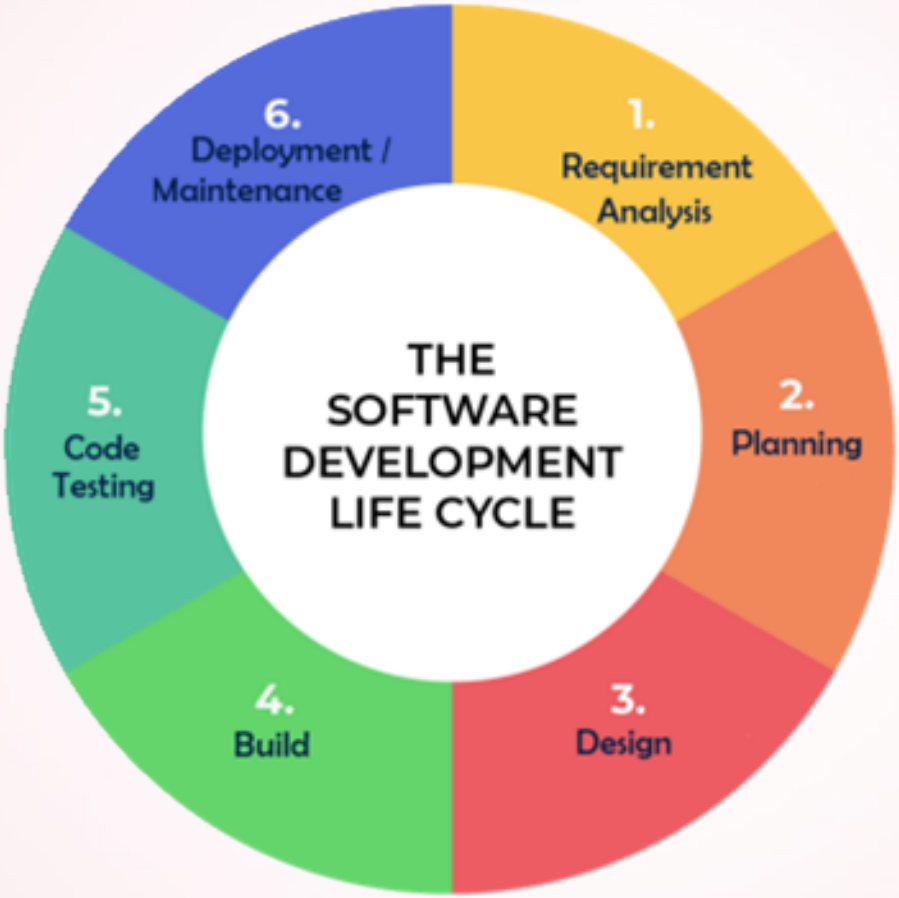
การทดสอบซอฟต์แวร์

จัดทำ Slide โดย ผศ.ดร.เมทินี เขียวกันยะ
เนื้อหาหลัก โดย ผศ.ดร.ฐิมาพร เพชรแก้ว

วัฏจักรของการพัฒนาซอฟต์แวร์



Credit รูป : Sunanda Karunajeewa

การทดสอบซอฟต์แวร์

- **การทดสอบซอฟต์แวร์ (Software Testing) หมายถึง กระบวนการในการตรวจสอบ และติดตามผลการพัฒนาซอฟต์แวร์ เพื่อให้แน่ใจว่าซอฟต์แวร์ที่จะส่งมอบงานมีความถูกต้อง สมบูรณ์ และมีประสิทธิภาพตามที่ผู้ใช้งานคาดหวังไว้**
- **การทดสอบซอฟต์แวร์มีความสำคัญสองประการ คือ เพื่อพิจารณาว่าการทำงานของระบบตรงกับข้อกำหนดคุณลักษณะระบบ (System specification) หรือข้อกำหนดความต้องการซอฟต์แวร์ (Software requirements specification) หรือไม่ และเพื่อพิจารณาถึงคุณภาพในการทำงานของระบบว่ายอมรับได้หรือไม่**

คำศัพท์ที่เกี่ยวข้อง

ข้อผิดพลาด (Error, Mistake หรือ Bug)

- คือความผิดพลาดที่เกิดจากคนเป็นผู้ทำให้เกิดขึ้น
  เช่น คนทำให้เกิดข้อผิดพลาดขึ้นในขณะที่เขียนโปรแกรม โดยเลือกใช้คำสั่งผิดหรือเขียนคำสั่งเรียงลำดับผิด
- ข้อผิดพลาดอาจเกิดขึ้นได้ตั้งแต่ขั้นตอนการรวบรวมความต้องการซอฟต์แวร์จากลูกค้า เนื่องจากลูกค้าบอกความต้องการของตนเองไม่ถูกต้อง ไม่ครบถ้วนหรือไม่ชัดเจน หรือผู้พัฒนาซอฟต์แวร์มีความเข้าใจไม่ตรงกันกับความต้องการของลูกค้า
- ข้อผิดพลาดอาจเกิดขึ้นใน ขั้นตอนการทดสอบได้เช่นกัน เช่น ผู้ออกแบบการทดสอบกำหนดข้อมูลนำเข้าไม่สอดคล้องกับผลลัพธ์ที่ต้องการ ทำให้ได้ผลการทดสอบที่ไม่ถูกต้อง
- ข้อผิดพลาดมีแนวโน้มที่จะส่งผลกระทบต่อเนื่อง ข้อผิดพลาดจากขั้นตอนการรวบรวมความต้องการซอฟต์แวร์จะส่งผลให้เกิดข้อผิดพลาด ขึ้นในขั้นตอนการออกแบบระบบ และขยายผลไปยังขั้นตอนการเขียนโปรแกรม เป็นต้น

คำศัพท์ที่เกี่ยวข้อง

ความผิดพร่อง (Fault หรือ Defect)

- คือผลลัพธ์ที่เกิดขึ้นจากข้อผิดพลาดหรืออาจกล่าวได้ว่า เป็นตัวแทนของข้อผิดพลาดในรูปแบบต่างๆ
- เช่น หากมีข้อผิดพลาดเกิดขึ้นกับเอกสารความต้องการซอฟต์แวร์ เมื่อนำไปออกแบบระบบจะได้แผนภาพกระแสข้อมูล (Dataflow diagram) หรือ ผังงานโปรแกรม (Program flowchart) ที่ไม่ตรงตามความต้องการของลูกค้า เมื่อนำไปพัฒนาเป็นโปรแกรมก็จะได้รหัสต้นฉบับ (Source code) ที่ทำงานได้ไม่ตรงตามความต้องการของลูกค้า
- ความผิดพร่องสามารถแบ่งได้เป็น 2 ประเภท คือ

1) ความผิดพร่องที่เกิดจากการลงมือทำ (Fault of commission) เป็นการลงมือทำแต่ทำไม่ตรงตามความต้องการซอฟต์แวร์ที่กำหนดไว้ ทำให้มีความผิดพร่องเกิดขึ้นเป็นตัวแทนของข้อผิดพลาด

2) ความผิดพร่องที่เกิดจากการละเลย (Fault of omission) คือ ซอฟต์แวร์เกิดข้อผิดพลาดขึ้นเพราะลืมหรือละเลยสิ่งที่ต้องทำ ทำให้ไม่มีความผิดพร่องเกิดขึ้น เป็นตัวแทนของข้อผิดพลาดนั้น ซึ่งความผิดพร่องประเภทนี้ค้นพบและแก้ไขได้ยากกว่า

คำศัพท์ที่เกี่ยวข้อง

ความขัดข้อง (Failure)

- คือสิ่งที่เกิดขึ้นเมื่อมีการนำส่วนที่มีความผิดพร่องไปประมวลผล
- มักจะพบความขัดข้องได้จากการประมวลผลรหัสต้นฉบับที่มีความผิดพร่องปรากฏอยู่ เช่น มีการใช้คำสั่งผิด เมื่อประมวลผลโปรแกรมก็จะได้ผลลัพธ์ที่ผิด
- สำหรับความผิดพร่องที่เกิดจากการละเลย จะไม่พบข้อขัดข้องเมื่อประมวลผลโปรแกรมเพราะไม่มีรหัสต้นฉบับสำหรับส่วนนั้น
- วิธีที่จะช่วยให้ ค้นหาความผิดพร่องประเภทนี้พบได้ คือ ทำการทบทวน (Review) อย่างรอบคอบ เช่น เตรียมเอกสารการออกแบบระบบให้นักออกแบบระบบช่วยกันทบทวนหาความผิดพร่อง หรือค้นหาสิ่งที่ลืมหรือละเลยในการออกแบบระบบ เพื่อป้องกันไม่ให้มีความขัดข้องเกิดขึ้น

คำศัพท์ที่เกี่ยวข้อง

เหตุการณ์ (Incident)

- คือเหตุการณ์ใดๆ ที่เกิดขึ้นเมื่อมีการประมวลผลโปรแกรมซึ่งบ่งบอกให้รู้ว่ามีความขัดข้องเกิดขึ้น
- โดยเหตุการณ์ที่เกิดขึ้นอาจแสดงอาการออกมาให้ผู้ใช้ ลูกค้า หรือผู้ ทดสอบสังเกตเห็นหรือไม่ก็ได้
- เป็นเหตุการณ์ที่ผู้ทดสอบจะต้องสืบสวนหาข้อผิดพลาดที่เป็นสาเหตุของความขัดข้องนั้น

คำศัพท์ที่เกี่ยวข้อง

การทดสอบ (Test)

- คือการนำกรณีทดสอบมาทดสอบกับโปรแกรมตามขั้นตอนที่ได้กำหนดขึ้น มีการสังเกต จดบันทึก และประเมินผลโปรแกรมหรือส่วนประกอบ (Component) ของโปรแกรม
- การทดสอบมีเป้าหมายสองประการด้วยกัน คือ เพื่อค้นหาความขัดข้องให้พบ และเพื่อแสดงให้เห็นว่าโปรแกรมทำงานได้ถูกต้องแล้ว
- การทดสอบอาจเป็นการทดสอบความถูกต้องในการทำงานของโปรแกรมในเส้นทาง (Path) ที่สนใจ หรือเป็นการทดสอบว่าโปรแกรมทำงานถูกต้องตามข้อกำหนดคุณลักษณะระบบหรือไม่

กรณีทดสอบ (Test case)

- จะสร้างขึ้นตามลักษณะพฤติกรรมการทำงานของโปรแกรม
- ประกอบด้วย ข้อมูลนำเข้า (Input) เพื่อทดสอบโปรแกรม เงื่อนไขในการทดสอบ และผลลัพธ์ที่คาดหวังว่าจะได้รับ (Expected output)

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

- **กระบวนการพัฒนาซอฟต์แวร์ (Software development process) คือ กลุ่มของขั้นตอนการทำงาน ประกอบด้วยชุดของกิจกรรม ข้อจำกัด และทรัพยากรที่จะผลิตเป็นผลลัพธ์ตามต้องการ**
- **กระบวนการพัฒนาซอฟต์แวร์มีหลายรูปแบบ โดยส่วนใหญ่จะมีกิจกรรมหลักที่ต้องทำไม่แตกต่างกันมากนัก**
- **ในแต่ละกิจกรรมมีโอกาสเกิดข้อผิดพลาดขึ้นได้ทั้งสิ้น ทำให้ผลลัพธ์ของข้อผิดพลาดหรือความผิดพร่องถูกส่งต่อไปยังกิจกรรมอื่นๆ ในกระบวนการพัฒนาซอฟต์แวร์**
- **การทดสอบจึงเป็นกิจกรรมอย่างหนึ่งที่จำเป็นต้องมีในกระบวนการพัฒนาซอฟต์แวร์ทุกรูปแบบ**

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

กิจกรรมหลักของการพัฒนาซอฟต์แวร์มีดังนี้

1. การสื่อสาร (Communication)

