สัปดาห์ที่ 2: คุณภาพซอฟต์แวร์และเมตริกคุณภาพ (Software Quality and Quality Metrics)

วัตถุประสงค์การเรียนรู้ (Learning Objectives)

นิสิตจะสามารถ

- นิยามคุณภาพซอฟต์แวร์และเข้าใจความสำคัญของคุณภาพซอฟต์แวร์
- ระบุและอธิบายลักษณะคุณภาพ 8 ประการจากมาตรฐาน ISO 25010
- คำนวณเมตริกคุณภาพที่สำคัญ (Defect Density, Test Coverage, DRE)
- นำโมเดลคุณภาพไปใช้ประเมินระบบซอฟต์แวร์
- เข้าใจการคืนทุน (ROI) ของการทดสอบซอฟต์แวร์

ส่วนที่ 1: การทำความเข้าใจคุณภาพซอฟต์แวร์

คุณภาพซอฟต์แวร์คืออะไร (What is Software Quality?)

นิยาม (Definition): คุณภาพซอฟต์แวร์คือระดับของความสอดคล้องระหว่างผลิตภัณฑ์ซอฟต์แวร์กับความต้องการที่ระบุไว้และความคาดหวังของผู้ใช้

มุมมองที่สำคัญ (Key Perspectives):

1. **มุมมองของผู้ใช้ (User Perspective)**: ทำสิ่งที่ฉันต้องการได้หรือไม่? ใช้งานง่ายหรือไม่?
2. **มุมมองของผู้พัฒนา (Developer Perspective)**: โค้ดนั้นสามารถบำรุงรักษาได้หรือไม่? ทำให้เป็นโครงสร้างที่ดีหรือไม่?
3. **มุมมองของธุรกิจ (Business Perspective)**: มันส่งมูลค่ามาให้หรือไม่? คุ้มค่าหรือไม่?

เหตุใดคุณภาพจึงสำคัญ (Why Quality Matters)

ตัวอย่างจากโลกจริง (Real-World Examples):

1. **เครื่องฉายรังสี Therac-25 (1985-1987)**
    - ความผิดพลาดของซอฟต์แวร์ทำให้ผู้ป่วยได้รับรังสีในปริมาณที่สูงเกินขนาดหลายพันเท่า
    - ผลลัพธ์: บาดเจ็บสาหัสและเสียชีวิตหลายราย
    - สาเหตุหลัก: การควบคุมคุณภาพที่ไม่ดีและการทดสอบที่ไม่เพียงพอ
2. **ข้อผิดพลาดการซื้อขายของ Knight Capital Group (2012)**
    - ข้อผิดพลาดในการใช้งานซอฟต์แวร์ในอัลกอริทึมการซื้อขาย
    - ผลลัพธ์: ขาดทุน 440 ล้านดอลลาร์ใน 45 นาที
3. **ปัญหาการเร่งความเร็วที่ไม่ตั้งใจของโตโยต้า (2009-2010)**
    - ปัญหาคุณภาพซอฟต์แวร์ในการควบคุมเบรกอิเล็กทรอนิกส์
    - ผลลัพธ์: การเรียกคืนหลายล้านคันทั่วโลก
    - ต้นทุน: กว่า 3 พันล้านดอลลาร์ในการชดใช้ค่าเสียหาย

สิ่งที่ต้องจำ (Key Takeaway): คุณภาพที่ไม่ดี = ต้นทุนสูง

ส่วนที่ 2: โมเดลคุณภาพ ISO 25010

ISO 25010: มาตรฐานสากลสำหรับคุณภาพซอฟต์แวร์

ISO 25010 นิยามคุณลักษณะคุณภาพ **8 ประการ** ที่สร้างคุณภาพซอฟต์แวร์:

1. ความเหมาะสมของฟังก์ชันการทำงาน (Functional Suitability) ซอฟต์แวร์ทำสิ่งที่ควรจะทำได้หรือไม่?

ลักษณะย่อย (Sub-characteristics):

- **ความครบถ้วนของฟังก์ชันการทำงาน (Functional Completeness)**: มีฟังก์ชันที่ระบุไว้ทั้งหมด
- **ความถูกต้องของฟังก์ชันการทำงาน (Functional Correctness)**: ฟังก์ชันให้ผลลัพธ์ที่ถูกต้อง
- **ความเหมาะสมของฟังก์ชันการทำงาน (Functional Appropriateness)**: ฟังก์ชันอำนวยความสะดวกในการทำงานที่ระบุไว้

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): ค้นหาหนังสือให้ผลลัพธ์ที่ถูกต้อง, กระบวนการยืมใช้งานได้อย่างถูกต้อง

ไม่ดี (Bad): ค้นหาคืนหนังสือที่ไม่ตรงกับเกณฑ์, ยืมหนังสือที่ยืมแล้ว

วิธีทดสอบ (How to Test):

- ทดสอบแต่ละฟีเจอร์กับข้อกำหนด
- กรณีทดสอบบวกและลบ (Positive and Negative Test Cases)
- การวิเคราะห์ค่าขอบเขต (Boundary Value Analysis)

---

2. ประสิทธิภาพการใช้ทรัพยากร (Performance Efficiency) ซอฟต์แวร์ใช้ทรัพยากรได้ดีเพียงใด?

ลักษณะย่อย (Sub-characteristics):

- **พฤติกรรมเวลา (Time Behavior)**: เวลาตอบสนอง, เวลาประมวลผล
- **การใช้ทรัพยากร (Resource Utilization)**: CPU, หน่วยความจำ, เครือข่าย
- **ความจุ (Capacity)**: ขีดจำกัดสูงสุด (ผู้ใช้, ปริมาณข้อมูล)

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): ผลลัพธ์การค้นหาใน < 2 วินาที, รองรับผู้ใช้พร้อมกัน 1000 คน

ไม่ดี (Bad): ค้นหาใช้เวลา 30 วินาที, ขัดข้องกับผู้ใช้ 50 คน

เมตริก (Metrics):

- เวลาตอบสนอง (ค่าเฉลี่ย, เปอร์เซนไทล์ที่ 95)
- ปริมาณการประมวลผล (คำขอ/วินาที)
- การใช้ทรัพยากร (CPU %, หน่วยความจำ %)

---

3. ความเข้ากันได้ (Compatibility) สามารถทำงานร่วมกับระบบอื่น ๆ ได้หรือไม่?

ลักษณะย่อย (Sub-characteristics):

- **การอยู่ร่วมกัน (Co-existence)**: สามารถทำงานเคียงข้างกับซอฟต์แวร์อื่น ๆ ได้
- **การทำงานร่วมกัน (Interoperability)**: สามารถแลกเปลี่ยนข้อมูลกับระบบอื่น ๆ ได้

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): รวมเข้ากับฐานข้อมูลนิสิตของมหาวิทยาลัย, ส่งออกข้อมูลไปยัง Excel

ไม่ดี (Bad): ทำให้เกิดความขัดแย้งกับระบบอื่น ๆ ในวิทยาเขต, รูปแบบข้อมูลเฉพาะ

วิธีทดสอบ (How to Test):

- ทดสอบการรวมเข้ากับระบบภายนอก
- ทดสอบการนำเข้า/ส่งออกข้อมูล
- ทดสอบบนแพลตฟอร์มต่างๆ

---

4. **ความใช้งานง่าย (Usability)** ใช้งานง่ายเพียงใด?

