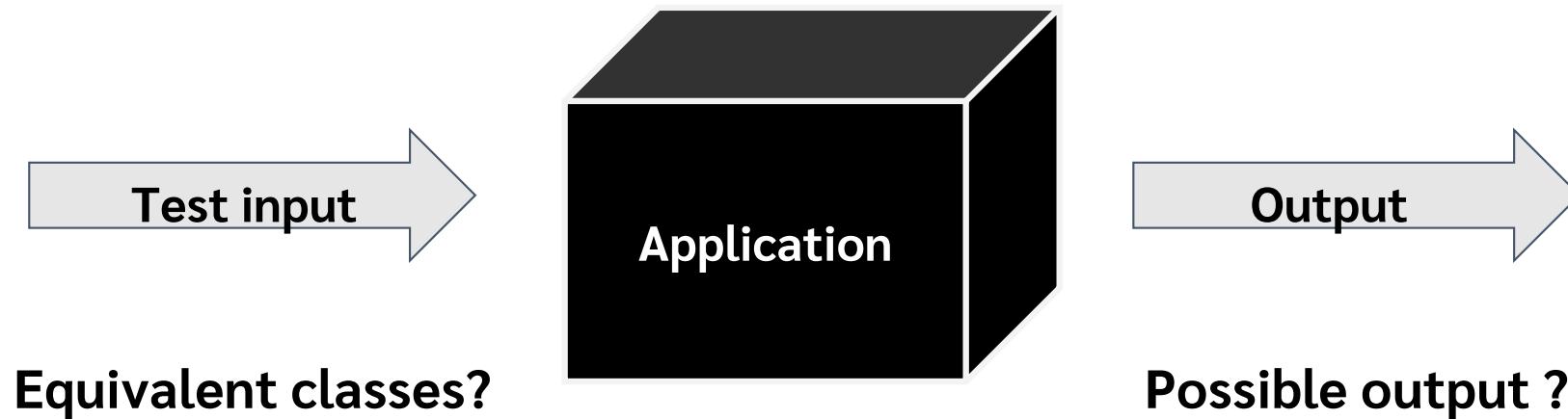
Equivalence Partitioning

Idea: จาก specification แบ่ง input data ออกเป็นหลายๆ คลาสที่ในแต่ละคลาสมีสมาชิกที่มีลักษณะเหมือนกันและคิดว่าเมื่อนำไปทดสอบกับระบบแล้วได้ผลออกมาเหมือนกัน **ให้เลือก 1 สมาชิกในแต่ละคลาสมาระบุเป็น test case**



Black-box Testing – An Introduction

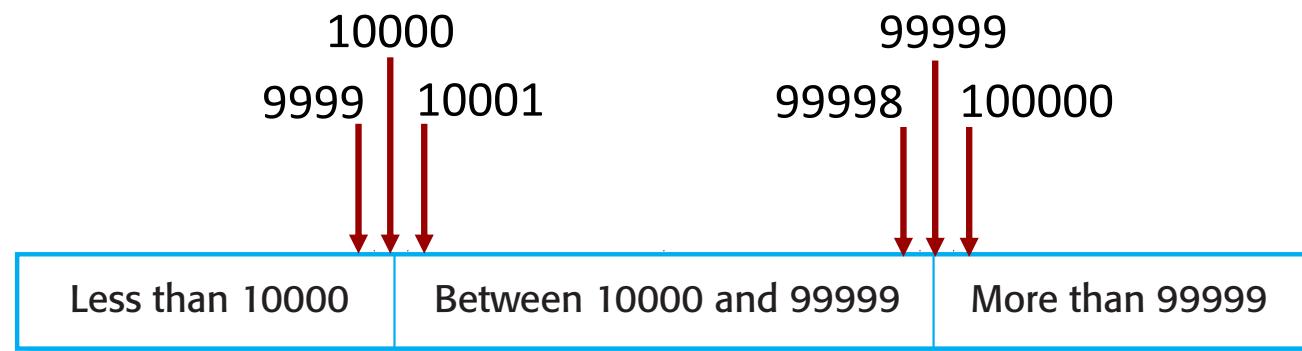
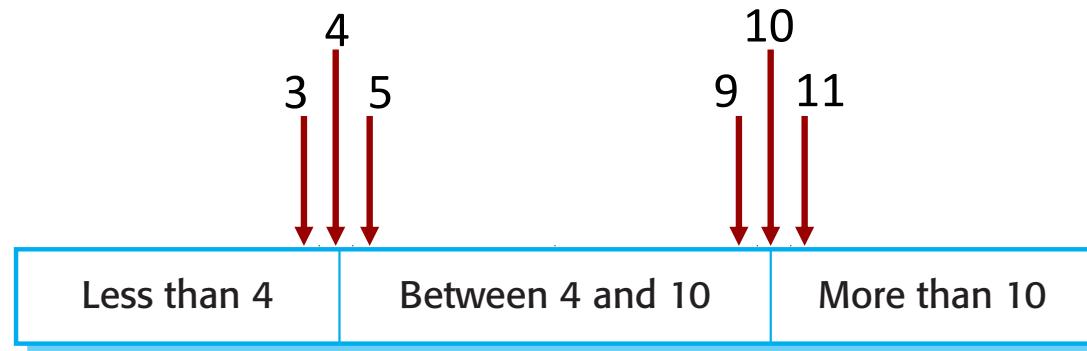
Scenario: One of the fields on a form contains a textbox which accepts numeric values in the range of 18 to 25.