- **ผู้พัฒนาซอฟต์แวร์จะติดต่อสื่อสารกับลูกค้า เพื่อรวบรวมความต้องการซอฟต์แวร์จากลูกค้า นำมาเขียนเป็นเอกสารความต้องการซอฟต์แวร์**
- **เป็นการอธิบายโจทย์ปัญหาที่ลูกค้าต้องการ รวมทั้งการติดต่อสื่อสารกับลูกค้าเพื่อให้ลูกค้าประเมินผลชิ้นงานที่เกิดขึ้นในขั้นตอนต่างๆ ในระหว่างกระบวนการพัฒนาซอฟต์แวร์และรับผลป้อนกลับ(Feedback) จากลูกค้า**
- **ในขั้นตอนการสื่อสารลูกค้าอาจบอกความต้องการไม่ถูกต้อง ไม่ครบถ้วนหรือไม่ชัดเจน หรือผู้ที่ทำหน้าที่รวบรวมความต้องการมีความเข้าใจไม่ตรงกับความต้องการของลูกค้า ทำให้ได้เอกสารความต้องการซอฟต์แวร์ที่มีข้อผิดพลาด**

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

1. การสื่อสาร (Communication)

Functional Requirement

1. Customer can see where are their parcel via tracking parcel feature on smartphone application.
2. Postman can access to route suggestion from website or smartphone application.
3. Customer can ask for the service rate.
4. Customer and Staff be able to know if the parcel arrived at destination and no one received.
5. System can notice to the sender immediately when the parcel arrived the destination.
6. Customer can view location of post offices.
7. Customer can create new account to access application.
8. Customer can
9. Customer can
10. Staff can do p
11. Staff can print
12. User and staff
13. Administrator
14. Administrator

Credit : https://sites.google.com/site/projectpandse

Non-functional requirement

1. Application must support on iOS and Android.
2. The application and website must been easy to develop and create a new feature.
3. Tracking system must been accuracy.
4. Application and website graphic user interface must been easy to use.
5. Application is needed to support up-to-date security system such as fingerprint scanner on today mobile devices.
6. The system should be used to store data on the cloud.
7. Website must use HTML5 platform
8. Smartphone Application and Website must use the same database.
9. Application and website is needed to provide appropriate amount of advertising to customers.

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

2. การวางแผน (Planning)

- เกี่ยวข้องกับการประมาณการค่าใช้จ่าย กำลังคน และเวลาที่ต้องใช้
- การกำหนดตารางเวลาในการทำงาน และการติดตามความก้าวหน้าในการพัฒนาซอฟต์แวร์
- ในแผนการพัฒนาซอฟต์แวร์จะต้องมีการวางแผนสำหรับกิจกรรมการทดสอบ และการทวนสอบ ซึ่งเป็นกิจกรรมที่ใช้ค้นหาข้อผิดพลาดของชิ้นงานจากขั้นตอนต่างๆ ในระหว่างกระบวนการพัฒนาซอฟต์แวร์ และค้นหาความขัดข้องเมื่อประมวลผลโปรแกรม รวมทั้งการประเมินว่าเป็นไปตามข้อกำหนดคุณลักษณะระบบหรือไม่

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

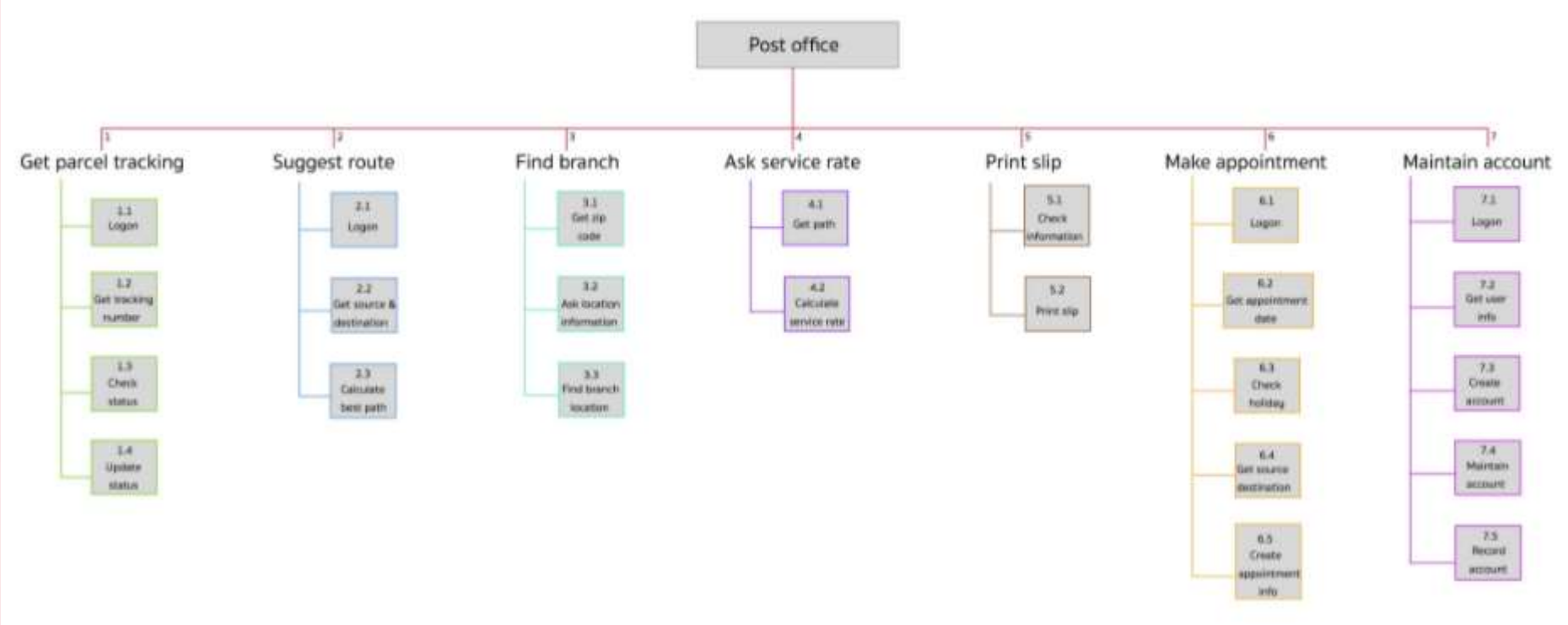
3. การสร้างแบบจำลอง (Modeling)

- เป็นการวิเคราะห์ความต้องการซอฟต์แวร์ของลูกค้า และนำมาเขียนเป็นข้อกำหนดคุณลักษณะซอฟต์แวร์ ซึ่งจะอธิบายรายละเอียดของงานที่จะต้องทำเพื่อแก้โจทย์ปัญหาตามที่ลูกค้าต้องการ
- เป็นการออกแบบระบบโดยสร้างตัวแทนของซอฟต์แวร์ในรูปแบบต่างๆ ขึ้นมา เช่น การออกแบบสถาปัตยกรรม การออกแบบกระบวนการทำงาน การออกแบบฐานข้อมูล การออกแบบส่วนติดต่อกับผู้ใช้งาน และการออกแบบรายงาน เป็นต้น
- ขั้นตอนการออกแบบระบบมีโอกาสเกิดข้อผิดพลาดได้ เช่น ออกแบบระบบไม่สอดคล้องกับข้อกำหนดคุณลักษณะซอฟต์แวร์ นอกจากนี้ข้อผิดพลาดของการออกแบบระบบอาจเกิดจากการออกแบบตามความผิดพร่องในเอกสารความต้องการซอฟต์แวร์

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

3. การสร้างแบบจำลอง (Modeling)

Functional decomposition diagram (Structure chart)



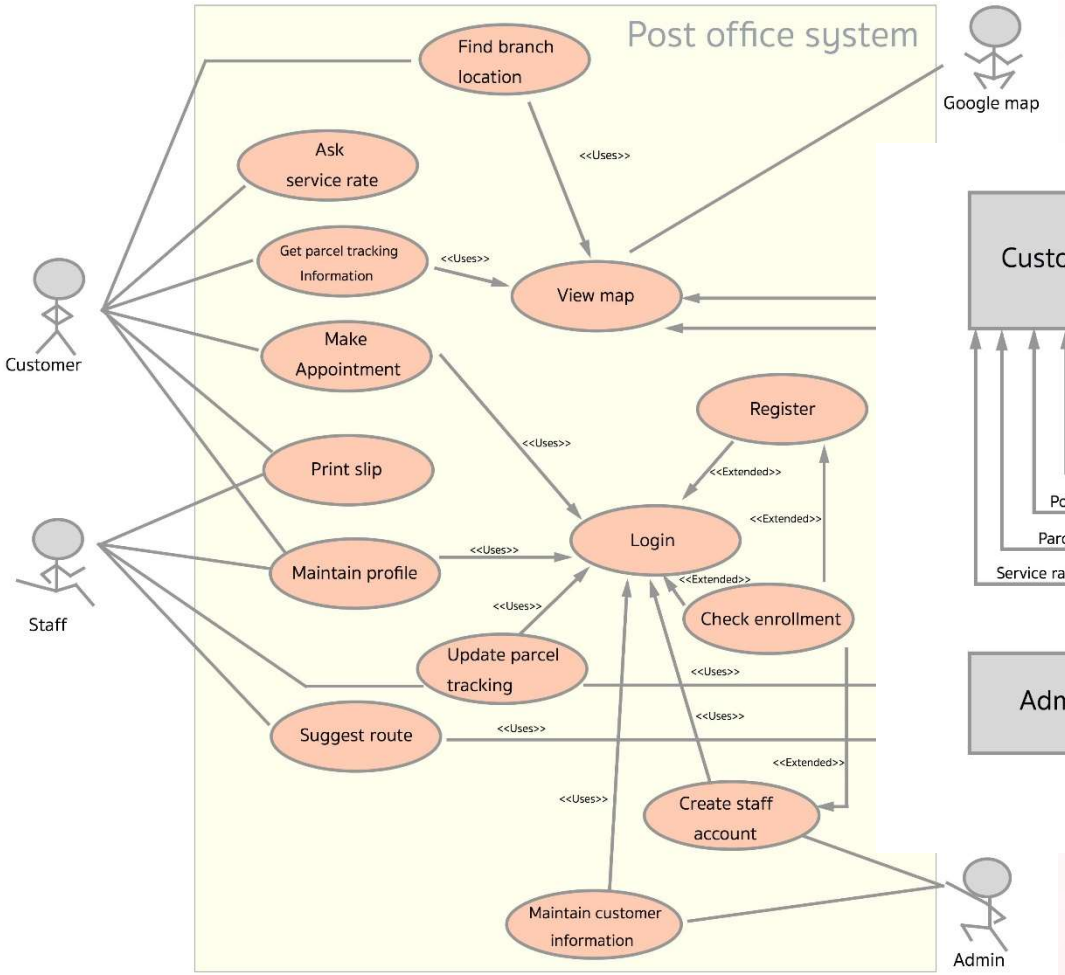
Credit รูป : https://sites.google.com/site/projectpandse

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

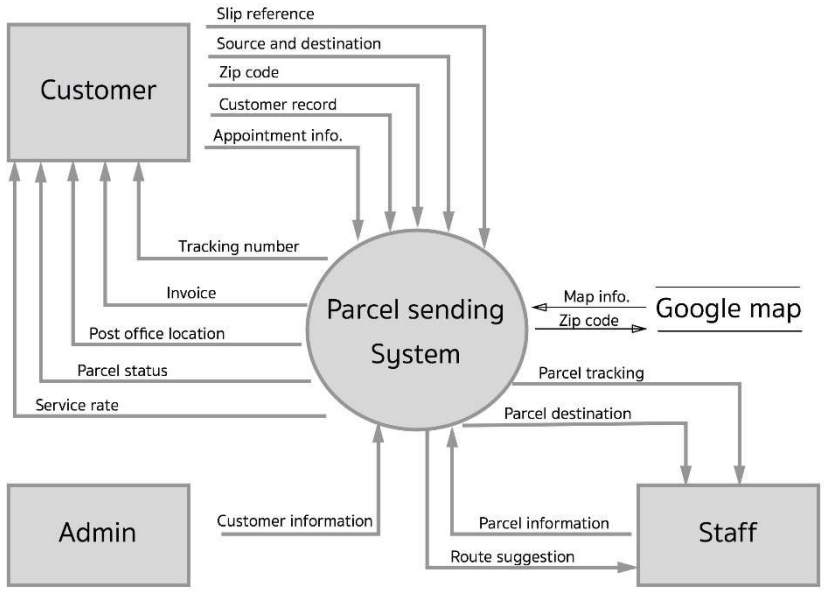
3. การสร้างแบบจำลอง (Modeling)