ลักษณะย่อย (Sub-characteristics):

- **ความเหมาะสมของการรับรู้ (Appropriateness Recognizability)**: ชัดเจนว่ามันทำอะไร?
- **ความเรียนรู้ (Learnability)**: ผู้ใช้สามารถเรียนรู้ได้เร็วแค่ไหน?
- **ความสามารถในการทำงาน (Operability)**: ใช้งานง่ายแค่ไหน?
- **การป้องกันข้อผิดพลาดของผู้ใช้ (User Error Protection)**: ป้องกันข้อผิดพลาดของผู้ใช้
- **สุนทรียะของส่วนติดต่อผู้ใช้ (User Interface Aesthetics)**: การออกแบบที่ดึงดูดและน่าพอใจ
- **ความสามารถในการเข้าถึง (Accessibility)**: สามารถใช้งานได้โดยผู้ที่มีความพิการ

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): การนำทางที่สัญชาตญาณ, ข้อความข้อผิดพลาดที่ชัดเจน, แป้นพิมพ์ลัด

ไม่ดี (Bad): เมนูที่สับสน, ไม่มีข้อความช่วยเหลือ, ไม่สามารถเข้าถึงได้สำหรับผู้อ่านหน้าจอ

วิธีทดสอบ (How to Test):

- การทดสอบความใช้งานง่ายกับผู้ใช้จริง (Usability Testing)
- การทดสอบความสามารถในการเข้าถึง (Accessibility Testing) - การปฏิบัติตามมาตรฐาน WCAG
- การวัดเวลาในการทำงาน (Time-to-complete-task Measurements)

---

5. **ความเชื่อถือได้ (Reliability)** ทำงานอย่างสม่ำเสมอโดยไม่มีความล้มเหลวหรือไม่?

ลักษณะย่อย (Sub-characteristics):

- **ความสำเร็จในการใช้งาน (Maturity)**: ล้มเหลวบ่อยแค่ไหน?
- **ความพร้อมใช้งาน (Availability)**: พร้อมใช้งานเมื่อต้องการหรือไม่?
- **ความทนต่อข้อผิดพลาด (Fault Tolerance)**: สามารถจัดการข้อผิดพลาดอย่างสง่างามได้หรือไม่?
- **ความสามารถในการกู้คืน (Recoverability)**: สามารถกู้คืนจากความล้มเหลวได้หรือไม่?

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): ระดับความพร้อมใช้งาน 99.9%, บันทึกข้อมูลอัตโนมัติ, การจัดการข้อผิดพลาดอย่างสง่างาม

ไม่ดี (Bad): ขัดข้องทุกวัน, สูญเสียข้อมูลเมื่อเกิดข้อผิดพลาด, เวลาในการกู้คืนนาน

เมตริก (Metrics):

- เวลาเฉลี่ยระหว่างความล้มเหลว (Mean Time Between Failures - MTBF)
- เวลาเฉลี่ยในการซ่อมแซม (Mean Time To Repair - MTTR)
- ความพร้อมใช้งาน % = (เวลาทั้งหมด - เวลาหยุดทำงาน) / เวลาทั้งหมด × 100

---

6. **ความปลอดภัย (Security)** ป้องกันข้อมูลได้ดีเพียงใด?

ลักษณะย่อย (Sub-characteristics):

- **ความลับ (Confidentiality)**: ป้องกันการเข้าถึงข้อมูลโดยไม่ได้รับอนุญาต
- **ความสมบูรณ์ (Integrity)**: ป้องกันการแก้ไขข้อมูลโดยไม่ได้รับอนุญาต
- **การไม่ปฏิเสธ (Non-repudiation)**: สามารถพิสูจน์ได้ว่าการกระทำเกิดขึ้นจริง
- **ความรับผิดชอบ (Accountability)**: สามารถติดตามการกระทำไปยังเอนทิตี้ได้
- **ความสามารถในการตรวจสอบสิ่งตัวตน (Authenticity)**: สามารถพิสูจน์ตัวตนได้

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): รหัสผ่านที่เข้ารหัส, การเข้าถึงตามบทบาท, บันทึกการตรวจสอบ

ไม่ดี (Bad): รหัสผ่านแบบข้อความธรรมชาติ, ใครก็ได้สามารถเข้าถึงฟังก์ชันผู้ดูแล

วิธีทดสอบ (How to Test):

- การทดสอบการแทรกซึมระบบ (Penetration Testing)
- การทดสอบการฉีด SQL (SQL Injection Testing)
- การทดสอบการพิสูจน์ตัวตนและการให้สิทธิ์

---

7. **ความสามารถในการบำรุงรักษา (Maintainability)** แก้ไขและปรับปรุงได้ง่ายแค่ไหน?

ลักษณะย่อย (Sub-characteristics):

- **ความเป็นโมดูลาร์ (Modularity)**: ประกอบด้วยส่วนประกอบที่เป็นอิสระ
- **ความสามารถในการนำกลับมาใช้ (Reusability)**: สามารถใช้ส่วนประกอบในระบบอื่น ๆ ได้
- **ความสามารถในการวิเคราะห์ (Analyzability)**: ง่ายในการวินิจฉัยปัญหา
- **ความสามารถในการปรับเปลี่ยน (Modifiability)**: ง่ายในการแก้ไขโดยไม่นำเข้าข้อบกพร่อง
- **ความสามารถในการทดสอบ (Testability)**: ง่ายในการทดสอบการปรับเปลี่ยน

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): โค้ดแบบโมดูลาร์, เอกสารที่ชัดเจน, การทดสอบที่ครอบคลุม

ไม่ดี (Bad): โค้ดสปาเกตตี (Spaghetti Code), ไม่มีความเห็น, การเปลี่ยนสิ่งหนึ่งทำลายทุกอย่าง

เมตริก (Metrics):

- ความซับซ้อนแบบวัฏจักร (Cyclomatic Complexity)
- ความครอบคลุมของโค้ด (Code Coverage)
- หน่วยโค้ดที่มีปัญหา (Code Smells) - จากการวิเคราะห์แบบคงที่

---

8. **ความสามารถในการปรับตัว (Portability)** โอนไปยังสภาพแวดล้อมอื่นได้ง่ายแค่ไหน?

ลักษณะย่อย (Sub-characteristics):

- **ความสามารถในการปรับตัว (Adaptability)**: สามารถปรับตัวให้เข้ากับสภาพแวดล้อมต่างๆ ได้
- **ความสามารถในการติดตั้ง (Installability)**: ติดตั้งได้ง่ายในสภาพแวดล้อมต่างๆ
- **ความสามารถในการเปลี่ยน (Replaceability)**: สามารถแทนที่ผลิตภัณฑ์ซอฟต์แวร์อื่นได้

ตัวอย่างระบบห้องสมุด (Library System Example):

ดี (Good): ทำงานบน Windows/Mac/Linux, Docker Container, พร้อมใช้ในระบบคลาวด์

ไม่ดี (Bad): ทำงานเฉพาะบน Windows XP เท่านั้น, คู่มือการติดตั้ง 10 หน้า

วิธีทดสอบ (How to Test):

- ทดสอบบนระบบปฏิบัติการต่างๆ
- ทดสอบบนเบราว์เซอร์ต่างๆ
- ทดสอบกระบวนการปรับใช้ (Deployment Process)

ส่วนที่ 3: เมตริกคุณภาพ (Quality Models & Standards)

ทำไมต้องวัดคุณภาพ (Why Measure Quality?)

"คุณไม่สามารถจัดการสิ่งที่คุณไม่สามารถวัดได้" - Peter Drucker

ประโยชน์ของเมตริก (Benefits of Metrics):

- ติดตามความก้าวหน้าเมื่อเวลาผ่านไป
- ระบุพื้นที่ที่มีปัญหา
- ตัดสินใจโดยยึดข้อมูล
- สื่อสารคุณภาพอย่างเป็นกลาง

เมตริกคุณภาพที่สำคัญ (Key Quality Metrics)

1. ความหนาแน่นของข้อบกพร่อง (Defect Density) มีบั๊ก (Bugs) กี่ตัวต่อหนึ่งหน่วยของโค้ด?