Black-box Testing – An Introduction

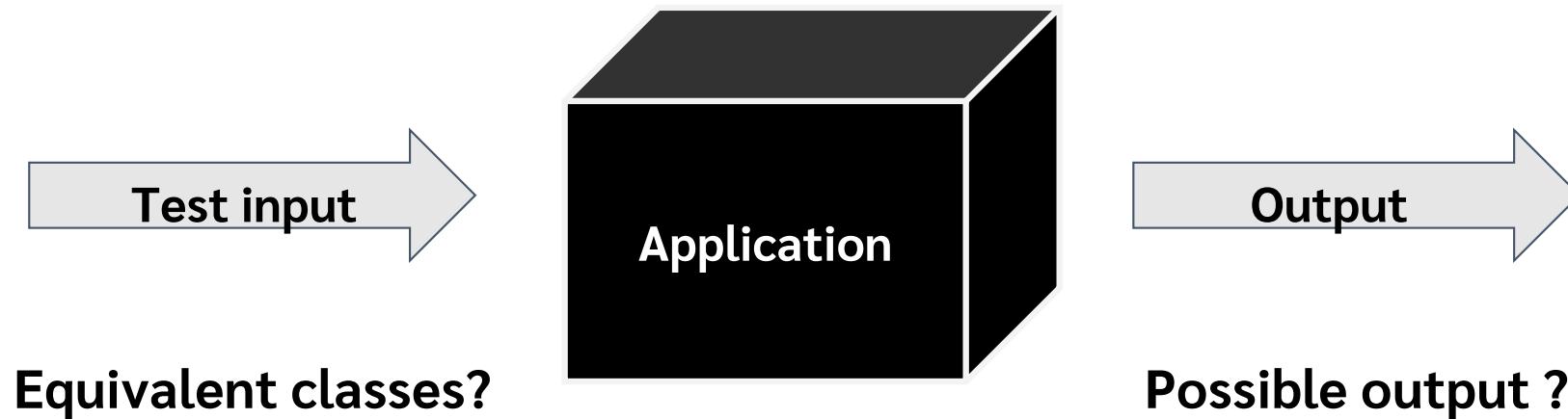
Boundary Conditions

Idea: ข้อผิดพลาดมักจะเกิดขึ้นใกล้ๆ กับขอบเขตของ class \Rightarrow ให้นำ test inputs ที่อยู่ใกล้ๆ กับขอบเขตมาทดสอบด้วย.



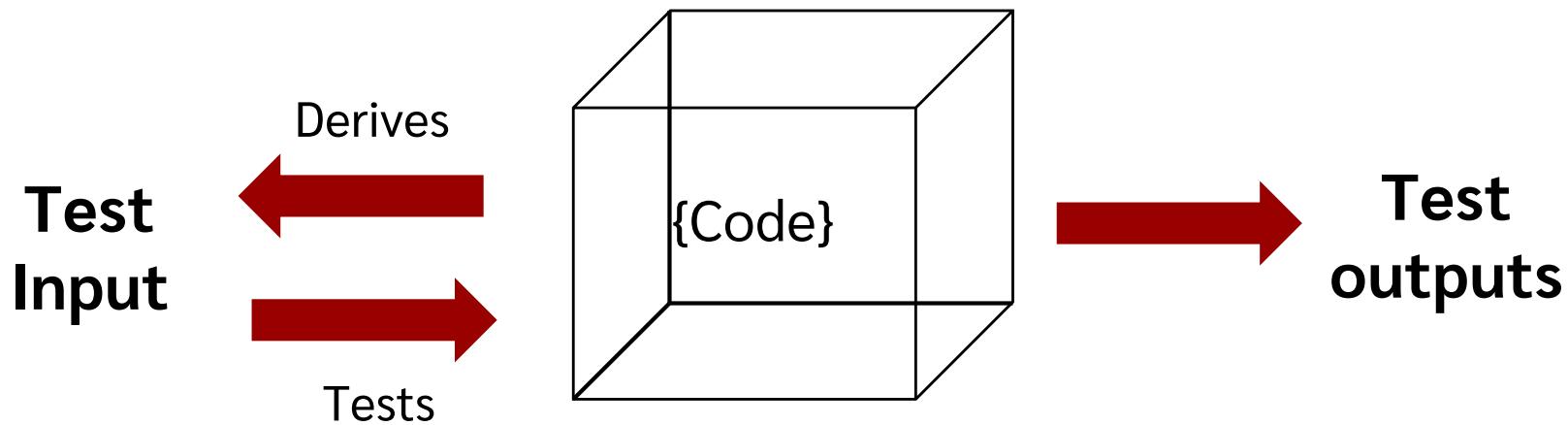
Black-box Testing – An Introduction

Scenario: One of the fields on a form contains a textbox which accepts numeric values in the range of 18 to 25.