Use case diagram

Credit รูป : https://sites.google.com/site/projectpandse



Context diagram

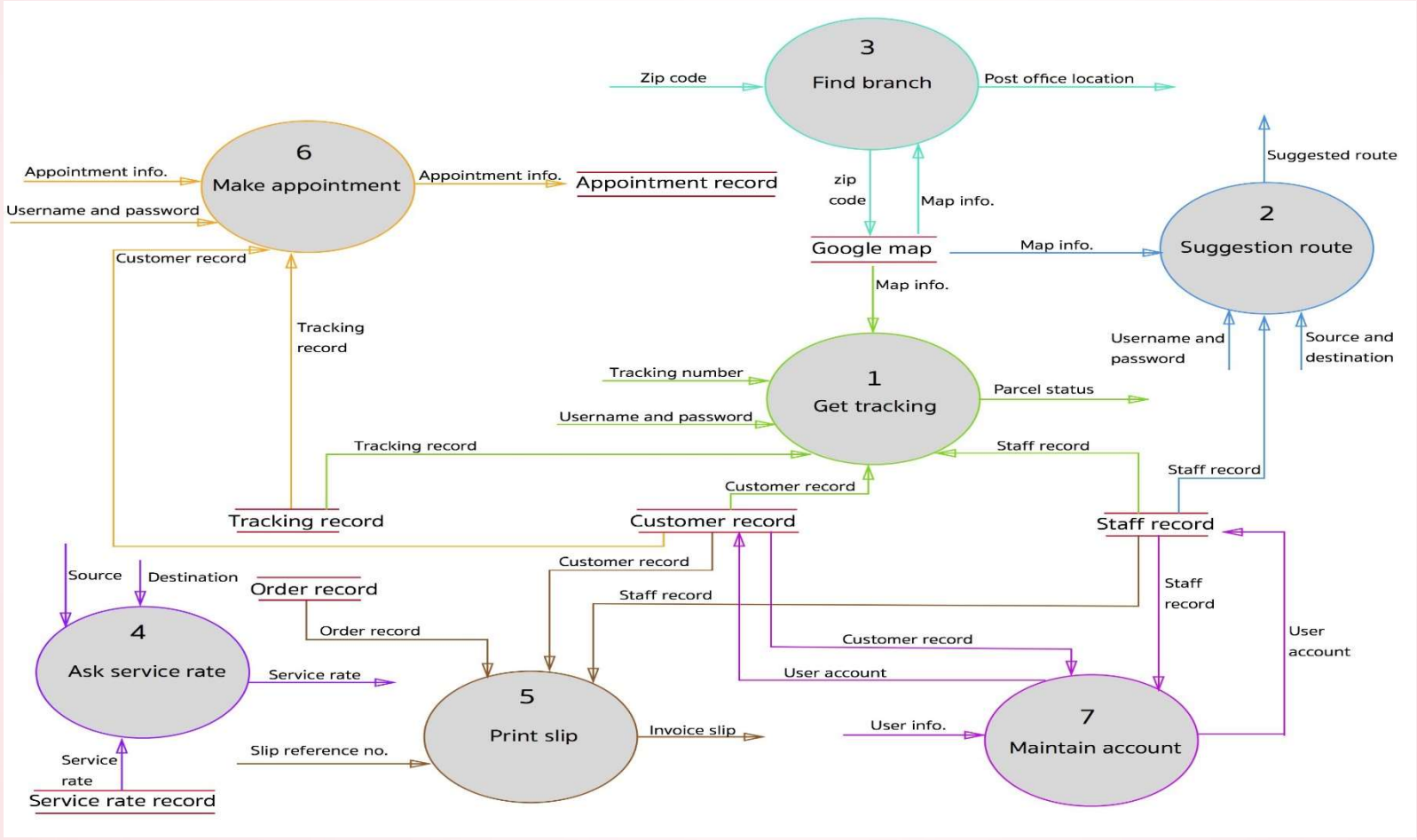


กิจกรรมหลักของการพัฒนาซอฟต์แวร์

3. การสร้างแบบจำลอง (Modeling)

Data flow diagram- level 1

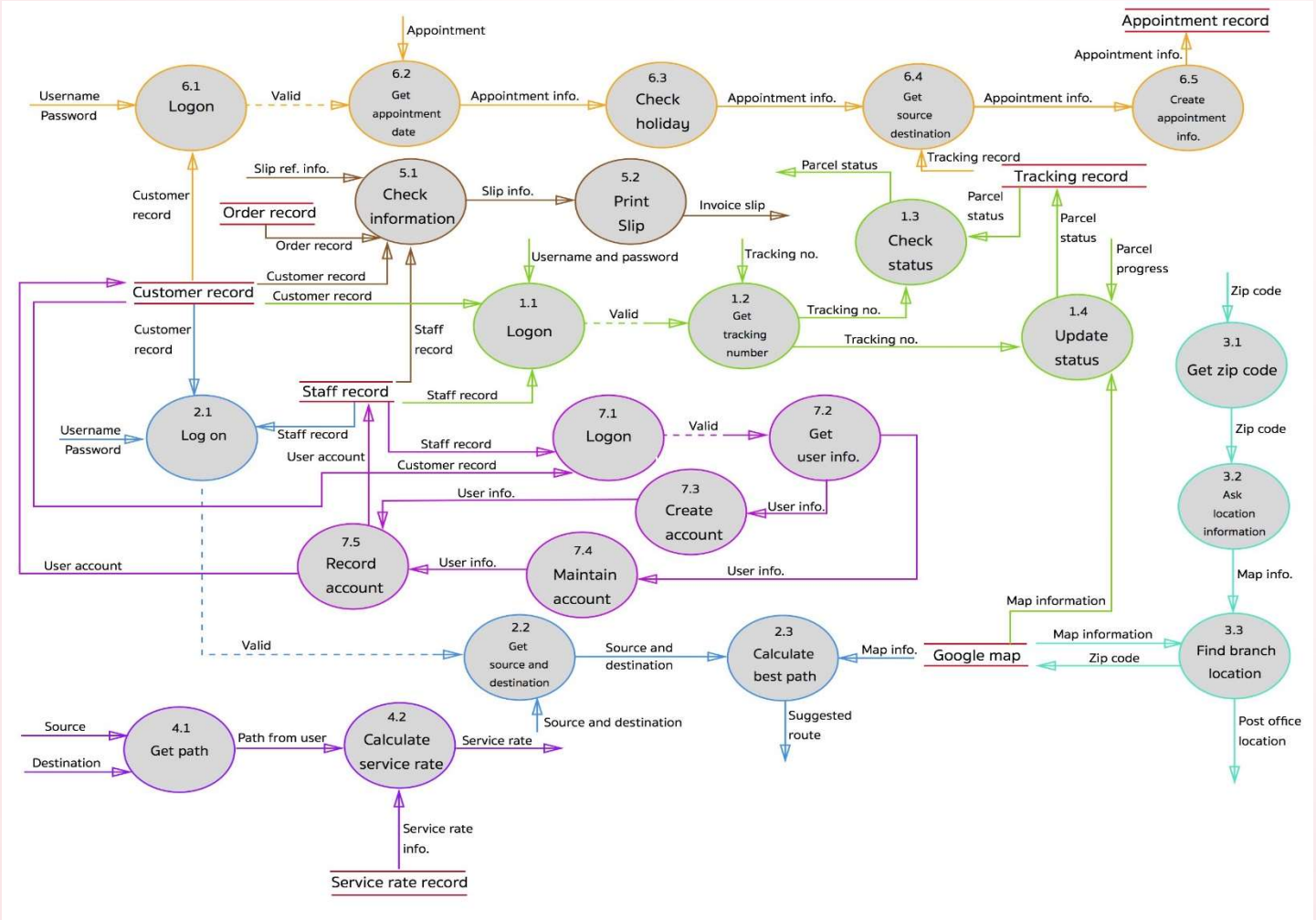
Credit รูป : https://sites.google.com/site/projectpandse



กิจกรรมหลักของการพัฒนาซอฟต์แวร์

3. การสร้างแบบจำลอง (Modeling)

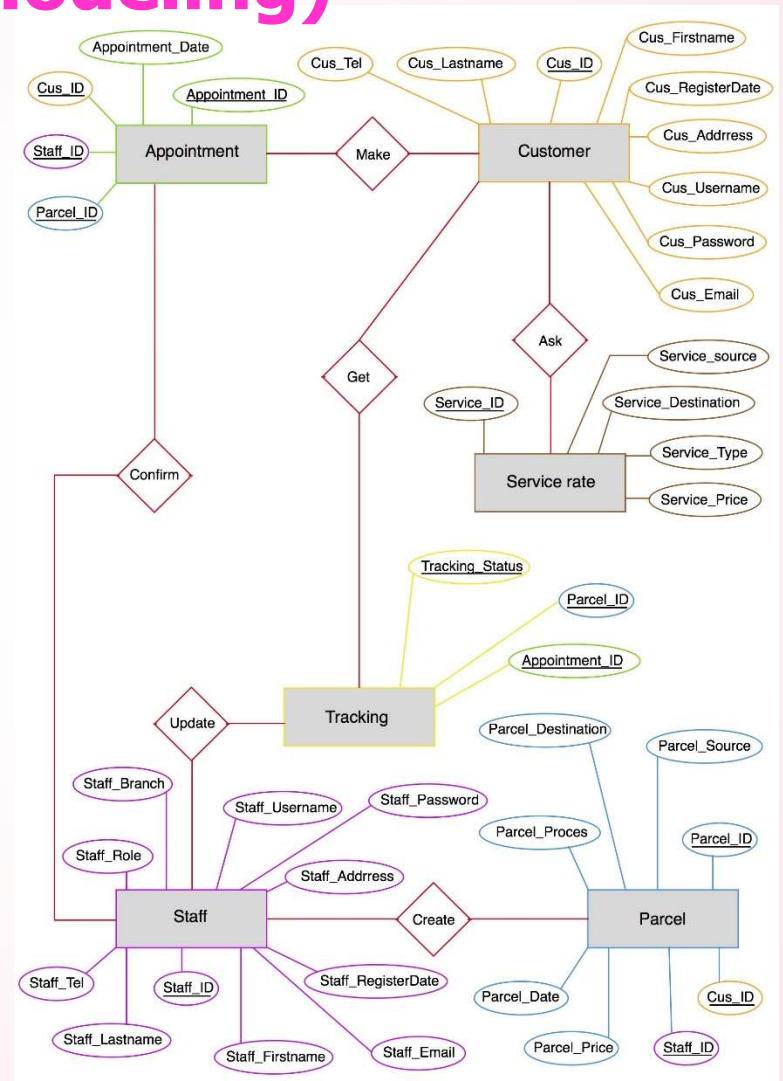
Data flow diagram- level 2 **Credit รูป : https://sites.google.com/site/projectpandse**



กิจกรรมหลักของการพัฒนาซอฟต์แวร์

3. การสร้างแบบจำลอง (Modeling)