สูตร (Formula):

Defect Density = จำนวนข้อบกพร่อง / ขนาดของซอฟต์แวร์

โดยที่ ขนาดสามารถเป็น:

- บรรทัดของโค้ด (Lines of Code - LOC)
- ฟังก์ชันพอยต์ (Function Points - FP)
- สตอรี่พอยต์ (Story Points)

ตัวอย่าง: ระบบห้องสมุด (Library System)

- ข้อบกพร่องทั้งหมดที่พบ: 45
- บรรทัดของโค้ด: 15,000
- Defect Density = 45 / 15,000 = 0.003
  ข้อบกพร่อง/LOC = 3 ข้อบกพร่อง/KLOC

เกณฑ์มาตรฐานของอุตสาหกรรม (Industry Benchmarks):

- 0-1 ข้อบกพร่อง/KLOC = ยอดเยี่ยม (Excellent)
- 1-5 ข้อบกพร่อง/KLOC = ดี (Good)
- 5-10 ข้อบกพร่อง/KLOC = ปานกลาง (Average)
- 10+ ข้อบกพร่อง/KLOC = ไม่ดี (Poor)

สิ่งนี้บอกอะไรกับเรา (What it tells us):

- ยิ่งต่ำยิ่งดี
- เปรียบเทียบในทั้งโมดูลเพื่อค้นหาพื้นที่ที่มีปัญหา
- ติดตามเมื่อเวลาผ่านไปเพื่อดูความปรับปรุง

---

2. **ความครอบคลุมของการทดสอบ (Test Coverage)** มีการทดสอบโค้ดเท่าไหร่?

ประเภทของความครอบคลุม (Types of Coverage):

ก) ความครอบคลุมของคำสั่ง (Statement Coverage)

```
Statement Coverage = (คำสั่งที่ดำเนินการแล้ว / คำสั่งทั้งหมด) × 100%
```

ข) ความครอบคลุมของกิ่ง (Branch Coverage)

```
Branch Coverage = (กิ่งที่ดำเนินการแล้ว / กิ่งทั้งหมด) × 100%
```

ค) ความครอบคลุมของฟังก์ชัน (Function Coverage)

```
Function Coverage = (ฟังก์ชันที่ดำเนินการแล้ว / ฟังก์ชันทั้งหมด) × 100%
```

ตัวอย่าง: ระบบห้องสมุด (Library System)

```
function calculateFine(daysLate) {
  if (daysLate <= 0) {
    // Branch 1
    return 0;
  } else if (daysLate <= 7) {
    // Branch 2
    return daysLate * 5;
  } else {
    // Branch 3
    return 35 + (daysLate - 7) * 10;
  }
}
```

กรณีทดสอบ (Test Cases):

```
// ทดสอบ 1: ตรงเวลา (On time)
calculateFine(0); // ครอบคลุม Branch 1 ✓

// ทดสอบ 2: ล่าช้า แต่ในสัปดาห์แรก (Late but within first week)
calculateFine(3); // ครอบคลุม Branch 2 ✓

// ทดสอบ 3: ล่าช้ามากกว่าสัปดาห์ (Late more than a week)
calculateFine(10); // ครอบคลุม Branch 3 ✓
```

ความครอบคลุมที่บรรลุ (Coverage Achieved):

- Statement Coverage: 100% (ดำเนินการบรรทัดทั้งหมด)
- Branch Coverage: 100% (ใช้กิ่งทั้งหมด)
- Function Coverage: 100% (เรียกใช้ฟังก์ชัน)

มาตรฐานของอุตสาหกรรม (Industry Standards):

- Unit Tests (การทดสอบหน่วย): เป้าหมายความครอบคลุม 80-90%
- Integration Tests (การทดสอบการรวม): เป้าหมายความครอบคลุม 70-80%
- E2E Tests (การทดสอบแบบปลายต่อปลาย): เน้นเส้นทางที่สำคัญ (ไม่ใช่เปอร์เซนต์ความครอบคลุม)

สิ่งสำคัญ (Important): ความครอบคลุม 100% ≠ ไม่มีบั๊ก!

นั่นหมายความว่าโค้ดถูกดำเนินการทดสอบแล้ว ไม่ได้หมายความว่าได้ทดสอบอย่างถูกต้อง

The Coverage Paradox: 100% Coverage ≠ Bug-Free Code

ตัวอย่างจริง (Real Example):

```
// ฟังก์ชัน: การหารตัวเลข
function divide(a, b) {
  return a / b;  // บรรทัดเดียว แต่มีหลาย edge cases
}

// Test case ที่เขียน
test('divide should calculate 4 / 2', () => {
  expect(divide(4, 2)).toBe(2);  // ✓ Pass
});
```

ผลลัพธ์ (Results):

- ✅ Coverage: 100% (บรรทัดทั้งหมดดำเนินการแล้ว)
- ✅ Test Pass Rate: 100%
- ❌ **แต่ bugs ยังอยู่!**

```
divide(5, 0);      // Infinity ❌
divide(0, 0);      // NaN ❌
divide(-5, 2);     // -2.5 (อาจต้องเข้ารหัส?) ❌
divide(5, null);   // NaN ❌
```

บทเรียน (Lesson):

- Coverage วัดเพียง "โค้ดถูกดำเนินการหรือไม่"
- Coverage **ไม่วัด** "โค้ดทำสิ่งที่ถูกต้องหรือไม่"
- ต้องผสมผสาน: Coverage + Test Logic Quality

วิธีแก้ (Solution):

```
// ทดสอบ edge cases อย่างเหมาะสม
describe('divide function', () => {
  test('should divide positive numbers correctly', () => {
    expect(divide(4, 2)).toBe(2);
    expect(divide(10, 4)).toBe(2.5);
  });

  test('should handle zero dividend', () => {
    expect(divide(0, 5)).toBe(0);
  });

  test('should return Infinity when dividing by zero', () => {
    expect(divide(5, 0)).toBe(Infinity);
  });

  test('should handle negative numbers', () => {
    expect(divide(-10, 2)).toBe(-5);
    expect(divide(10, -2)).toBe(-5);
  });

  test('should validate inputs', () => {
    expect(() => divide(5, null)).toThrow();
    expect(() => divide(undefined, 2)).toThrow();
  });
});
```

---

3. **ประสิทธิภาพการกำจัดข้อบกพร่อง (Defect Removal Efficiency - DRE)** ประสิทธิผลของการทดสอบในการค้นหาบั๊กเพียงใด?

สูตร (Formula):

DRE = (ข้อบกพร่องที่พบก่อนการปล่อยตัว / ข้อบกพร่องทั้งหมด) × 100%

โดยที่:
ข้อบกพร่องทั้งหมด = ข้อบกพร่องที่พบก่อนการปล่อยตัว + ข้อบกพร่องที่พบหลังการปล่อยตัว

ตัวอย่าง: ระบบห้องสมุด (Library System)

- ข้อบกพร่องที่พบในการทดสอบ: 45
- ข้อบกพร่องที่พบโดยผู้ใช้ (3 เดือนแรก): 5
- ข้อบกพร่องทั้งหมด: 45 + 5 = 50
- DRE = (45 / 50) × 100% = 90%

เกณฑ์มาตรฐานของอุตสาหกรรม (Industry Benchmarks):

- 85-95% = ดี (Good)
- 95-99% = ยอดเยี่ยม (Excellent)
- < 85% = ต้องการปรับปรุง (Needs Improvement)

สิ่งนี้บอกอะไรกับเรา (What it tells us):

- ประสิทธิผลของกระบวนการทดสอบของเรา
- DRE ที่สูงกว่า = ข้อบกพร่องน้อยกว่าถึงผู้ใช้
- ติดตามเมื่อเวลาผ่านไปเพื่อดูความปรับปรุง

---

4. อัตราการปฏิบัติการทดสอบ (Test Execution Rate) ทำการทดสอบได้เร็วแค่ไหน?