White-box Testing

Overview:



Test case เขียนขึ้นมาโดยอ้างอิงจาก โครงสร้างภายในของโค้ด
structure elements of the code

มักจะเรียกมันว่า “Structural Testing” เพื่อทดสอบ
 เช่น ว่าโค้ดทุกบรรทัดถูกรันหรือไม่, ทุก Boolean
 expression ถูกทดสอบหรือไม่ เป็นต้น

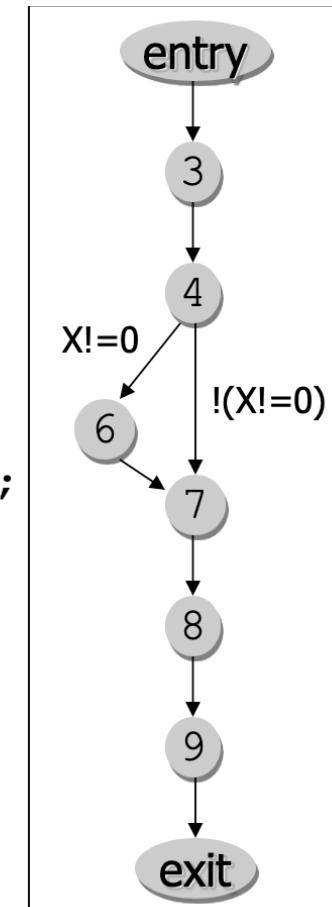
Why Structure Elements of the Code?

Idea: เงื่อนไขที่จำเป็นในการตรวจเจอ fault คือโค้ดที่เป็น fault นั้นต้องถูกรันใช้งาน (ถ้าไม่รันมัน ก็ไม่อาจรู้ได้ว่ามันเป็น fault)

วิธีการ

- แปลงโค้ดเป็นกราฟ
- หา test case ที่มาทดสอบ
โครงสร้างของกราฟ
เพื่อให้ได้ค่าครอบคลุม
(coverage) ที่ต้องการ

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x!=0)  
6.         x = x+10;  
7.     y = y/x;  
8.     write(x);  
9.     write(y);  
10. }
```



White-box Coverage Criteria

Statement Coverage

Basic Condition Coverage

Branch Coverage

Branch and Basic
Condition
Coverage

Multiple Condition
Coverage

Statement Coverage (White-box Testing)

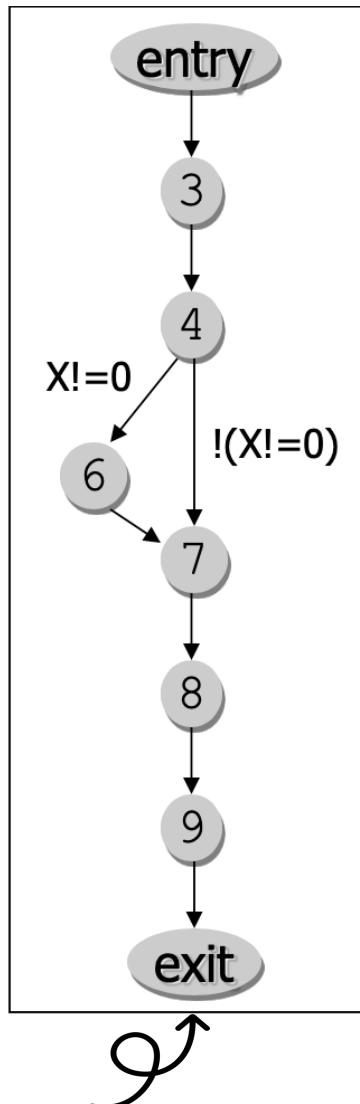
- **Statement Coverage** คือ ค่าการครอบคลุมของ statement (โค้ดในแต่ละบรรทัดหรือคำสั่ง) ที่ Test Case ได้รันมั่น
- คิดเป็น % โดยหาจาก

$$\text{Statement Coverage} = \frac{\text{จำนวน Statement ที่ถูกรัน}}{\text{จำนวน Statement ทั้งหมด}} \times 100$$

- **เป้าหมาย:** เรายังจะหา test suite ที่ทำให้ **ทุก Statement** ถูกรันอย่างน้อย 1 ครั้ง (เพื่อให้ได้ 100% Statement Coverage)

Statement Coverage Example

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x!=0)  
6.         x = x+10;  
7.     y = y/x;  
8.     write(x);  
9.     write(y);  
10. }
```



ในกราฟ Statement = Node (วงกลม)
โค้ดนี้มีทั้งหมด 8 Statements

Test Case อะไรบางที่เมื่อนำมา
ทดสอบกับ main() แล้วได้
100% Statement Coverage
(แต่ละ statement ถูกรันอย่าง
น้อย 1 ครั้ง)?

ข้อคิดที่ได้จากการตัวอย่าง

Branch Coverage (White-box Testing)

- **Branch Coverage** คือ ค่าการครอบคลุมของ Branch (เงื่อนไขใน if statement ถูกเรียกใช้งานทั้ง true และ false) ที่ Test Case ได้รันมัน
- คิดเป็น % โดยหาจาก