ER diagram

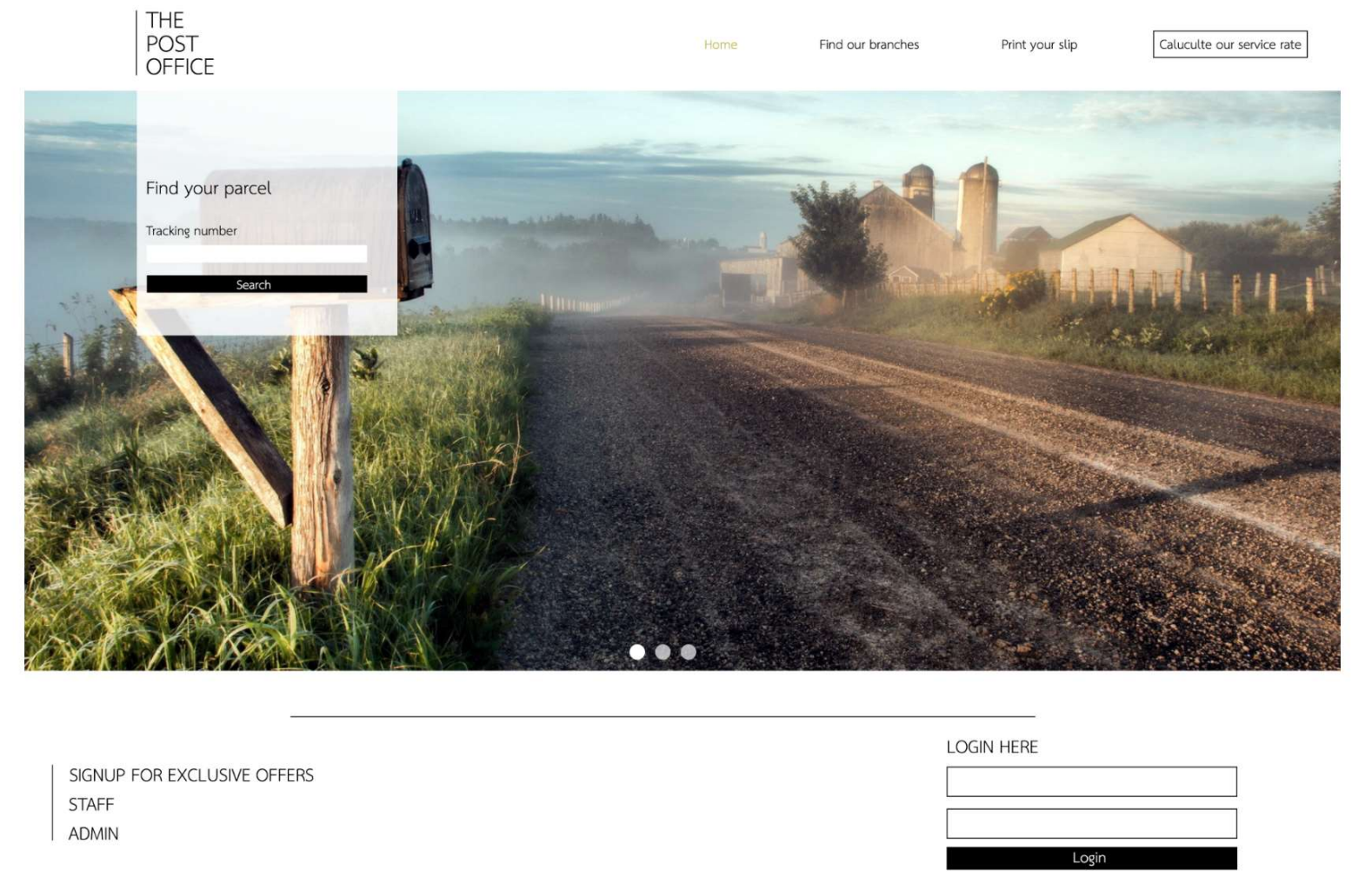


Credit รูป : https://sites.google.com/site/projectpandse

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

3. การสร้างแบบจำลอง (Modeling)

Conceptual design prototype **Credit รูป : https://sites.google.com/site/projectpandse**



กิจกรรมหลักของการพัฒนาซอฟต์แวร์

4. การสร้าง (Construction)

- การสร้างซอฟต์แวร์เป็นการเขียนโปรแกรมด้วยภาษาที่กำหนดเพื่อสร้างรหัสต้นฉบับขึ้นมาตามที่ได้มีการออกแบบระบบเอาไว้ และทำการทดสอบโปรแกรมนั้น
- ทั้งนี้จะต้องทดสอบว่ารหัสต้นฉบับที่สร้างขึ้นมีความสอดคล้องตรงกันกับเอกสารการออกแบบระบบหรือไม่ และทดสอบซอฟต์แวร์ที่พัฒนาเสร็จแล้วว่าเป็นไปตามข้อกำหนดคุณลักษณะระบบหรือไม่
- ขั้นตอนการเขียนโปรแกรมมีโอกาสเกิดข้อผิดพลาดได้ เช่น ผู้เขียนโปรแกรมใช้คำสั่งผิดหรือข้อผิดพลาดอาจเกิดขึ้นจากการเขียนโปรแกรมตามความผิดพร่องในเอกสารการออกแบบระบบ

กิจกรรมหลักของการพัฒนาซอฟต์แวร์

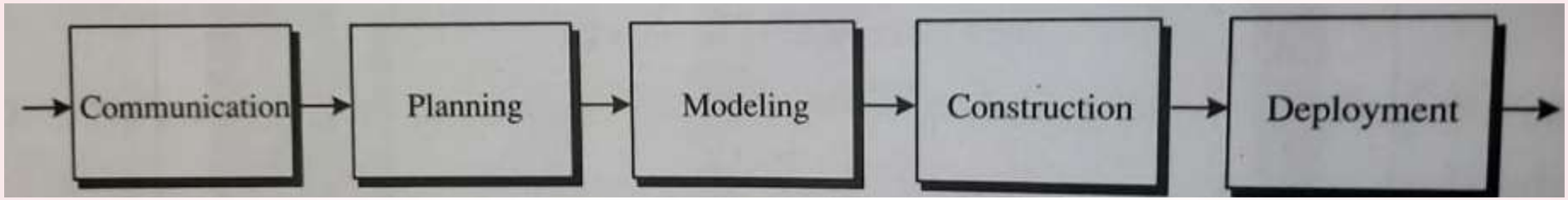
5. การติดตั้งใช้งาน (Deployment)

- เกี่ยวข้องกับการส่งมอบงาน การติดตั้งระบบ การจัดทำคู่มือการใช้งาน การอบรมผู้ใช้ การบำรุงรักษาระบบ และการรับผลป้อนกลับจากลูกค้า

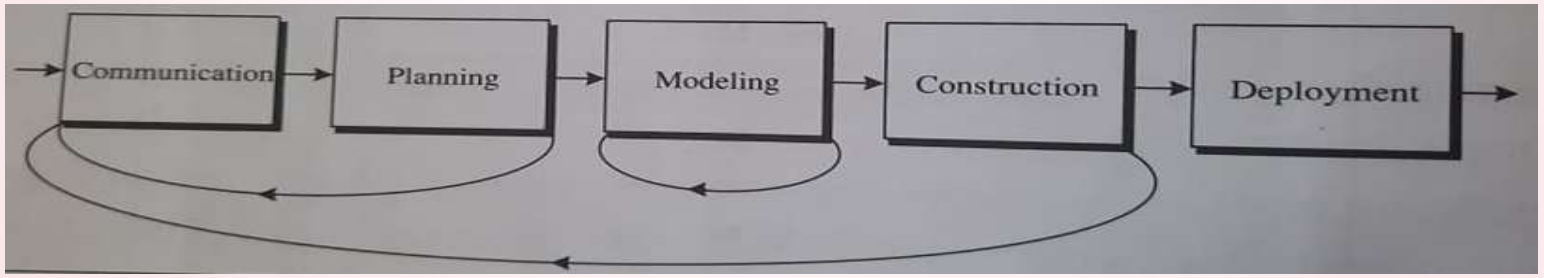
การทำกิจกรรมการทดสอบ

จะขึ้นอยู่กับกระบวนการพัฒนาซอฟต์แวร์ที่เลือกใช้ซึ่งมีหลายรูปแบบ เช่น

1. กระบวนการที่ทำงานตามลำดับขั้น (Linear process flow) เป็นการทำกิจกรรมตามลำดับ แต่ละกิจกรรมจะทำครั้งเดียวจนเสร็จจึงจะเริ่มทำกิจกรรมถัดไป ดังนั้นการทดสอบจะทำครั้งเดียวเมื่อเขียนโปรแกรมเสร็จแล้ว



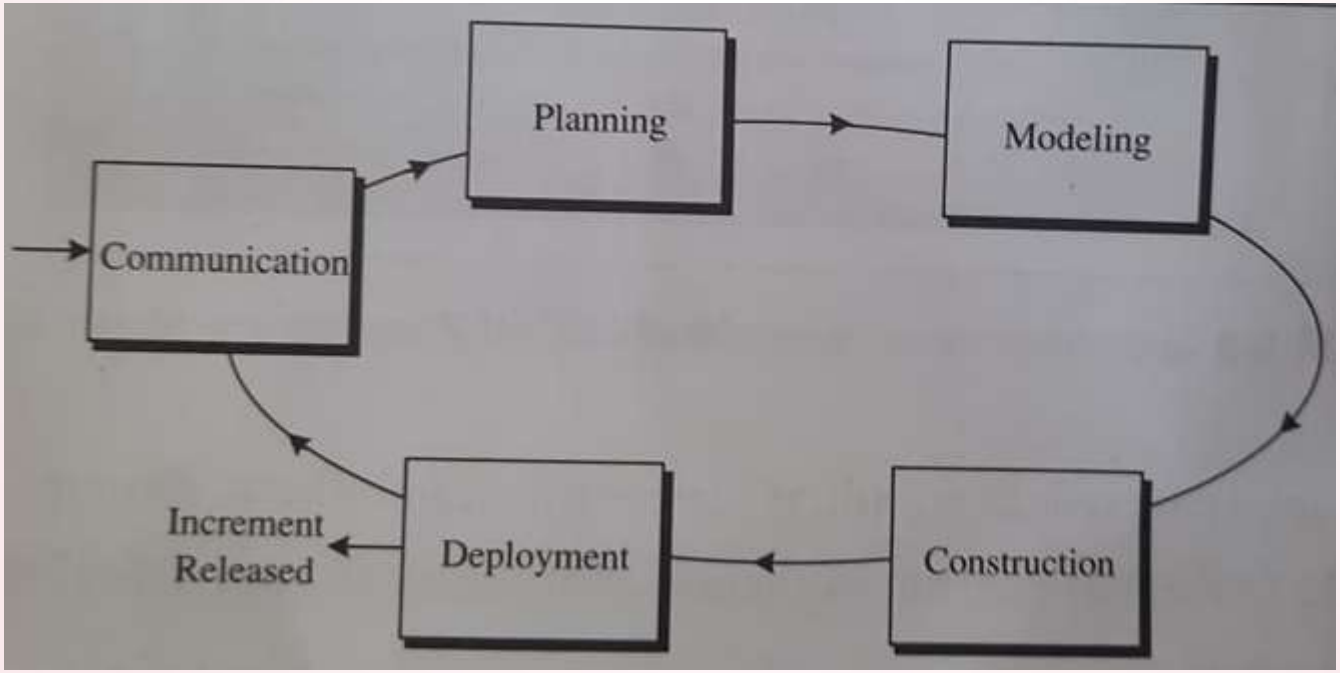
2. กระบวนการที่วนทำซ้ำได้ (Iterative process flow) สามารถวนกลับมาทำกิจกรรมเดิมซ้ำอย่างน้อย 1 กิจกรรม เช่น เมื่อทำการสื่อสารกับลูกค้าแล้วนำข้อมูลกลับมาวางแผน จากนั้นวนกลับไปทำการสื่อสารกับลูกค้าซ้ำอีก แล้วปรับปรุงแผนการที่วางไว้ ดังนั้นการทดสอบอาจกลับมาทำซ้ำได้หลายครั้ง