สูตร (Formula):

```
Test Execution Rate = จำนวนกรณีทดสอบที่ดำเนินการ / ช่วงเวลา
```

ตัวอย่าง (Example):

- กรณีทดสอบ 500 กรณีดำเนินการใน 5 วัน
- Test Execution Rate = 100 กรณีทดสอบ/วัน

มีประโยชน์สำหรับ (Useful for):

- การวางแผนตารางการทดสอบ
- ระบุคอขวดการทำงาน (Bottlenecks)
- ประมาณการกำหนดเวลาโครงการ

---

5. อัตราการผ่าน/ล้มเหลว (Pass/Fail Rate) ผ่านการทดสอบกี่เปอร์เซนต์?

สูตร (Formula):

```
Pass Rate = (ทดสอบที่ผ่าน / ทดสอบทั้งหมดที่ดำเนินการ) × 100%
```

ตัวอย่าง: ระบบห้องสมุด Sprint 3 (Library System Sprint 3)

- ทดสอบทั้งหมด: 200
- ผ่าน: 180
- ล้มเหลว: 15
- ถูกบล็อก: 5
- Pass Rate = (180 / 200) × 100% = 90%

การตีความ (Interpretation):

- 95-100% = ระบบมีเสถียรภาพและพร้อมปล่อยตัว
- 85-95% = มีปัญหาบ้าง, อาจต้องทำงานเพิ่มเติม
- < 85% = ปัญหาที่มีนัยสำคัญ, ยังไม่พร้อมปล่อยตัว

---

6. เวลาเฉลี่ยในการตรวจจับ (Mean Time To Detect - MTTD) ค้นหาบั๊กใช้เวลานานเท่าไหร่?

สูตร (Formula):

```
MTTD = เวลาทั้งหมดจากนำเข้าบั๊กถึงการตรวจจับ / จำนวนบั๊ก
```

ตัวอย่าง (Example):

- นำเข้าบั๊กในวันที่ 1
- ตรวจจับบั๊กในวันที่ 5
- MTTD = 4 วัน

ยิ่งต่ำยิ่งดี - บ่งชี้ถึงการปฏิบัติการทดสอบที่มีประสิทธิผล

---

7. เวลาเฉลี่ยในการซ่อมแซม (Mean Time To Repair - MTTR) แก้ไขบั๊กใช้เวลานานเท่าไหร่?

สูตร (Formula):

```
MTTR = เวลาทั้งหมดจากการตรวจจับบั๊กถึงการแก้ไข / จำนวนบั๊ก
```

ตัวอย่าง (Example):

- รายงานบั๊กในวันจันทร์ เวลา 9 โมง
- แก้ไขและตรวจสอบบั๊กในวันอังคาร เวลา 14.00 น.
- MTTR = 29 ชั่วโมง

ค่าเฉลี่ยของอุตสาหกรรม (Industry Average): 24-48 ชั่วโมงสำหรับบั๊กที่สำคัญ

ส่วนที่ 4: การคืนทุนของการทดสอบ (Return on Investment (ROI) of Testing)

ต้นทุนของคุณภาพ (The Cost of Quality)

กรอบการคิดต้นทุนของคุณภาพ (Cost of Quality Framework):

1. **ต้นทุนการป้องกัน (Prevention Costs)** - เชิงรุก (Proactive)
    - การฝึกสอนผู้ทดสอบ
    - การตรวจสอบโค้ด (Code Reviews)
    - เครื่องมือการทดสอบอัตโนมัติ
    - การวางแผนคุณภาพ
2. **ต้นทุนการประเมิน (Appraisal Costs)** - การตรวจจับ (Detection)
    - การปฏิบัติการทดสอบ
    - การตั้งค่าสภาพแวดล้อมการทดสอบ
    - การประชุมตรวจสอบข้อบกพร่อง (Bug Triage Meetings)
    - การตรวจสอบคุณภาพ (Quality Audits)
3. **ต้นทุนความล้มเหลวภายใน (Internal Failure Costs)** - พบก่อนการปล่อยตัว (Found before release)
    - การแก้จุดบกพร่องและแก้ไขข้อบกพร่อง
    - การทดสอบอีกครั้ง (Re-testing)
    - สูญเสียประสิทธิผลของการทำงาน
    - การบำรุงรักษากรณีทดสอบ

4. **ต้นทุนความล้มเหลวภายนอก (External Failure Costs)** - พบหลังการปล่อยตัว (Found after release)
    - การสนับสนุนลูกค้า
    - แพตช์ฉุกเฉิน
    - สูญเสียชื่อเสียง
    - ความรับผิดชอบทางกฎหมาย
    - สูญเสียลูกค้า

กฎ 1-10-100 (The 1-10-100 Rule):

- $1 เพื่อแก้ไขในการพัฒนา (Development)
- $10 เพื่อแก้ไขในการทดสอบ (Testing)
- $100 เพื่อแก้ไขในการผลิต (Production)

การคำนวณการคืนทุนของการทดสอบ (Calculating ROI of Testing)

สูตร ROI พื้นฐาน (Basic ROI Formula):

```
ROI = (ประโยชน์ - ต้นทุน) / ต้นทุน × 100%
```

ตัวอย่าง: ระบบห้องสมุด (Library Management)

สถานการณ์ (Scenario): เราควรลงทุนในการทดสอบอัตโนมัติหรือไม่?

ต้นทุน (Costs):

- เครื่องมือการทดสอบอัตโนมัติ: $0 (ใช้ Playwright/Jest - ฟรี)
- เวลาการฝึกสอน: 40 ชั่วโมง × $50/ชั่วโมง = $2,000
- การพัฒนาสคริปต์เริ่มต้น: 160 ชั่วโมง × $50/ชั่วโมง = $8,000
- **ลงทุนทั้งหมด: $10,000**

ประโยชน์ (Benefits) (ต่อปี):

- เวลาการทดสอบด้วยตนเองที่หยุด: 400 ชั่วโมง × $50/ชั่วโมง = $20,000
- บั๊กที่จับได้เร็ว (Shift Left): ป้องกันบั๊ก 10 ตัวในการผลิต
    - ต้นทุนต่อบั๊กในการผลิต: $2,000
    - ประหยัด: 10 × $2,000 = $20,000
- ความเร็วรอบการผลิตที่เร็วขึ้น: เพิ่มปล่อยตัว 2 ครั้ง/ปี
    - รายได้ต่อปล่อยตัว: $15,000
    - รายได้พิเศษ: 2 × $15,000 = $30,000
- **ประโยชน์ทั้งหมด: $70,000**

การคำนวณ ROI (ROI Calculation):

```
ROI = ($70,000 - $10,000) / $10,000 × 100% = 600%
```

ระยะเวลาคืนทุน (Payback Period):

```
Payback = $10,000 / $70,000 ต่อปี = 0.14 ปี ≈ 1.7 เดือน
```

สรุป: เป็นการลงทุนที่ยอดเยี่ยม! คืนทุนในเวลาน้อยกว่า 2 เดือน

ทำไมการทดสอบจึงไม่ใช่ทางเลือก (Why Testing is NOT Optional)

ผลการวิจัย (Research Findings):

1. **การศึกษาของ IBM**: บั๊กที่พบใน production มีค่าใช้จ่ายสูงกว่า 15 เท่าต่อบั๊กที่พบในการพัฒนา
2. **การวิจัยของ Capers Jones**:
    - โครงการที่มี DRE < 85%: เกินงบประมาณ 50%
    - โครงการที่มี DRE > 95%: ตรงตามงบประมาณหรือต่ำกว่า
3. **ต้นทุนของการขาดคุณภาพ** (Cost of Poor Quality):
    - โครงการซอฟต์แวร์เฉลี่ย: 25-40% ของงบประมาณใช้ในการแก้ไขทำซ้ำ (Rework)
    - ด้วยการทดสอบที่ดี: 5-10% ใช้ในการแก้ไขทำซ้ำ

ข้อมูลสรุป (Bottom Line):

การทดสอบคือการลงทุน ไม่ใช่ต้นทุน

ข้ามการทดสอบ = จ่ายมากขึ้นในภายหลัง

Quality vs Cost Trade-offs: **เมื่อต้องสมดุลคุณภาพและต้นทุน** (When to Compromise?)