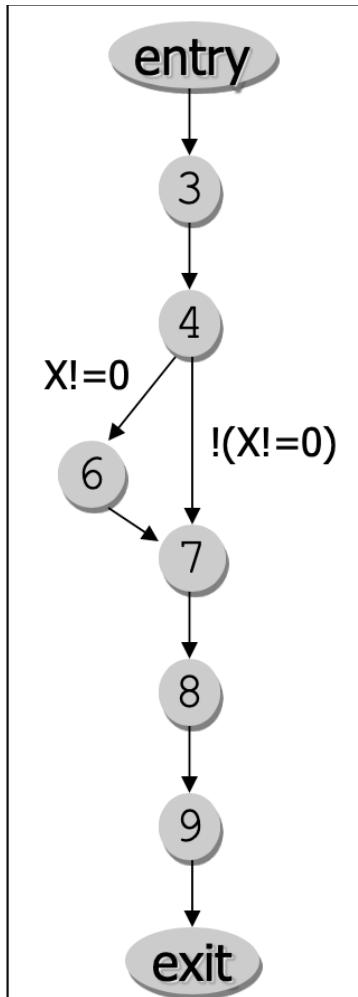
$$\text{Branch Coverage} = \frac{\text{จำนวน Branch ที่ถูกรัน}}{\text{จำนวน Branch ทั้งหมด}} \times 100$$



- **เป้าหมาย:** เรากำหนด test suite ที่ทำให้ **ทุก Branch** ถูกรันอย่างน้อย 1 ครั้ง (เพื่อให้ได้ 100% Branch Coverage)

Branch Coverage Example #1

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x!=0)  
6.         x = x+10;  
7.     y = y/x;  
8.     write(x);  
9.     write(y);  
10. }
```

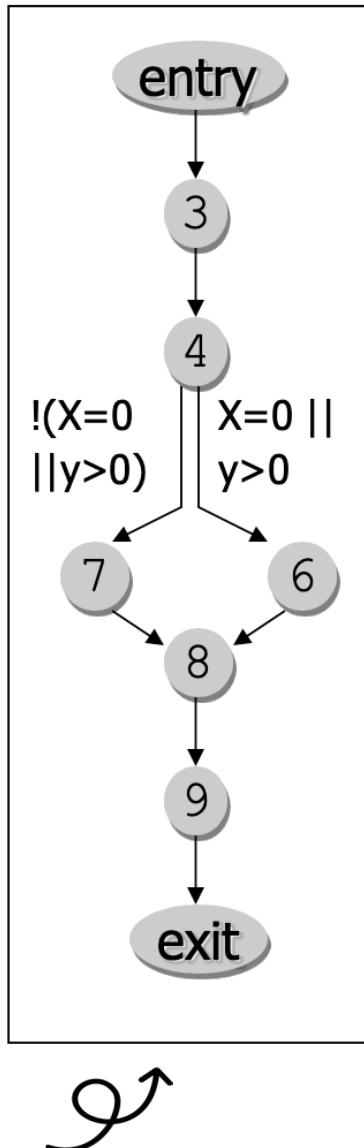


↗
Test Case อะไรบางที่เมื่อนำมา
ทดสอบกับ main() แล้วได้
100% Branch Coverage
(แต่ละ branch ถูกรันอย่างน้อย 1
ครั้ง)?

ในกราฟมีทั้งหมด 2 Branches 4-6-7 และ 4-7

Branch Coverage Example #2

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x==0) || (y>0)  
6.         y = y/x;  
7.     else x = y+2/x;  
8.     write(x);  
9.     write(y);  
10.}
```



ในกราฟมีทั้งหมด 2 Branches: 4-7-8 และ 4-6-8

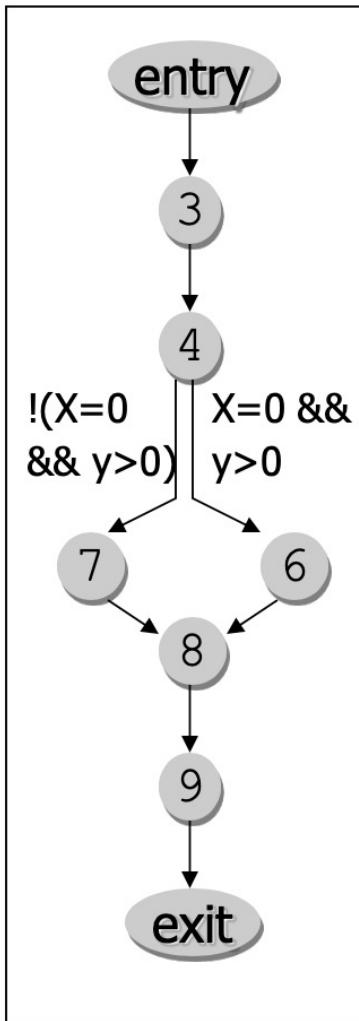
Test Case อะไรบ้างที่เมื่อนำมา
ทดสอบกับ main() แล้วได้
100% Branch Coverage
(แต่ละ branch ถูกรันอย่างน้อย 1
ครั้ง)?