Credit รูป : ผศ.ดร. น้ำฝน อัศวเมฆิน

การทำกิจกรรมการทดสอบ

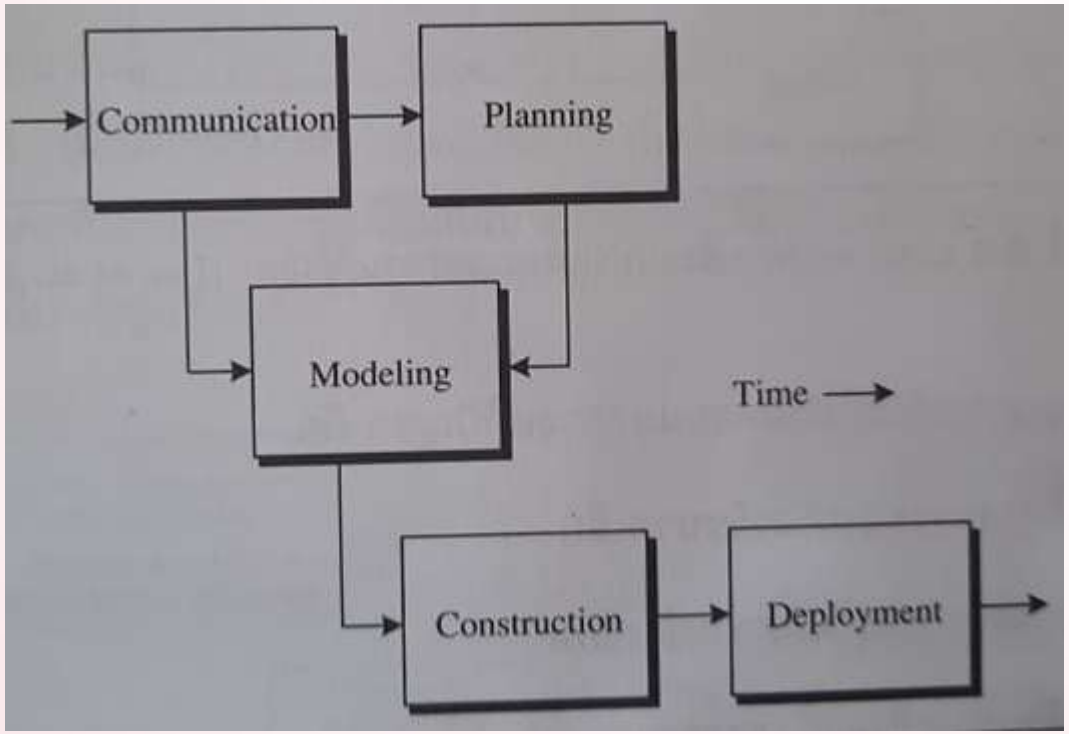
3. **กระบวนการเชิงวิวัฒนาการ (Evolutionary process flow) เป็นการทำซ้ำหลายรอบได้ แต่ละรอบจะทำซ้ำครบทุกกิจกรรมตามลำดับ โดยแต่ละรอบจะได้ซอฟต์แวร์ที่สมบูรณ์มากขึ้น เรื่อยๆ ดังนั้นการทดสอบในแต่ละรอบอาจเป็นการทดสอบฟังก์ชันที่สร้างขึ้นเพิ่มเติม**



Credit รูป : ผศ.ดร. น้ำฝน อัศวเมฆิน

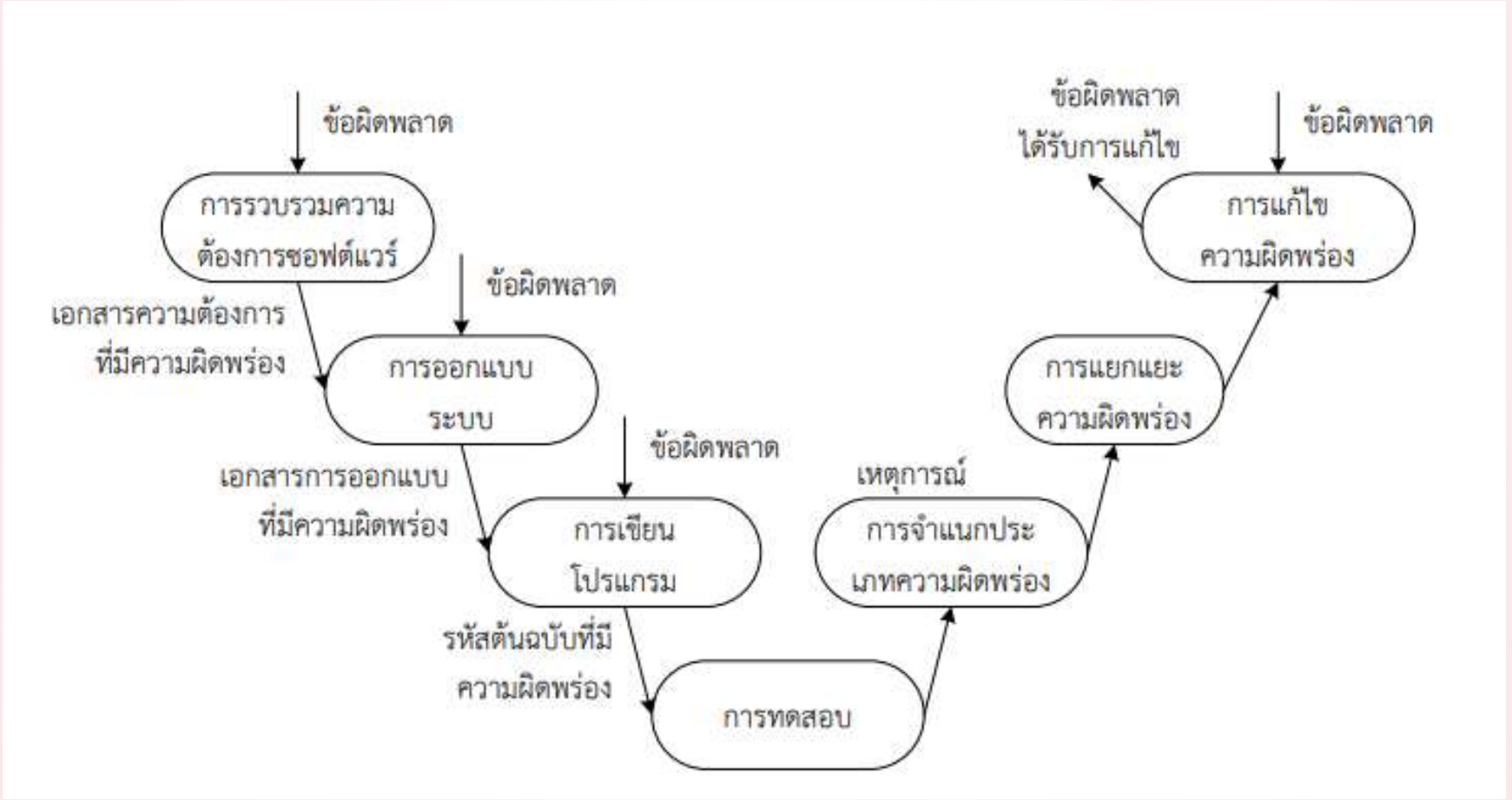
การทำกิจกรรมการทดสอบ

4. กระบวนการแบบขนาน (Parallel process flow) มีการทำกิจกรรมมากกว่า 1 กิจกรรมไปพร้อมๆ กันได้ เช่น การทำกิจกรรม Communication ควบคู่กับการ Planning เช่นเดียวกันการทดสอบสำหรับแต่ละส่วนของซอฟต์แวร์อาจทำไปพร้อมกันได้



Credit รูป : ผศ.ดร. น้ำฝน อัศวเมฆิน

วัฏจักรของการทดสอบ



Test cases

- หัวใจสำคัญของการทดสอบซอฟต์แวร์ คือการเลือก Test case
- การออกแบบ Test case แต่ละกรณีจะต้องมีความเฉพาะเจาะจงและมีเหตุผลมาสนับสนุน เช่น ออกแบบมาเพื่อทดสอบตามข้อกำหนดคุณลักษณะระบบข้อใด
- Test case ที่สร้างขึ้นจำเป็นต้องผ่านกระบวนการทบทวนด้วย เพื่อให้มั่นใจว่าได้ออกแบบการทดสอบครอบคลุมกรณีต่างๆ อย่างเพียงพอแล้ว

Test cases

ในการออกแบบ **Test case** จะประกอบด้วยข้อมูลสำคัญดังนี้

1) **Test case ID:** หมายเลขกรณีทดสอบที่ไม่ซ้ำกัน เพื่อใช้ในการอ้างอิง

2) **Project ID:** อ้างอิงหมายเลขโครงการพัฒนาซอฟต์แวร์ เพื่อให้ทราบว่าเป็นกรณีออกแบบมาเพื่อใช้ในโครงการใด

3) **Requirement No.:** อ้างอิงหมายเลขความต้องการซอฟต์แวร์จากเอกสารข้อกำหนดคุณลักษณะซอฟต์แวร์

4) **Purpose:** วัตถุประสงค์ของการออกแบบกรณีทดสอบ เช่น ออกแบบมาเพื่อทดสอบข้อกำหนดคุณลักษณะระบบข้อใด

Test cases

5) Environment: สภาพแวดล้อมของระบบคอมพิวเตอร์ที่ต้องจัดเตรียมในการทดสอบ อธิบายฮาร์ดแวร์ ซอฟต์แวร์ ลักษณะเครือข่ายคอมพิวเตอร์ที่ต้องใช้ เป็นต้น

6) Procedure: ขั้นตอนในการทดสอบ

7) Input: ข้อมูลนำเข้าที่จะป้อนเข้าไปในโปรแกรมหรือระบบ

8) Expected output: ผลลัพธ์ที่คาดว่าจะได้รับจากโปรแกรมหรือระบบ

9) Execution history: ประวัติการทดสอบ เช่น วันที่ทำการทดสอบ รุ่นของโปรแกรมที่ทำการทดสอบ ผลลัพธ์ของการทดสอบ ผู้ทำหน้าที่ทดสอบ เป็นต้น

หมวดหมู่ของความผิดพร่อง (Fault หรือ Defect)

การแบ่งหมวดหมู่ของความผิดพร่องสามารถทำได้ในหลายลักษณะ เช่น แบ่งตามขั้นตอนของการพัฒนาซอฟต์แวร์ที่ทำให้เกิดข้อผิดพลาดขึ้น แบ่งตามลักษณะของความขัดข้องที่พบ แบ่งตามความยากง่ายในการแก้ปัญหา หรือแบ่งตามลักษณะการเกิดข้อผิดพลาด

ตัวอย่างการจัดหมวดหมู่ของความผิดพร่องแบ่งได้ดังนี้

1. ความผิดพร่องในการรับข้อมูลนำเข้า (Input fault) เช่น โปรแกรมไม่ยอมรับค่าที่ถูกต้อง โปรแกรมยอมรับค่าที่ไม่ถูกต้อง คำชี้แจงว่าต้องป้อนข้อมูลนำเข้าอะไรเข้าไปในโปรแกรมบรรยายไว้ผิดหรือไม่มีคำอธิบาย
2. ความผิดพร่องในการแสดงผลลัพธ์ (Output fault) เช่น รูปแบบของผลลัพธ์ผิดไปจากที่กำหนด แสดงผลลัพธ์ผิด แสดงผลลัพธ์ถูกต้องแต่แสดงเร็วเกินไปหรือช้าเกินไป แสดงผลลัพธ์ไม่ครบถ้วน การสะกดคำไม่ถูกต้องหรือผิดหลักไวยากรณ์

หมวดหมู่ของความผิดพร่อง (Fault หรือ Defect)