ความจริงในปัจจุบัน (Real-World Truth):

"ไม่มีโครงการไหนมีทรัพยากรอนันต์ เราต้องตัดสินใจอย่างชาญฉลาด"

สถานการณ์ที่ 1: Startup MVP (Minimum Viable Product)

สถานการณ์ (Scenario):

- ทรัพยากร: น้อย (1-2 ผู้พัฒนา, 2-4 สัปดาห์)
- ความต้องการ: รีลีสเร็วเข้าตลาด
- ความสำคัญ: ความเป็นไปได้ (Feasibility) มากกว่าคุณภาพสูง

Strategy:

- ✅ โฟกัส: Black Box testing + Manual testing เท่านั้น
- ✅ ข้าม: White Box, Performance testing, Full security testing
- ✅ Coverage: 50-70% ก็พอ (เน้นเส้นทางสำคัญ)
- ✅ DRE: 70-80% ยอมรับได้
- ❌ ไม่ต้อง: CI/CD pipelines, Load testing, Security scanning

ตัวอย่าง (Example):

MVP Library System:

- 2 สัปดาห์ development

- 2 วัน testing เท่านั้น

- Bug จำนวนมากยอมรับได้ (จะ fix ใน v1.1)

- Metric: DRE 75%, Coverage 60%

- ผลลัพธ์: บ่อนได้เข้าตลาด ได้ฟีดแบ็ก ทำให้ดีขึ้น

สถานการณ์ที่ 2: Enterprise System

สถานการณ์ (Scenario):

- ทรัพยากร: มาก (Team 10+ คน, ระยะยาว)
- ความต้องการ: ระบบต้องเชื่อถือได้ ดำเนินการ 24/7
- ความสำคัญ: คุณภาพเป็นสิ่งสำคัญสูงสุด

Strategy:

- ✅ โฟกัส: Unit + Integration + E2E + Performance + Security
- ✅ Coverage: 85-95% เป้าหมาย
- ✅ DRE: 95%+
- ✅ ต้องมี: CI/CD, Load testing, Security scanning, Chaos engineering

ตัวอย่าง (Example):

```
Enterprise Banking System:
- 6+ เดือน development
- 2 เดือน testing + QA
- เล็กน้อยบั๊กยอมรับไม่ได้
- Metric: DRE 96%, Coverage 90%
- ผลลัพธ์: ระบบเสถียร ประหยัดจากบั๊ก production
```

สถานการณ์ที่ 3: Safety-Critical Systems

สถานการณ์ (Scenario):

- ความสำคัญ: บั๊ก 1 ตัว = หลายชีวิตหรือเงินหลายล้าน
- ตัวอย่าง: ยานยนต์, อากาศยาน, โรงพยาบาล, ธนาคาร

Strategy:

- ✅ Coverage: 100% statement + branch coverage
- ✅ DRE: 99%+
- ✅ Formal Verification, Stress Testing, Penetration Testing
- ✅ สอบเทียบและการรับรองมาตรฐาน

ตัวอย่าง (Example):

```
Medical Device Software:
- Strict FDA certification required
- 100% code review + formal testing
- DRE > 99.5%
- ต้นทุนทดสอบอาจ > ต้นทุนพัฒนา
- แต่ค่าของการบันทึกชีวิต = ไม่มีราคา
```

Decision Matrix: เลือก Quality Level อะไร?

Type of System	Coverage	DRE Target	Testing Time
MVP / Prototype	40-60%	60-75%	10% effort
Standard App	70-80%	80-90%	25% effort
Enterprise	85-95%	95-99%	40% effort
Safety-Critical	100%	99.5-99.9%	50%+ effort

เคล็ดลับการตัดสินใจ (Decision Tips):

1. ถาม: บั๊กมีต้นทุนเท่าไหร่?
2. ถาม: ต้องมีความพร้อมใช้งานเท่าไหร่?
3. ถาม: มีเรื่องความปลอดภัย/ความเป็นส่วนตัวหรือไม่?
4. ตัดสินใจ: ระดับคุณภาพที่เหมาะสม

ส่วนที่ 5: การสาธิตการคำนวณเมตริกในทางปฏิบัติ

สถานการณ์: ระบบห้องสมุดหลังจาก Sprint 3 (Scenario: Library System After Sprint 3)

ข้อมูลที่กำหนด (Given Data):

สถิติโครงการ (Project Stats):

- บรรทัดของโค้ด: 15,000
- ฟังก์ชัน: 350
- ขนาดทีม: ผู้พัฒนา 5 คน + ผู้ทดสอบ 2 คน

ผลการทดสอบ (Test Results):

- กรณีทดสอบทั้งหมด: 200
- ทดสอบที่ผ่าน: 180
- ทดสอบที่ล้มเหลว: 15
- ทดสอบที่ถูกบล็อก: 5

ข้อมูลข้อบกพร่อง (Defect Data):

- ข้อบกพร่องที่พบในการทดสอบหน่วย: 20
- ข้อบกพร่องที่พบในการทดสอบการรวม: 15
- ข้อบกพร่องที่พบในการทดสอบระบบ: 10
- ข้อบกพร่องที่พบในการทดสอบ UAT: 5
- ข้อบกพร่องที่พบโดยผู้ใช้ (เดือนแรก): 3
- ข้อบกพร่องทั้งหมด: 53

ความครอบคลุมของโค้ด (Code Coverage):

- ความครอบคลุมของคำสั่ง: 85%
- ความครอบคลุมของกิ่ง: 78%
- ความครอบคลุมของฟังก์ชัน: 90%

ตารางเวลา (Timeline):

- ระยะเวลา Sprint: 2 สัปดาห์
- เวลาการปฏิบัติการทดสอบทั้งหมด: 5 วัน

คำนวณเมตริกทั้งหมด (Let's Calculate All Metrics)

1. ความหนาแน่นของข้อบกพร่อง (Defect Density)

Defect Density = 53 ข้อบกพร่อง / 15 KLOC = 3.53 ข้อบกพร่อง/KLOC

การประเมิน: ดี (ในช่วง 1-5)

2. ความครอบคลุมของการทดสอบ (Test Coverage)

ความครอบคลุมเฉลี่ย = (85% + 78% + 90%) / 3 = 84.3%

การประเมิน: ดี ใกล้เป้า 85%

3. ประสิทธิภาพการกำจัดข้อบกพร่อง (Defect Removal Efficiency - DRE)

ข้อบกพร่องก่อนปล่อยตัว = 20 + 15 + 10 + 5 = 50

ข้อบกพร่องทั้งหมด = 50 + 3 = 53

DRE = (50 / 53) × 100% = 94.3%

การประเมิน: ยอดเยี่ยม! (> 90%)

4. อัตราการผ่านการทดสอบ (Test Pass Rate)

Pass Rate = (180 / 200) × 100% = 90%

การประเมิน: ดี ระบบมีเสถียรภาพค่อนข้างมาก

5. อัตราการปฏิบัติการทดสอบ (Test Execution Rate)