ข้อคิดที่ได้จากการตัวอย่าง

Basic Condition Coverage

เป้าหมาย: เรากำหนดให้ test suite ที่ทำให้ทุก Boolean term (expression) ได้ผลเป็น True หรือ False อย่างน้อยอย่างละครั้ง

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x==0) && (y>0)  
6.         y = y/x;  
7.     else x = y+2;  
8.     write(x);  
9.     write(y);  
10.}
```



Test Case อะไรบางที่เมื่อนำมาทดสอบกับ main() แล้วได้ 100% Basic Condition Coverage

Test case #1:

- $X = 0$
- $Y = -2$

(T) && (F)

Test case #2:

- $X = 2$
- $Y = 3$

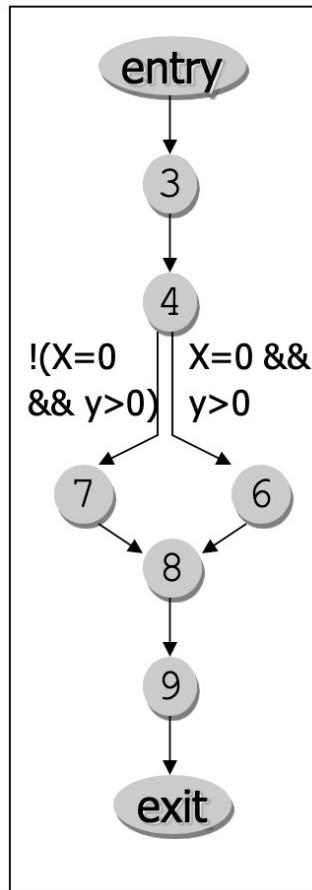
(F) && (T)

สอง Test cases ข้างบนนี้ทำให้เราได้ 100% Basic Condition Coverage เนื่องจาก แต่ละ expression เป็น T/F อย่างน้อย 1 ครั้ง

Branch and Basic Condition Coverage

เป้าหมาย: เรากำหนดให้ test suite ที่ทำให้ **ทุก Boolean term (expression)** ได้ผลเป็น True หรือ False อย่างน้อย 1 ครั้ง และ **ผล (decision)** รวมต้องออกมาเป็น True หรือ False อย่างน้อย 1 ครั้ง