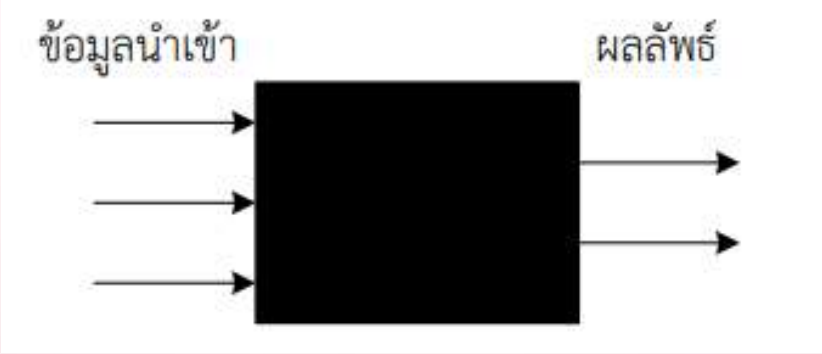
3. ความผิดพร่องเกี่ยวกับตรรกะ (Logic fault) เช่น จำนวนรอบในการวนทำซ้ำของโปรแกรมไม่ถูกต้อง การใช้เครื่องหมายทางคณิตศาสตร์ผิด (เช่น ใช้ < แต่ที่ควรจะเป็นคือ <=) เงื่อนไขของโปรแกรมไม่ครบถ้วน เงื่อนไขของโปรแกรมมีมากเกินกว่าที่กำหนด เงื่อนไขของโปรแกรมตรวจสอบตัวแปรผิดตัว
4. ความผิดพร่องในการคำนวณ (Computation fault) เช่น ขั้นตอนวิธีไม่ถูกต้อง ตัวแปรที่ใช้ในสูตรคำนวณไม่ถูกต้อง เครื่องหมายที่ใช้ในสูตรคำนวณไม่ถูกต้อง
5. ความผิดพร่องในการทำงานร่วมกัน (Interface fault) เช่น เรียกใช้ฟังก์ชันผิด หรือเรียกใช้ฟังก์ชันที่ไม่มีอยู่ ตัวแปรที่ส่งไปยังฟังก์ชันมีชนิดไม่ถูกต้อง หรือจำนวนตัวแปรที่ส่งไปยังฟังก์ชันไม่ครบ
6. ความผิดพร่องเกี่ยวกับข้อมูล (Data fault) เช่น กำหนดค่าเริ่มต้นของตัวแปรไม่ถูกต้อง การอ้างอิงดรรชนี (Index) ของตัวแปรไม่ถูกต้อง กำหนดข้อมูลให้กับตัวแปรซึ่งเป็นคนละชนิดกันหรือการจัดเก็บข้อมูลเกินกว่าขอบเขตของตัวแปร

วิธีการทดสอบ

- วิธีการในการทดสอบสามารถแบ่งเป็น 2 วิธีการหลักๆ คือ การทดสอบเชิงฟังก์ชัน และการทดสอบเชิงโครงสร้าง
- โดยแต่ละวิธีจะมีวัตถุประสงค์ในการทดสอบที่แตกต่างกัน
- **การทดสอบเชิงฟังก์ชัน** จะทดสอบเพื่อประเมินผลว่าซอฟต์แวร์ทำงานได้ถูกต้องตรงตามข้อกำหนดคุณลักษณะระบบหรือไม่
- **การทดสอบเชิงโครงสร้าง** ใช้เพื่อทดสอบว่ารหัสต้นฉบับที่สร้างขึ้นมีความ สอดคล้องตรงกันกับการออกแบบระบบหรือไม่

การทดสอบเชิงฟังก์ชัน (Functional testing หรือ Black box testing)

- การทดสอบเชิงฟังก์ชันใช้เมื่อโปรแกรมมีลักษณะเป็นฟังก์ชันใดๆ ที่ทำหน้าที่ประมวลผลข้อมูลนำเข้าแล้วได้ผลลัพธ์ออกมา
- จะมองระบบทั้งระบบเหมือนเป็นกล่องดำ (Black box) ผู้ทดสอบไม่จำเป็นต้องรู้กระบวนการทำงานภายในโปรแกรม
- เปรียบเทียบกับการทดสอบในชีวิตประจำวันของคนเราได้ เช่น ลูกค้าทดลองขับรถยนต์โดยที่ไม่ต้องรู้กลไกการทำงานภายในของเครื่องยนต์
- การสร้างกรณีทดสอบจะสร้างโดยพิจารณาความต้องการซอฟต์แวร์ของลูกค้าจากเอกสารข้อกำหนดคุณลักษณะระบบ
- ตัวอย่างเทคนิค ได้แก่ Equivalence class partitioning และ Boundary value analysis



Equivalence class partitioning

- แบ่งส่วนข้อมูลนำเข้าออกเป็นกลุ่มๆ แล้วเลือกข้อมูล ตัวแทนของกลุ่ม ซึ่งมีเพียง 2 สถานะ คือ **Valid** กับ **Invalid** เพื่อนำมาทดสอบ
- ข้อมูลตัวแทนของกลุ่ม ได้แก่
  - ค่าตัวแทนกลุ่ม
  - ค่าต่ำสุดของกลุ่ม
  - ค่าสูงสุดของกลุ่ม
  - ค่าเกินพิกัด ซึ่งอาจเป็นค่าที่น้อยกว่าค่าต่ำสุด หรือ ค่าที่มากกว่าค่าสูงสุด
  - ค่าผิดปกติ เช่น ค่าติดลบ ค่าที่มีชนิดแตกต่างจากข้อมูลของกลุ่ม
- หลักการ: ข้อมูลนำเข้าที่ **Valid** ย่อมให้ผลลัพธ์ที่ **Valid** ด้วย

Equivalence class partitioning

- ระบบ ATM ของธนาคารแห่งหนึ่ง มีฟังก์ชันการทำงานดังนี้
  - เลือก 1 หรือ 2 เพื่อระบุประเภทบัญชี
    - 1. บัญชีออมทรัพย์ (Saving)
    - 2. บัญชีกระแสรายวัน (Checking)
  - เลือก D เพื่อทำรายการฝากเงิน (Deposit)
  - เลือก W เพื่อทำรายการถอนเงิน (Withdraw)
  - ระบุจำนวนเงิน(Amount) ฝากหรือถอน เป็นจำนวนเท่าของ 100 ซึ่งมีค่าตั้งแต่ 100 ถึง 20,000 บาท

ECP	ผลลัพธ์
1. เลือกบัญชี 1 หรือ 2	Valid
2. เลือกบัญชี < 1	Invalid
3. เลือกบัญชี > 2	Invalid
4. เลือก D หรือ W	Valid
5. เลือกที่ไม่ใช่ D หรือ W	Invalid
6. ระบุจำนวนเงิน 100-20,000 บาทและหารด้วย 100 ลงตัว	Valid
7. ระบุจำนวนเงิน < 100	Invalid
8. ระบุจำนวนเงิน > 20,000	Invalid
9. ระบุจำนวนเงิน 100-20,000 บาทและหารด้วย 100 ไม่ลงตัว	Invalid

- จากข้อกำหนดข้างต้น สามารถแบ่งข้อมูลนำเข้าออกเป็นกลุ่ม เพื่อทดสอบดังตัวอย่าง

Credit : ผศ.ดร. วัชรี จำปามูล

กรณีทดสอบ	ครอบคลุม ECP	ผลลัพธ์ที่คาดหวัง
T1: 1, D, 500	ECP1, ECP4, ECP6	Valid
T2: 0, W, 100	ECP2, ECP4, ECP6	Invalid

การทดสอบเชิงโครงสร้าง (Structural testing หรือ White box testing)

- การทดสอบเชิงโครงสร้าง อาจเรียกอีกอย่างว่าการทดสอบแบบกล่องขาว (White box testing) หรือ กล่องแก้ว (Glass box testing)
- เป็นการทดสอบที่คำนึงถึงชุดคำสั่งเป็นหลัก กรณีทดสอบจะถูกออกแบบตามโครงสร้างของโปรแกรม
- ใช้เมื่อรู้กระบวนการทำงานภายในของโปรแกรม จะใช้ขั้นตอนการทำงานนั้นในการกำหนดกรณีทดสอบขึ้นมา
- อาจใช้ทฤษฎีกราฟมาช่วยในการอธิบายเส้นทางการทำงานของโปรแกรม ทำให้สามารถตรวจสอบได้ว่าเส้นทางใดได้ออกแบบกรณีทดสอบไปแล้วหรือยัง
- นอกจากนี้ยังมีมาตรวัดความครอบคลุมของการทดสอบ (Test coverage metrics) ที่ช่วยให้ทราบว่าการทดสอบครอบคลุมเส้นทางการทำงานของโปรแกรมหรือคำสั่งเป็นสัดส่วนเท่าใดของ เส้นทางการทำงานของโปรแกรมหรือคำสั่งทั้งหมด

การทดสอบเชิงโครงสร้าง (Structural testing หรือ White box testing)

- ใช้กราฟสายงานควบคุม (Control Flow graph) หรือ กราฟสายงาน (Flow graph) แสดงแทนการทำงานของโปรแกรมที่ต้องการทดสอบ โดย
  - วงกลมหรือโหนด (Node) แทนคำสั่ง
  - ลูกศรหรือเส้นเชื่อม (Edge) แทนการไหล/ลำดับการทำงาน
  - แต่ละโหนดอาจมีเส้นเข้าได้ตั้งแต่หนึ่งเส้น แต่เส้นออกเพียงเส้นเดียว ยกเว้นโหนดเงื่อนไข ที่มีเส้นออก 2 เส้นที่สอดคล้องตามเงื่อนไข
- การทดสอบแบบกล่องขาวมีหลายประเภท เช่น
  - การทดสอบการไหลของส่วนควบคุม (Control flow testing)
  - การทดสอบเส้นมูลฐาน (Basis path testing)
  - การทดสอบการไหลของข้อมูล (Data flow testing)
  - การทดสอบแบบวนซ้ำ (Loop testing)
  - การทดสอบการกลายพันธ์ (Mutation testing)
  - การทดสอบแบบสุ่ม (Random testing)

Basis path testing

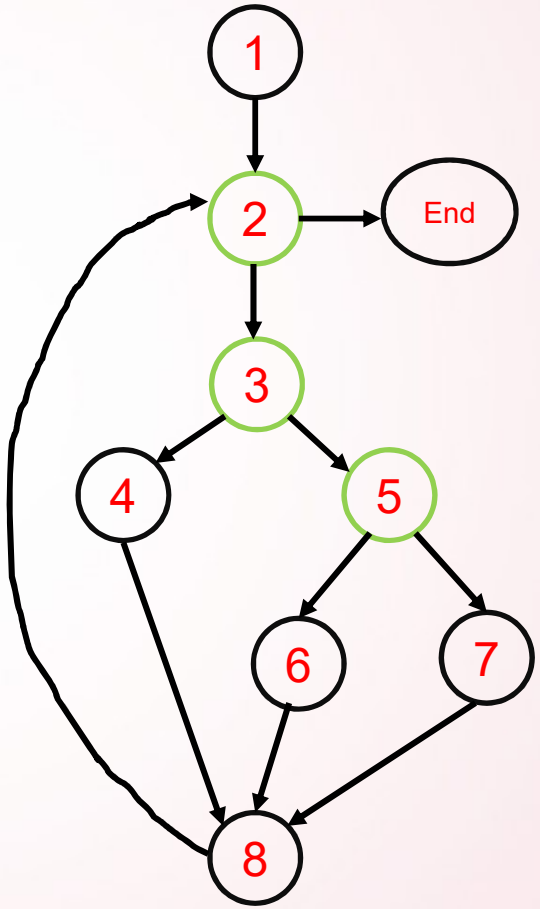
- มอดูลจะถูกทดสอบทุกๆเส้นทางที่เรียกว่าเป็น **Execution Path**
- **Exercise all statements :** ทดสอบทุกๆคำสั่งในโปรแกรม
- **Exercise all logical decision on T/F branches :** ทดสอบทุกการตัดสินใจทั้งเมื่อเงื่อนไขเป็นค่าจริงและค่าเท็จ
- **Exercise all independent paths :** ทดสอบทุกๆเส้นทางที่ประกอบด้วยโหนดไม่ซ้ำกัน
- **Exercise loop boundaries :** ทดสอบการทำงานของการวนซ้ำตามค่าขอบเขตของการซ้ำ
- **Exercise data structures :** ทดสอบโครงสร้างข้อมูลภายในให้ถูกต้องก่อนส่งไปประมวลผลที่หน่วยอื่น

Basis path testing

วนรับค่า x ทีละจำนวน แล้วแสดงผลว่าเป็นจำนวนลบ หรือเป็นเลขคู่ หรือเป็นเลขคี่ จนกว่าจะเจอค่า x ที่ 0 จึงจะหยุดการทำงาน