Execution Rate = 200 ทดสอบ / 5 วัน = 40 ทดสอบ/วัน

6. ข้อบกพร่องตามระดับการทดสอบ (Defects by Test Level)

การทดสอบหน่วย:	20 ข้อบกพร่อง (38%)
การทดสอบการรวม:	15 ข้อบกพร่อง (28%)
การทดสอบระบบ:	10 ข้อบกพร่อง (19%)
UAT:	5 ข้อบกพร่อง (9%)
Production:	3 ข้อบกพร่อง (6%)

นี่คือการกระจายที่ดี! จับส่วนใหญ่ของข้อบกพร่องในช่วงแรก

สรุป

- **ความหนาแน่นของข้อบกพร่อง:** 3.53 ข้อบกพร่อง/KLOC
- **ความครอบคลุมของการทดสอบ:** 84.3%
- **DRE:** 94.3%
- **อัตราการผ่าน:** 90%
- **อัตราการปฏิบัติการทดสอบ:** 40 ทดสอบ/วัน

→ **การประเมินรวม: พร้อมปล่อยตัวได้** ✓

Metrics Anti-patterns: การใช้เมตริกแบบผิด (Common Misuses of Metrics)

Anti-pattern #1: Chasing Vanity Metrics

ปัญหา (Problem):

> QA Manager: "เราต้องได้ 100% coverage!"
>
> Developer: "ฉันเขียน test ที่ไม่สำคัญเพื่อให้ coverage เพิ่ม"

ผลลัพธ์ (Result):

- ✓ Coverage: 100% (ตามตัวชี้วัด!)
- ✗ Quality: ลดลง (test ไม่มีความหมาย)
- ✗ Maintenance: Tougher (test ไม่ดี)

ตัวอย่าง (Real Example):

```
// Developer เพิ่ม coverage โดยทดสอบ getter/setter ที่ไม่สำคัญ
class User {
  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
}

// Test (ไร้ความหมาย)
test('getName returns name', () => {
  const user = new User('John');
  expect(user.getName()).toBe('John');  // Trivial test
});
```

วิธีแก้ (Solution):

- ✓ ตั้งเป้า coverage ที่สมเหตุสมผล (70-85%)
- ✓ โฟกัส: Test important logic ไม่ใช่ทั้งหมด
- ✓ Code review: ตรวจสอบคุณภาพของ test ไม่ใช่เฉพาะจำนวน

Anti-pattern #2: Gaming the Metrics

ปัญหา (Problem):

> Manager: "DRE ต้องเป็น 95% ขึ้นไป!"
>
> Tester: เจตนา inject bugs ไป แล้ว find มัน ⟶ DRE ดูสูง

ผลลัพธ์ (Result):

- ✓ DRE: 95% (ตามเป้า!)
- ✗ Actual quality: ไม่รู้จริง ๆ ว่าดีหรือไม่
- ✗ Trust: เสียศรัทธา

วิธีแก้ (Solution):

- ✓ วัด: Bugs found by QA vs Bugs found by users (ควร 80:20 ขึ้นไป)
- ✓ Monitor: Defect escape rate เมื่อเวลาผ่านไป
- ✓ Focus: ปรับปรุง process ไม่ใช่เพียง metric

Anti-pattern #3: Misinterpreting Single Metrics

ปัญหา (Problem):

นักเรียน: "Pass Rate 95% = โปรแกรมดี!"

แต่ความจริง: Test 100 ข้อ แต่มี 90 ข้อเป็น trivial happy path

ตัวอย่าง (Example):

```
// Happy path tests เท่านั้น (95% pass rate)
test('add should add two numbers', () => {
  expect(add(2, 3)).toBe(5);  // ✓ Pass
  expect(add(10, 20)).toBe(30); // ✓ Pass
});

// ไม่มี edge cases
// add(-5, null) ⟶ ?
// add(undefined, undefined) ⟶ ?
```

วิธีแก้ (Solution):

- ✓ ดูหลาย metrics พร้อมกัน (triangle: Coverage + DRE + Pass Rate)
- ✓ ดูการกระจายของ test (positive, negative, edge cases)
- ✓ ถาม: "ระบบนี้พร้อม release หรือไม่?" ไม่ใช่เฉพาะ "metrics ดีหรือไม่"

Anti-pattern #4: Over-relying on One Metric

ปัญหา (Problem):

Coverage = 100% ⟶ "โอเค release ได้!"

ผลลัพธ์: โปรแกรมขัดข้องในการผลิต (Edge cases ไม่ได้ทดสอบ)

ตัวอย่าง (Example):

```
function calculateTotal(items) {
  let total = 0;
  for (let item of items) {
    total += item.price;  // Covered ✓
```

```
  }
  return total;
}

// 100% Statement Coverage ด้วยการทดสอบนี้:
test('calculateTotal works', () => {
  const items = [{ price: 10 }, { price: 20 }];
  expect(calculateTotal(items)).toBe(30);  // ✓ 100% coverage
});

// แต่ไม่ได้ทดสอบ:
calculateTotal(null);          // Crash
calculateTotal([]);            // Edge case
calculateTotal([{ price: '10' }]);  // Type error
```

วิธีแก้ (Solution):

- ✓ ใช้ **Metrics Triangle**: Coverage + DRE + Test Quality
- ✓ ดูเทรนด์เมื่อเวลาผ่านไป ไม่ใช่ snapshot
- ✓ รวมกับ: Manual testing, Code review, User feedback

❌ **Anti-pattern #5: Setting Wrong Targets**

ปัญหา (Problem):

```
Manager: "ทีม B ได้ 90% coverage เราต้องได้ 95%!"
ความจริง: ทีม B ทำสิ่งที่เหมาะสม เราต้อง 70% ก็พอ
```

วิธีแก้ (Solution):

✓ ตั้งเป้า: ตามประเภทระบบไม่ใช่เปรียบเทียบ

✓ ดูประวัติ: "ผ่านมาเราเป็น 70% bugs นี้ระดับไหน?" ⟶ เป้า 75%

Quick Checklist: **เมตริกถูกต้องหรือไม่?**

✓ Metric มี actionable insights?

✓ Metric ตอบคำถามธุรกิจ?

✓ Metric ไม่สามารถ game ได้ง่าย?

✓ ดู multiple metrics ด้วยกัน?

✓ Trend ดีขึ้นหรือแย่ลง?

✓ เทียบกับ Baseline ตรรมชาติไม่ใช่อุตสาหกรรม?

เชื่อมโยงเมตริกกับเทคนิคการทดสอบ (How Metrics Guide Testing Strategy)

"เมตริก = แผนที่บอกคุณว่าต้องทดสอบอะไรและทดสอบแค่ไหน"

ISO 25010 ⟶ Test Design Strategy

ISO 25010 Attribute	ลักษณะย่อย	Testing Strategy	Key Metrics
Functionality	ความครบถ้วน	Black Box (EP, BVA, DT)	Defect Density
Performance	เวลาตอบสนอง	Load/Stress Testing	Response Time, Throughput
Compatibility	Interoperability	Integration Testing	Integration Success Rate
Usability	Learnability	Usability Testing	User Error Rate, SUS Score
Reliability	MTBF	Reliability Testing	Mean Time Between Failures
Security	Confidentiality	Security Testing	Vulnerabilities Found
Maintainability	Analyzability	White Box, Code Review	Cyclomatic Complexity
Portability	Adaptability	Cross-Platform Testing	Platform Compatibility %

Coverage Metrics ⟶ Test Level Selection

Unit Tests (White Box):
- ├── Coverage Target: 80-90%
- ├── Focus: Statement + Branch coverage
- ├── Tools: Jest, pytest
- └── Metrics: Function coverage, Line coverage