```
1. void main() {  
2.     float x, y;  
3.     read(x);  
4.     read(y);  
5.     if (x==0) && (y>0)  
6.         y = y/x;  
7.     else x = y+2;  
8.     write(x);  
9.     write(y);  
10.}
```



Test case #1:

- X = 0
- Y = -2

Test case #2:

- X = 2
- Y = 3



100% Basic Condition Coverage

_____ % Branch Coverage

Multiple Condition Coverage

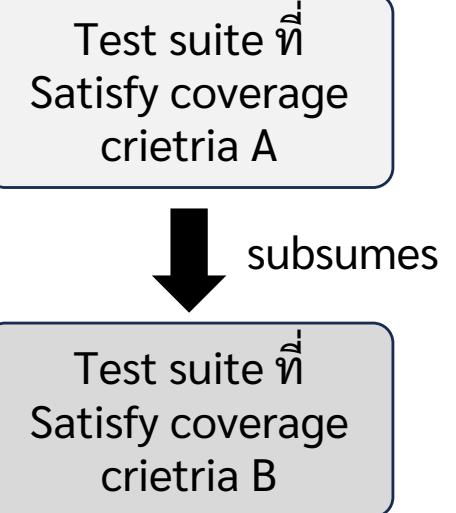
- **เป้าหมาย:** เรากำจดหา test suite ที่ทดสอบทุก combination ของ basic condition
- จัดเป็น theoretical เพราะสิ่นเปลี่ยนสูงและเพราะไม่ practical ถ้าจะทดสอบทุก combination

$$((((a \parallel b) \&& c) \parallel d) \&& e)$$

ต้องใช้ทั้งหมดกี่ test case?

Subsumption Hierarchy

เกณฑ์ความครอบคลุมการทดสอบ A จะรวม เกณฑ์ความครอบคลุมการทดสอบ B อิ่มทัยในตัวมันด้วย ก็ต่อเมื่อ หากใช้ test suite ทั้งหมดที่ satisfy A สำหรับโปรแกรม P ก็จะ satisfy B สำหรับ P ด้วย
ง่ายๆคือ Test Criteria A มีสมรรถภาพสูงกว่า B



Theoretical Criteria

Practical Criteria

Other Coverage Criteria

ต้องหา Test suite ที่...

Path Coverage

- ทุกเส้นทาง (path) ในโปรแกรมต้องถูกใช้งานอย่างน้อย 1 ครั้ง

Loop Coverage

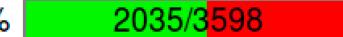
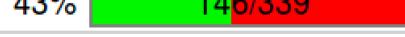
- loop ต้องถูกใช้งานอย่างน้อย 0 ครั้ง, 1 ครั้ง, หลายครั้ง $< X$, หรือ X ครั้ง

Data Flow Coverage

- All Defs – Definition** (โค้ดที่มีการ assign value to a variable (เช่น $x = 5$, $x = y+2$) ของทุก variable ต้องถูกใช้งาน
- All Uses –** เส้นทางจาก def (เช่น $x+5$) ไปถึง use (เช่น $total = x+y \leftarrow x$ ถูกใช้งาน) ของแต่ละ variable ต้องถูกทดสอบอย่างน้อย 1 เส้นทาง (อาจมีหลายเส้นทางจาก def – use)
- All DU paths –** ทุกเส้นทางจาก definition ไป use ของแต่ละ variable ต้องถูกใช้งาน

Coverage Report Example (Java JSP):

Coverage Report - org.apache.jsp

Package	# Classes	Line Coverage	Branch Coverage
org.apache.jsp	9	56%  2035/3598	28%  553/1932
Classes in this Package		Line Coverage	Branch Coverage
AdminMenu.jsp		45%  160/349	18%  36/200
Default.jsp		62%  289/461	29%  63/217
DepsGrid.jsp		48%  182/373	22%  44/199
DepsRecord.jsp		58%  231/394	34%  83/239
EmpDetail.jsp		55%  248/447	22%  50/221
EmpsGrid.jsp		67%  299/443	44%  101/225
EmpsRecord.jsp		79%  412/518	51%  135/264
Header.jsp		24%  68/274	3%  6/165
Login.jsp		43%  146/339	17%  35/202

Class ຕຳຫຼາດໃນ package

% coverage

Coverage Report Example (JS):