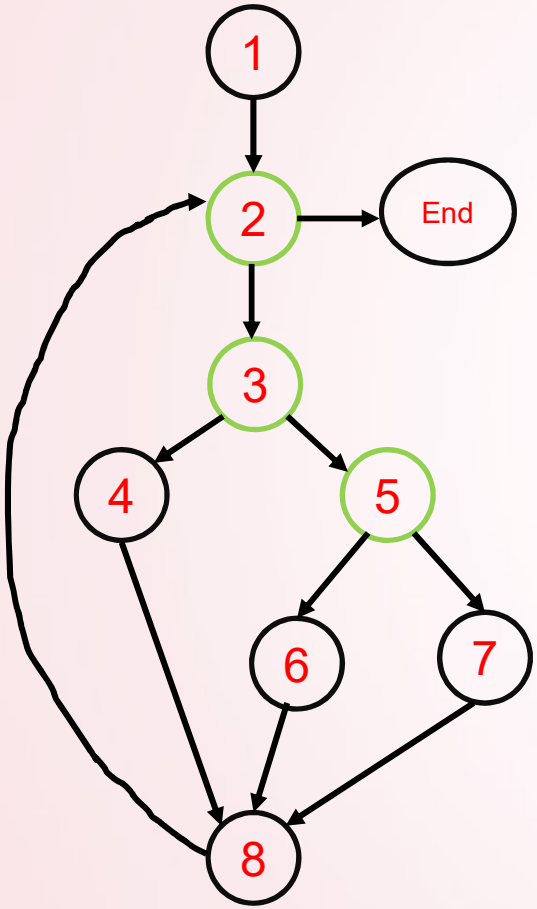
```
read x;                                  (1)
while (x!=0) {                           (2)
  if (x<0)                               (3)
        print "x เป็นค่าติดลบ "; (4)
  else
    if (x%2==0)                          (5)
        print "x เป็นเลขคู่ "      (6)
    else
        print "x เป็นเลขคี่ "      (7)
  read x;                                (8)
}
```

ขั้นตอนที่ 1 วาด **Flow graph**



Basis path testing

ขั้นตอนที่ 2 คำนวณหาจำนวนเส้นทางอิสระ (Independent Path)



การทดสอบเส้นทางมูลฐาน ต้องทราบจำนวนเส้นทางอิสระ ซึ่งมีเท่ากับค่าความซับซ้อนของปัญหา (Cyclomatic Complexity) แทนด้วย V(G) ซึ่งมีวิธีคำนวณ 3 วิธี

1) McCabe Cyclomatic Complexity ใช้สูตร

$$V(G) = E - N + 2$$

เมื่อ E แทนจำนวนเส้นเชื่อม

N แทนจำนวนโหนด

V(G) = 11 – 9 + 2 = 4

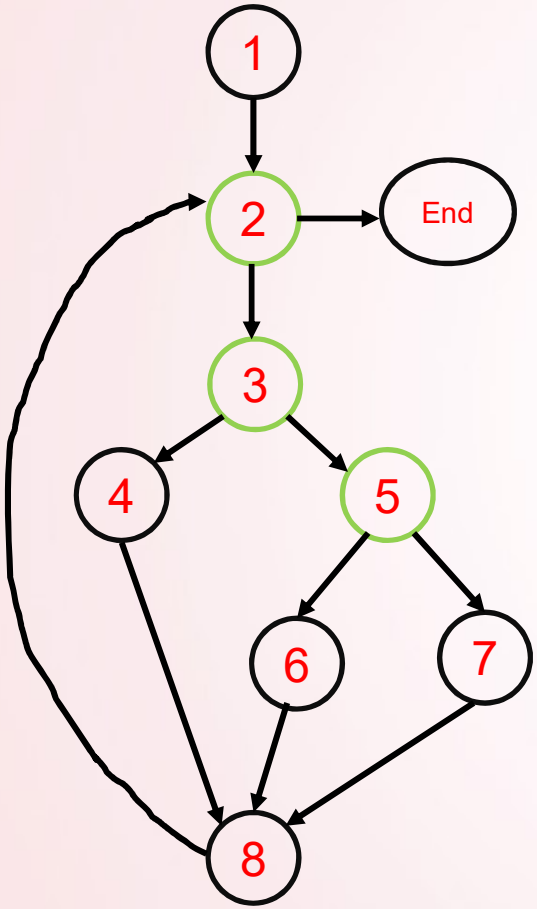
2) จำนวนพื้นที่ปิด + 1

V(G) = 3 + 1 = 4

3) จำนวนเงื่อนไขการตัดสินใจ + 1

V(G) = 3 + 1 = 4

Basis path testing



ขั้นตอนที่ 3 หาเส้นทางอิสระ

เนื่องจากจำนวนเส้นทางอิสระ = V(G) = 4

ดังนั้น เส้นทางอิสระทั้ง 4 เส้น ได้แก่

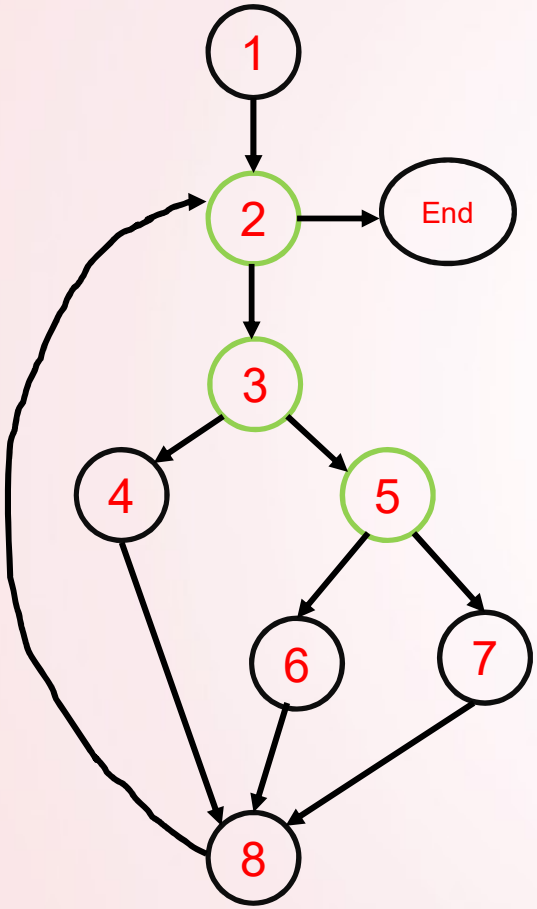
Path1: 1, 2, End

Path2: 1, 2, 3, 4, 8, 2, End

Path3: 1, 2, 3, 5, 6, 8, 2, End

Path4: 1, 2, 3, 5, 7, 8, 2, End

Basis path testing



<u>ขั้นตอนที่ 4</u> ออกแบบกรณีทดสอบที่สอดคล้องกับแต่ละเส้นทาง เพื่อให้ทุกเส้นทางอิสระได้รับการทดสอบ

กรณีทดสอบ	ครอบคลุม	Input	Expected Output
1	1, 2	x=0	
2	1, 2, 3, 4, 8, 2	x=-1 0	x เป็นจำนวนลบ
3	1, 2, 3, 5, 6, 8, 2	x=2 0	x เป็นเลขคู่
4	1, 2, 3, 5, 7, 8, 2	x=1 0	x เป็นเลขคี่

ตัวอย่างที่ 2 Basis path testing

- กรณี for loop ให้แปลงเป็น while loop ก่อน
- คำสั่งที่ทำต่อเนื่องกัน ให้ใช้หมายเลข node เดียวกัน

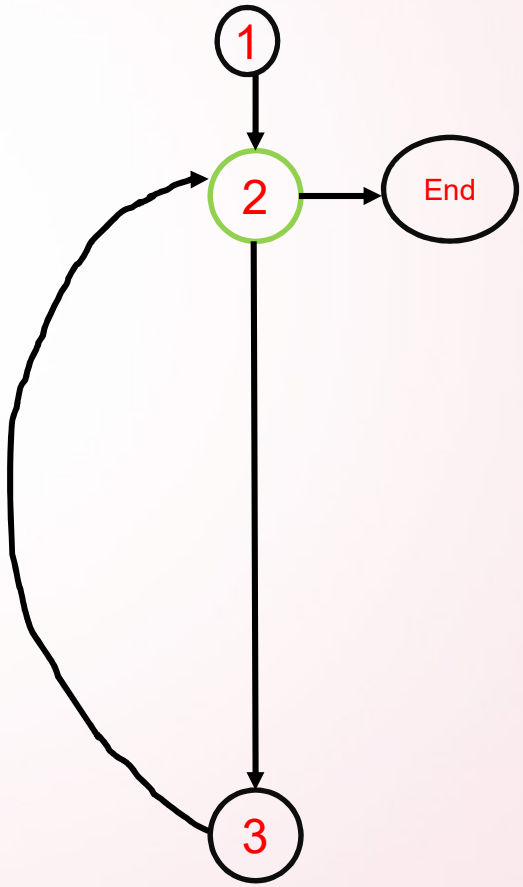
เช่น

```
read N;
for  (i=1;i<=N;i++) {
    print("2 * "+i+ " is ");
    x=2*i;
    println (x);
 }
```

แปลงเป็น while loop

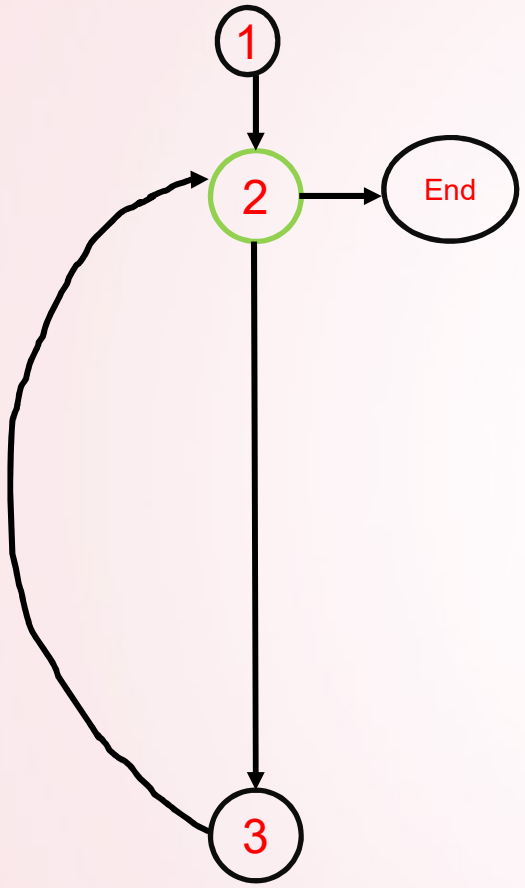
```
read N;
i=1;                    (1)
while(i<=N) (2) {
    print("2 * "+i+ " is ");
    x=2*i;                      (3)
    println (x);
    i++;
 }
```

ขั้นตอนที่ 1 วาด Flow graph



Basis path testing

ขั้นตอนที่ 2 คำนวณหาจำนวนเส้นทางอิสระ (Independent Path)



1) McCabe Cyclomatic Complexity ใช้สูตร

$$V(G) = E - N + 2$$

เมื่อ E แทนจำนวนเส้นเชื่อม

N แทนจำนวนโหนด

V(G) = 4-4 + 2 = 2

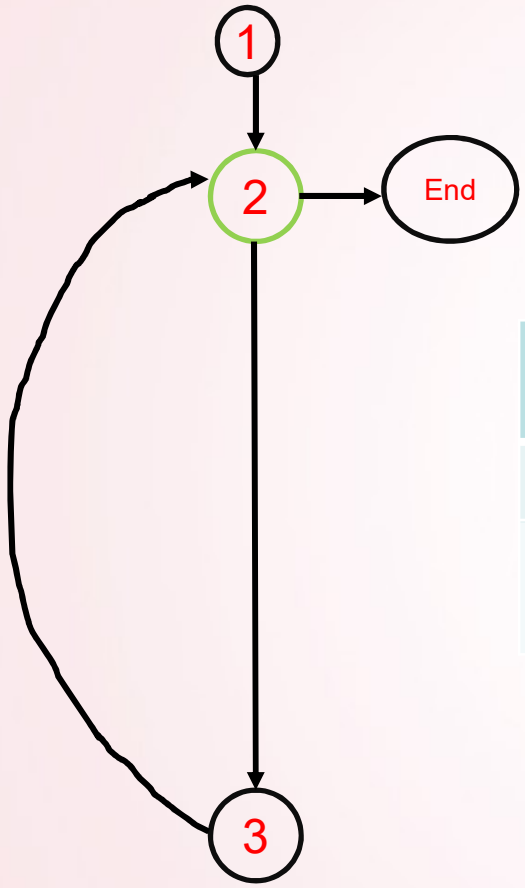
2) จำนวนพื้นที่ปิด + 1

V(G) = 1 + 1 = 2

3) จำนวนเงื่อนไขการตัดสินใจ + 1

V(G) = 1 + 1 = 2

Basis path testing



<u>ขั้นตอนที่ 4</u> ออกแบบกรณีทดสอบที่สอดคล้องกับแต่ละเส้นทาง เพื่อให้ทุกเส้นทางอิสระได้รับการทดสอบ