Integration Tests (Gray Box):
- ├── Coverage Target: 70-80%
- ├── Focus: Module interactions, API contracts
- ├── Tools: supertest, Postman
- └── Metrics: Module integration success %

E2E Tests (Black Box):
- ├── Coverage Target: 60-70% (critical paths)
- ├── Focus: User workflows, business scenarios
- ├── Tools: Playwright, Selenium
- └── Metrics: End-to-end success rate

DRE ⟶ When to Stop Testing

DRE Lifecycle ในโปรเจกต์ (Project Timeline):

Week 1-2 (Early Testing):
- ├── DRE: 20-30% (ยังหา bugs เยอะ)
- ├── Action: หา defects ให้เร็ว ⟶ Unit + Integration tests

└── Goal: ล่าสุดไป Functionality basics

Week 3-4 (Mid Testing):

├── DRE: 60-70% (ได้ majority bugs)

├── Action: หาข้อมูล edge cases ⟶ Black Box + exploratory

└── Goal: ทดสอบทั้งระบบ

Week 5-6 (Final Testing):

├── DRE: 85-95% (ยากหา bugs ที่เหลือ)

├── Action: Exploratory, stress testing, security

└── Goal: ค้นหา bugs ขั้นสูง

Release Readiness:

└── DRE ควร: 90-95%+ ตามประเภทระบบ

ตัดสินใจ "ยอมรับได้":

ถ้า DRE ≥ 90% + Pass Rate ≥ 95% + Coverage ≥ 80%

⟶ พร้อม Release ได้ (แต่ต้องดู Defect profile)

Black Box vs White Box: เลือกแบบไหน?

ตัดสินใจโดยใช้เมตริก:

เลือก Black Box Testing ถ้า:

├── Coverage ยังต่ำ < 60% (ต้อง cover basic functionality)

├── DRE ต่ำ < 70% (ยังหา bugs ได้เยอะ)

└── User-facing features (ต้องทดสอบ user perspective)

เลือก White Box Testing ถ้า:

├── Coverage ดี > 80% แล้ว (หาส่วนที่ไม่ cover)

├── Complex logic (if-else, loops อเยอะ)

└── Cyclomatic complexity สูง (ต้องทดสอบ paths ทั้งหมด)

หัวข้อถัดๆไป (Upcoming Topics) กับเมตริก

Week 3: Test Planning

└── ใช้: Defect Density targets ⟶ กำหนด test scope

Week 4-5: Black Box Testing

└── ใช้: Equivalence Partitioning ⟶ ลดจำนวน test cases

(ผสม 50 EP = ผสม 1000 random tests)

Week 6: Static Testing

└── ใช้: Code coverage + Cyclomatic complexity

⟶ ตัดสินใจ code areas ไหนต้อง code review

Week 7: White Box Testing

└── ใช้: Branch coverage ⟶ แน่ใจว่า covered ทั้งหมด

Week 8+: Automation

└── ใช้: Test execution rate, automation ROI

⟶ ตัดสินใจ automate ไหน

Quick Reference: ISO 25010 & Metrics at a Glance

ISO 25010 Attributes Cheat Sheet

#	Attribute	Definition	Simple Check
1	Functionality	ทำสิ่งที่ต้องการได้?	Test all features
2	Performance	ทำเร็วพอหรือไม่?	< 2 sec response time
3	Compatibility	ทำงานกับระบบอื่นได้?	Test integration, cross-platform
4	Usability	ใช้งานง่ายไหม?	Usability testing, WCAG check
5	Reliability	ทำงานถูกต้องเสมอ?	Stability, MTBF
6	Security	ปลอดภัยไหม?	Penetration test, OWASP check
7	Maintainability	แก้ไขง่ายไหม?	Code review, complexity analysis
8	Portability	ติดตั้งบนระบบอื่นได้?	Cross-OS testing, Docker

Metrics Decision Table

Metric	Target	What it means	Red Flag
Defect Density	1-5 /KLOC	Few bugs per 1000 LOC	> 10 /KLOC
Coverage	80-90%	Most code tested	< 60%
DRE	90-95%	Find most bugs before release	< 85%
Pass Rate	95%+	Most tests pass	< 85%
MTBF	> 30 days	System stable	< 7 days
Response Time	< 2 sec	Fast enough	> 5 sec

Metrics by Project Type

MVP / Startup:

├── Coverage: 40-60% (Quick to market)

├── DRE: 60-75% (Some bugs OK)

├── Timeline: 10% testing effort

└── Tools: Manual + basic automation

Standard Application:

├── Coverage: 70-85%

├── DRE: 80-90%

├── Timeline: 25% testing effort

└── Tools: Unit + Integration + E2E

Enterprise System:

├── Coverage: 85-95%

├── DRE: 95-99%

├── Timeline: 40% testing effort

└── Tools: Full test pyramid + performance

Safety-Critical:

├── Coverage: 100%

├── DRE: 99.5%+

├── Timeline: 50%+ testing effort

└── Tools: Formal verification + certification

ประเด็นสำคัญที่ต้องจำ (Key Takeaways)

นิยามคุณภาพ (Quality Definition)

- คุณภาพ = การตรงตามข้อกำหนด + ตรงตามความคาดหวังของผู้ใช้
- มุมมองหลายด้าน: ผู้ใช้, ผู้พัฒนา, ธุรกิจ

ISO 25010

- ลักษณะคุณภาพ 8 ประการให้กรอบการทำงานที่ครอบคลุม
- แต่ละลักษณะมีลักษณะย่อย
- ใช้เป็นรายการตรวจสอบสำหรับการประเมินคุณภาพ

เมตริกคุณภาพ (Quality Metrics)

- เมตริกทำให้คุณภาพเป็นปรนัย (Objective) และวัดได้
- เมตริกหลัก: Defect Density, Coverage, DRE

- ติดตามเมตริกเมื่อเวลาผ่านไปเพื่อดูแนวโน้ม
- ใช้เมตริกเพื่อตัดสินใจ ไม่ใช่เพื่อรายงานเท่านั้น

การคืนทุนของการทดสอบ (ROI of Testing)

- การทดสอบคือการลงทุน ไม่ใช่ต้นทุน
- กฎ 1-10-100: แก้ไขเร็ว ๆ ช่วยประหยัดเงิน
- คำนวณ ROI เพื่อปรับการลงทุนด้านการทดสอบ
- ซอฟต์แวร์ที่มีคุณภาพสูง = ต้นทุนรวมต่ำ

ความผิดพลาดทั่วไป (Common Mistakes Students Often Make)

ความผิดพลาด #1: เปลี่ยนเมตริกเยอะ = คุณภาพดี

สิ่งที่นิสิตคิด:

> "Defect Density 3.5, Coverage 85%, DRE 94%, Pass Rate 90% ⟶ โปรแกรมดี!"

ความจริง:

✓ Metrics นั่นดี แต่...

✗ Test cases อาจเป็น trivial happy path

✗ Edge cases อาจไม่ได้ทดสอบ

✗ Performance, Security อาจยังไม่ดี

วิธีแก้ (Fix):

> ดูเมตริก 3 ตัวนี้รวมกัน:
> 1. Coverage (broad): ครอบคลุมโค้ดเพียงพอหรือไม่?
> 2. DRE (deep): หา bugs ได้เยอะแค่ไหน?
> 3. Pass Rate (quality): Defects ที่พบดีแค่ไหน?