All files src/components

93.33% Statements 28/30 75% Branches 6/8 84.62% Functions 11/13 93.33% Lines 28/30

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements	Branches	Functions	Lines
AboutPage.js	50%	1/2	0%	50%
App.js	100%	3/3	100%	100%
FuelSavingsForm.js	100%	3/3	100%	100%
FuelSavingsPage.js	100%	7/7	100%	100%
FuelSavingsResults.js	100%	6/6	50%	100%
FuelSavingsTextInput.js	100%	4/4	100%	100%
HomePage.js	100%	2/2	100%	100%
NotFoundPage.js	50%	1/2	0%	50%
Root.js	100%	1/1	100%	100%

Using Cypress

Coverage Report Example (Python)

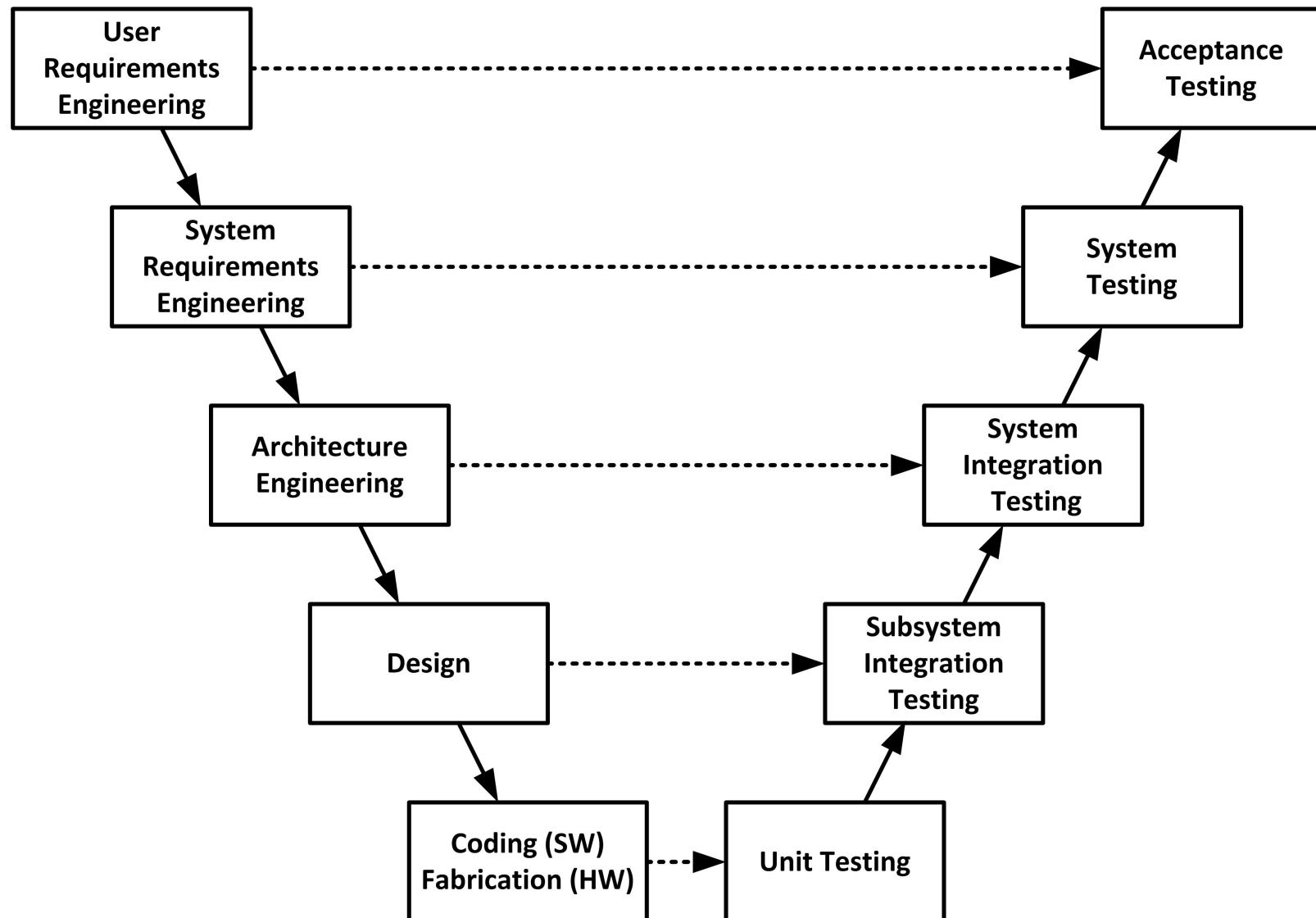
Cog coverage: 38.75%						
Module	statements	missing	excluded	branches	partial	coverage
cogapp/__init__.py	1	0	0	0	0	100.00%
cogapp/__main__.py	3	3	0	0	0	0.00%
cogapp/cogapp.py	500	224	1	210	30	49.01%
cogapp/makefiles.py	22	18	0	14	0	11.11%
cogapp/test_cogapp.py	845	591	2	24	1	29.57%
cogapp/test_makefiles.py	70	53	0	6	0	22.37%
cogapp/test_whiteutils.py	68	50	0	0	0	26.47%
cogapp/whiteutils.py	43	5	0	34	4	88.31%
Total	1552	944	3	288	35	38.75%

coverage.py v7.2.7, created at 2023-05-29 15:26 -0400

Using Python's Coverage Package

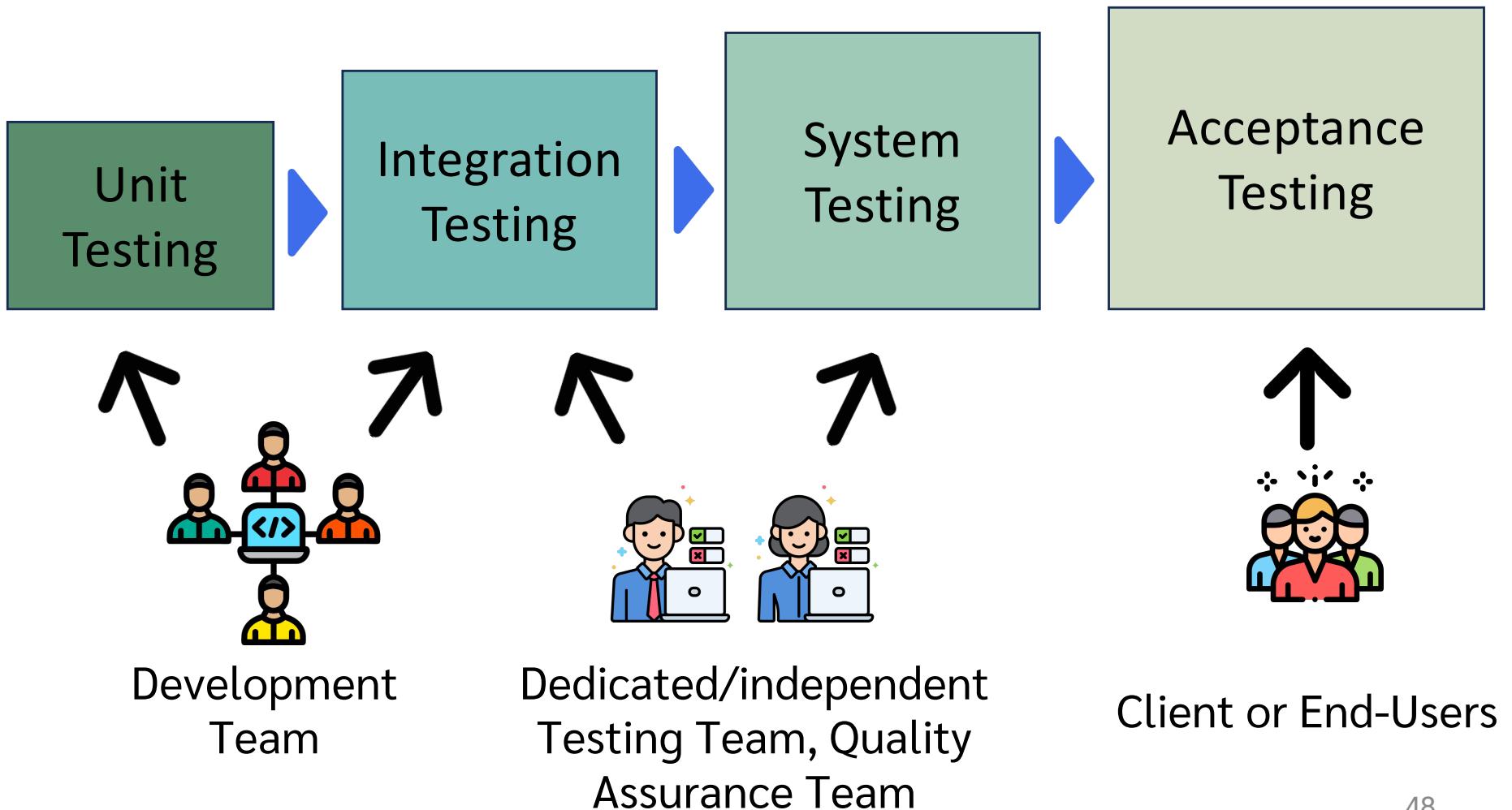
Levels of Testing

Levels of Testing (V-model)



Software Testing Levels

Who performs the tests?



Development Team Not Test Every Level?

เหตุผลหลักที่ Software Development Team มักจะมีทีม สำหรับทดสอบระบบโดยเฉพาะที่แยกต่างหาก เช่น ทีม Quality Assurance หรือ Independent Testers คือ



Bias and Familiarity

“เราเขียนโค้ดตรงนี้เอง ถูก/เวิร์คซั่ว ไม่ต้องทดสอบหรอก”

“มันเวิร์คบนเครื่องเราอ่ะ ไม่ต้องทดสอบหรอก เสียเวลา”

“เขียนโค้ดเองแล้วทดสอบเองแล้ว ผลคือผ่านหมดเลย”

Unit Testing

Does each unit work as specified?

- การทดสอบการทำงานของซอฟต์แวร์ในแต่ละหน่วยการทำงานที่เล็กที่สุดที่สามารถทดสอบได้ (มีการใช้งาน assert function และ สามารถเป็นได้ทั้ง White-box และ black-box)
- หน่วยในที่นี่อาจเป็น function, class, หรือ method หนึ่งๆ
- ผู้ทดสอบคือ **Development Team** ของระบบ เพราะเข้าใจแต่ละส่วนมากที่สุด
- อาจต้องมีการสร้าง **Mock** (Stub หรือ Driver) มาช่วยจำลองด้านต่างๆ เช่น ฐานข้อมูล, การเชื่อมต่อ network, การรับค่าข้อมูล เพื่อให้ unit ที่กำลังทดสอบสามารถทดสอบได้ (แต่ไม่ต้องไปต่อ service อื่นจริงๆ – no dependency)

Test Driver vs Stub

Driver และ **Stub** คือโมดูลหรือโปรแกรมจำลองบางหน่วยของระบบทำให้เราสามารถทดสอบโมดูลบางโมดูลได้หากมันต้องการค่าหรือข้อมูลอะไรบางอย่างหรือต้องถูกเรียกใช้งานบางอย่างจากโมดูลอื่น