ถ้าทั้ง 3 อย่างดี = พร้อม Release

ความผิดพลาด #2: นับโค้ด LOC (Lines of Code) ไม่ถูก

ปัญหาที่เห็น:

```
// เหล่านี้ ควรนับ 1 บรรทัด หรือ 5 บรรทัด?


// แบบ 1 บรรทัด
const total = items.reduce((sum, item) => sum + item.price, 0);


// แบบ 5 บรรทัด
let total = 0;
for (let item of items) {
  total += item.price;
}
```

วิธีที่ถูก (Correct Way):

- ใช้ Automatic tools (ESLint, SonarQube)
- ไม่นับ: Comments, Blank lines
- นับ: Physical lines ของ source code

```
// ใช้ npm library
const sloc = require('sloc');
sloc.countFile('app.js', (result) => {
  console.log(`Total LOC: ${result.total}`);
});
```

ความผิดพลาด #3: คิดว่า Pass Rate 100% = โปรแกรมสมบูรณ์

สิ่งที่เห็นจริง:

```
Pass Rate 100% (30/30 ผ่าน) VS DRE 60%
⟶ อันตราย! Defects ยังเยอะในการผลิต
```

เหตุผล:

- Test cases อาจเขียนแย่
- Happy path tests เท่านั้น
- Edge cases ไม่ได้ทดสอบ
-

ตัวอย่าง (Example):

```
// Test ที่ "ผ่าน" แต่ไม่ดี
test('process payment', () => {
  const result = processPayment(100);
  expect(result).toBe(true);      // ✓ Pass
});

// แต่ไม่ได้ทดสอบ:
processPayment(0);              // Edge case
processPayment(-100);           // Invalid
processPayment(null);           // Type error
processPayment('abc');          // Type error
```

วิธีแก้:

```
ดูการกระจายของ test:
- Positive cases (Happy path): 30%
- Negative cases (Invalid): 40%
- Edge cases (Boundary): 30%
```

ดูประเภท bugs ที่พบ:

- Functional: 60%
- Security: 20%
- Performance: 20%

ความผิดพลาด #4: ใช้ metric เดียวตัดสินใจ

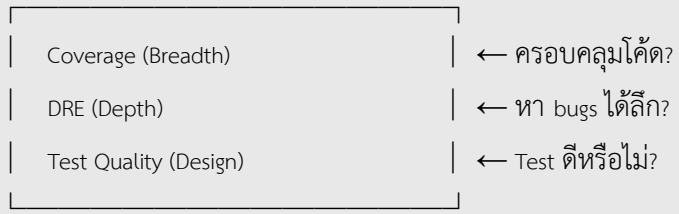
ตัวอย่าง (Example):

นิสิต: "Coverage 100% ⟶ Ready to release!" ✗

จริงๆ: "Coverage 100% but only happy paths ⟶ Many bugs in production"

คำตอบที่ถูก:

Metrics ต้องดู 3 มิติ (3D):



ทั้ง 3 ตัวต้องดี ถึงปลอดภัย

ความผิดพลาด #5: นำเมตริก "ทั่วไป" มาใช้ทั้งหมด

ปัญหา:

ทีม A (Startup):	ทีม B (Enterprise):
Coverage 50%	Coverage 90%
DRE 70%	DRE 96%
โอเค ✓	โอเค ✓

เทียบกัน: ทีม B ดีกว่า ✗ (ไม่ใช่!)

ความจริง:

- Startup MVP: 50% coverage พอ (ต้องเร็ว)
- Enterprise: 90% coverage ต้อง (ต้องเชื่อถือ)

วิธีแก้:

ตั้งเป้า metric ตามประเภทระบบ:

MVP:	Coverage 40-60%,	DRE 60-75%
Standard:	Coverage 70-85%,	DRE 85-90%
Enterprise:	Coverage 85-95%,	DRE 95-99%
Critical:	Coverage 100%,	DRE 99.5%+

ความผิดพลาด #6: ไม่ติดตาม Trend

เห็นแค่:

This Sprint: DRE = 92%

"ดี!" ✓

ต้องดูด้วย:

Sprint 1: DRE = 92%

Sprint 2: DRE = 90% ⟵ แย่ลง! ⚠️

Sprint 3: DRE = 88% ⟵ แย่ลงต่อ! 🚨

⟶ ต้องค้นหาว่าเกิดอะไร

ติดตาม (Track):

- Excel chart หรือ Google Sheets
- Update ทุก sprint
- ดูแนวโน้ม (trend) ไม่ใช่เฉพาะตัวเลข

Quick Self-Check: **ทำถูกหรือผิด?**

1. "Coverage 100% ⟶ Product ready"
   ✗ ผิด (coverage ไม่ = quality)

2. "DRE 95% เป้าหมายสำหรับ MVP"
   ✗ ผิด (MVP ต้อง 70-75% พอ)

3. "Defect Density 0.5 = ดี"
   ? ไม่แน่ (ต้องดู: ประเภท, type, severity)

4. "Pass Rate 95% แล้ว release ได้"
   ? ไม่แน่ (ต้องดู: test quality, DRE, Coverage)

5. "เมตริกเพิ่มขึ้น ต้องเพิ่มการทดสอบ"
   ✓ ถูก (DRE ลดลง = ต้องทำอะไร)

Additional Resources

มาตรฐาน (Standards):

- ISO/IEC 25010:2011 - ความต้องการคุณภาพของระบบและซอฟต์แวร์ และการประเมิน (Systems and software Quality Requirements and Evaluation - SQuaRE)
- IEEE 730-2014 - กระบวนการ ธรรมชาติการประกันคุณภาพซอฟต์แวร์ (Software Quality Assurance Processes)

หนังสือ (Books):

- "Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement" โดย Jeff Tian
- "Managing the Testing Process" โดย Rex Black

บทความ (Articles):

- "The Economic Impacts of Inadequate Infrastructure for Software Testing" - รายงาน NIST
- "The ROI of Software Testing" - การศึกษาต่าง ๆ ของอุตสาหกรรม

คำถามสำหรับการอภิปราย (Discussion Questions)

1. ลักษณะ ISO 25010 ใดที่สำคัญที่สุดสำหรับระบบห้องสมุด? เพราะเหตุใด
2. หากต้องเลือกระหว่าง 100% ความครอบคลุมของโค้ดหรือ 95% DRE คุณจะเลือกอันไหน? เพราะเหตุใด
3. คุณจะโน้มน้าวการจัดการให้ลงทุนมากขึ้นในการทดสอบได้อย่างไร
4. คุณสามารถบรรลุคุณภาพสูงโดยไม่วัดได้หรือไม่
5. จะเกิดอะไรขึ้นหากคุณเน้นเฉพาะความเหมาะสมของฟังก์ชันการทำงานและละเลยลักษณะอื่น ๆ

แบบสำรวจการประเมินตนเอง (Self-Assessment Quiz)

1. ระบุลักษณะคุณภาพ ISO 25010 ทั้ง 8 ประการ
2. คำนวณ Defect Density หากคุณมีบั๊ก 30 ตัวในโค้ด 10,000 บรรทัด
3. DRE คืออะไรหากพบบั๊ก 40 ตัวในการทดสอบและ 5 ตัวในการผลิต
4. ความแตกต่างระหว่าง Statement Coverage และ Branch Coverage คืออะไร
5. อธิบายกฎ 1-10-100 ด้วยคำพูดของคุณเอง

คำตอบ (Answers):

1. ความเหมาะสมของฟังก์ชันการทำงาน, ประสิทธิภาพการใช้ทรัพยากร, ความเข้ากันได้, ความใช้งานง่าย, ความเชื่อถือได้, ความปลอดภัย, ความสามารถในการบำรุงรักษา, ความสามารถในการปรับตัว
2. 3 ข้อบกพร่อง/KLOC
3. 88.9%
4. Statement = บรรทัดที่ดำเนินการแล้ว; Branch = เส้นทางการตัดสินใจที่ดำเนินการแล้ว
5. บั๊กมีค่า $1 ในการพัฒนา, $10 ในการทดสอบ, $100 ในการผลิต