Driver

โปรแกรมหรือโมดูลจำลองที่รับข้อมูลกรณีทดสอบแล้วส่งผ่านข้อมูลนั้นไปองค์ประกอบหรือโมดูลที่ถูกทดสอบในระดับที่ต่ำกว่า

เช่น ทดสอบ Stack ADT ที่มี method pop, push, และ print ตัว driver ก็คือโปรแกรมหรือโมดูลที่มารับค่าต่างๆ จาก user เพื่อให้ method พากนั้นถูกเรียกใช้งาน

Stub

โมดูลจำลองที่มี interface, API, method เหมือนโมดูลจริง แต่ไม่มีการประมวลผลข้อมูล อาจมีการตรวจ (verify) ข้อมูลอย่างง่าย ส่งค่าหรือให้ผลลัพธ์ด้วยค่าที่ถูกกำหนดไว้ เพื่อที่เราจะได้สามารถควบคุมพฤติกรรมของระบบส่วนด้านบนให้เป็นอย่างที่เราต้องการ เช่น โมดูลจำลองที่ให้ค่าเป็น true เมื่อเมื่อถูกเรียก

Integration Testing

Do the units work when combined?

- ทดสอบว่าเมื่อ unit ต่างๆมาร่วมกันมั้นยังสามารถทำงานได้ถูกต้องอยู่
- วิธีนี้จะทดสอบโปรแกรมโดยการเพิ่มจำนวนโมดูลขึ้นเรื่อยๆ
- จะเป็นการทดสอบโดยที่ unit ใน slide ก่อนหน้า มีการไปต่อ กับ Service อื่นๆที่มันจะต้องทำงานด้วยจริงๆ เช่น Network, Database, Other Service หรือ Party ต่างๆ
- จะเป็นการทดสอบด้วย Development Team หรือ ทีมทดสอบที่ไม่ใช่ ส่วนใน Dev เป็นทีมแยกออกจากต่างหาก
- รวมถึงการทดสอบตอนนำ increments ของแต่ละ sprint มารวมกัน

System Testing

Does the system work as a whole?
And how well?

- เป็นการทดสอบระบบโดยรวม (หรือทั้งระบบ) หลังจากที่ได้ทุกส่วนเชื่อมต่อกัน
กล้ายเป็นระบบแล้วหรือ integrate มาหมดแล้ว
- ระบบจะถูกทดสอบโดยทีมทดสอบที่ไม่ใช่ development team
- ทดสอบ functionality ที่ระบุใน specification (ตาม user story) หรือ requirements รวมทดสอบประสิทธิภาพของพาก non-functional requirements ร่วมด้วย เช่น
 - performance testing,
 - interface testing,
 - scalability testing
 - เป็นต้น

Acceptance Testing

Does the system meet the stakeholders' requirements and expectations?

- เป็นกระบวนการทดสอบในระดับท้ายสุดก่อนที่ระบบจะถูกปล่อยออกใช้งานจริง โดย client หรือ end-user
- Alpha (*in-house*) และ Beta (*few selected users* ไม่ใช่ dev) Testing ก็เป็นส่วนหนึ่ง
- กระบวนการคร่าวๆ คือ:
 - Dev จะต้องเตรียมข้อมูลทดสอบหรือ scenario ที่จะให้ผู้ใช้ทดสอบ
 - ผู้ใช้จะทำการทดสอบระบบตาม scenario โดยที่ dev ห้ามบอกผู้ใช้ว่าต้องกด ตรงนี้ไปตรงนี้เลือกตรงนี้
 - Dev ถาม feedback จากผู้ใช้หลังจากผู้ใช้ได้ทดสอบระบบ หรือสังเกตว่าเกิด ข้อผิดพลาดอะไรตรงไหนตอนผู้ใช้งานระบบบ้าง ให้บันทึกแล้วกลับไปแก้ไข
 - ถ้าข้อผิดพลาดมีน้อยหรืออยู่ในระดับที่ผู้ใช้พอใจ ก็สามารถปล่อยระบบออกใช้งานจริงได้

Questions ?