กรณีทดสอบ	ครอบคลุม	Input	Expected Output
1	1, 2	N=0	
2	1, 2, 3, 2	N=1	2 * 1 is 2

ตัวอย่างที่ 3 Basis path testing

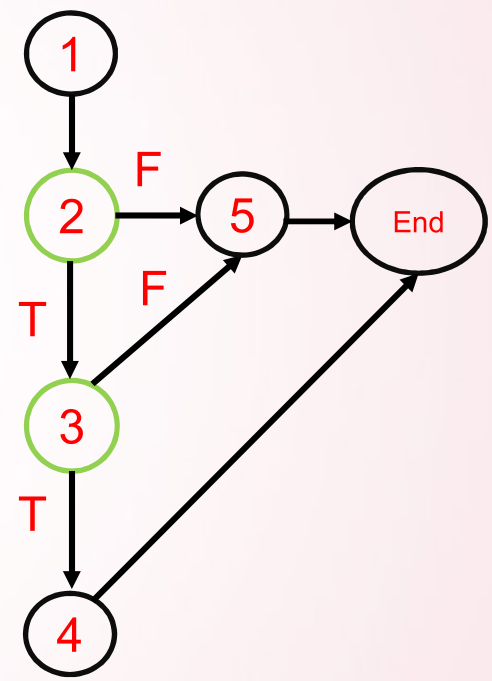
- เงื่อนไขที่มีการ and , or ให้แยก node

เช่น

read A;
read B; (1)

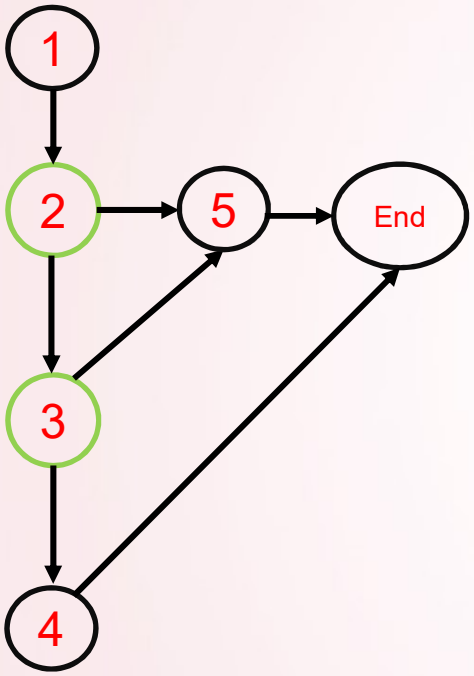
if ((B<A) (2) **&&** **(B>0))** (3)
print "B is a positive number less than A"; (4)
else
print "Oh! No." (5)

ขั้นตอนที่ 1 **วาด Flow graph**



Basis path testing

ขั้นตอนที่ 2 คำนวณหาจำนวนเส้นทางอิสระ (Independent Path)



1) McCabe Cyclomatic Complexity ใช้สูตร

$$V(G) = E - N + 2$$

เมื่อ E แทนจำนวนเส้นเชื่อม

N แทนจำนวนโหนด

V(G) = 7-6 + 2 = 3

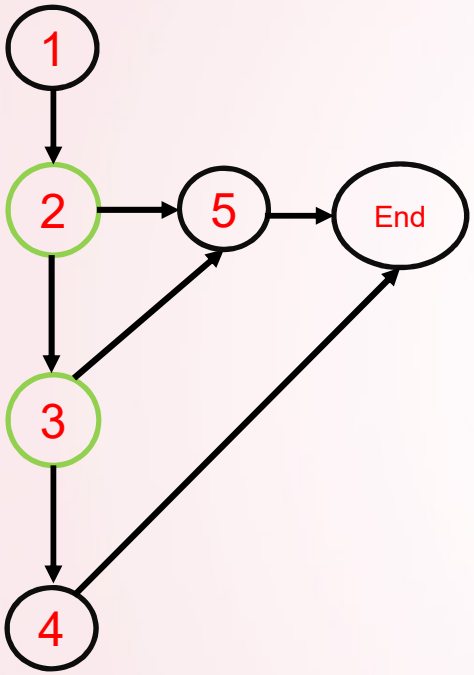
2) จำนวนพื้นที่ปิด + 1

V(G) = 2 + 1 = 3

3) จำนวนเงื่อนไขการตัดสินใจ + 1

V(G) = 2 + 1 = 3

Basis path testing



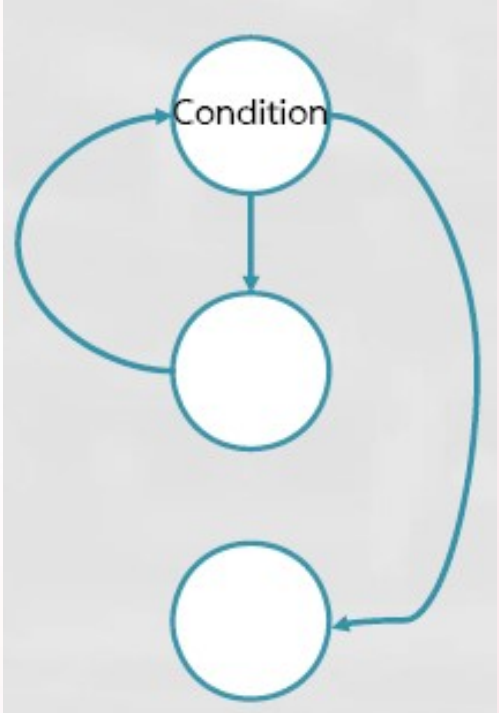
ขั้นตอนที่ 4 ออกแบบกรณีทดสอบที่สอดคล้องกับแต่ละเส้นทาง เพื่อให้ทุกเส้นทางอิสระได้รับการทดสอบ

กรณีทดสอบ	ครอบคลุม	Input	Expected Output
1	1, 2,5	A=-2 B=-1	**Oh! No.**
2	1, 2, 3, 5	A=2 B=0	**Oh! No.**
3	1, 2, 3, 4, 5	A=2 B=1	**B is a positive number less than A**

Loop Testing

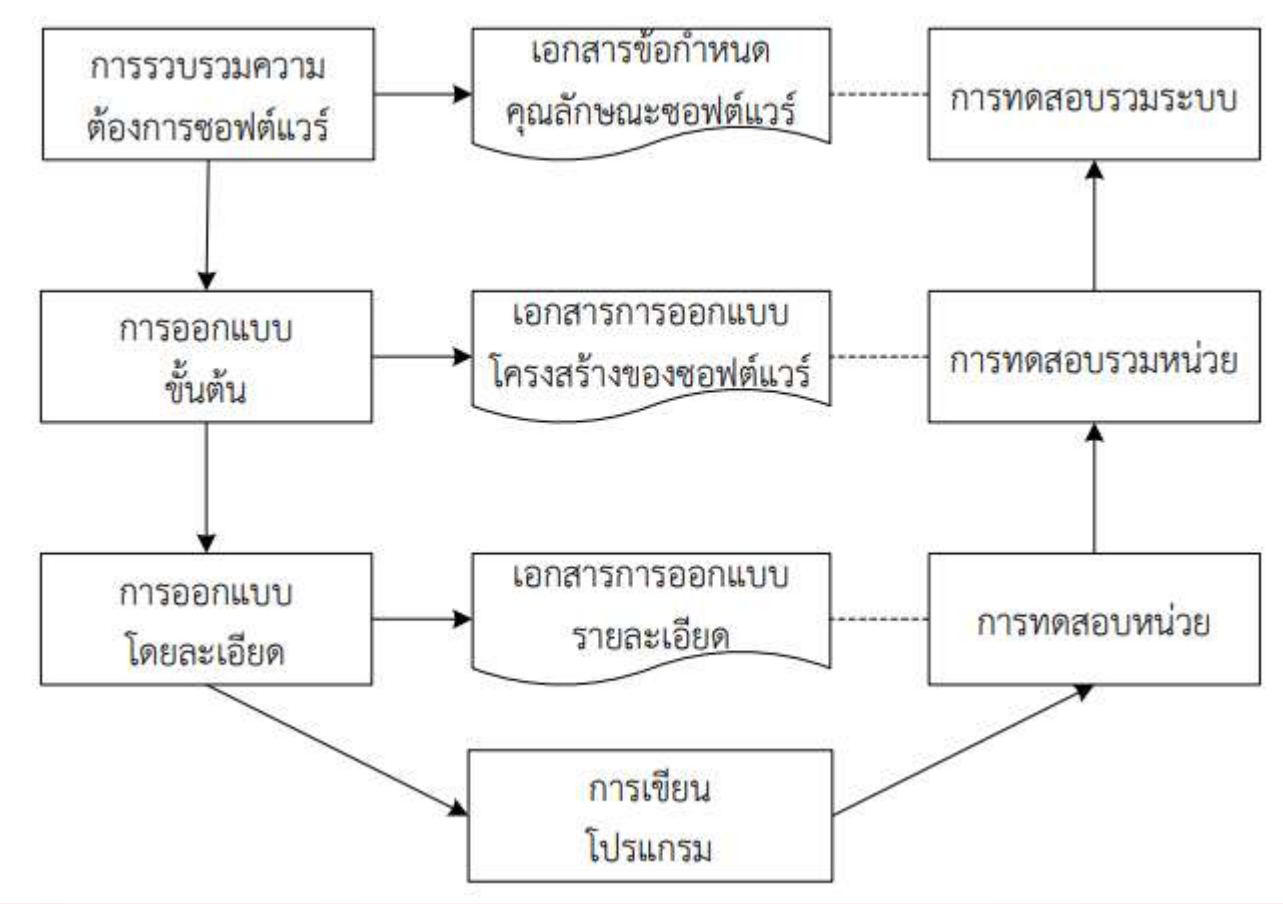
การวนซ้ำแบบง่าย (Simple Loop)

- ข้ามการวนซ้ำนั้นไป
- ทำงานผ่านการวนซ้ำ 1 รอบ
- ทำงานผ่านการวนซ้ำ 2 รอบ
- ทำงานผ่านการวนซ้ำ m รอบ เมื่อ m < n
- ทำงานผ่านการวนซ้ำ n-1, n, n+1 รอบ เมื่อ n เป็นจำนวนรอบที่มากที่สุดที่ยอมให้วนซ้ำ



ระดับของการทดสอบ

ระดับ (Level) ของการทดสอบจะเกี่ยวข้องกับผลลัพธ์ที่ได้จากแต่ละขั้นตอนของกระบวนการพัฒนาซอฟต์แวร์



ระดับของการทดสอบ

การทดสอบรวมระบบ (System testing)

- จะทดสอบตามความต้องการซอฟต์แวร์ที่กำหนดไว้ในเอกสารข้อกำหนดคุณลักษณะระบบ
- จะใช้ข้อมูลความต้องการของลูกค้าในการออกแบบกรณีทดสอบต่างๆ เพื่อทดสอบภาพรวมการทำงานของระบบทั้งระบบ
- ทดสอบว่าระบบสามารถทำงานฟังก์ชันใดได้บ้าง ระบบตอบสนองความต้องการใช้งานทั้งในส่วนของฟังก์ชันการทำงานและในส่วนของประสิทธิภาพของซอฟต์แวร์หรือไม่ โดยอ้างอิงเอกสารข้อกำหนดคุณลักษณะระบบเป็นหลัก
- การทดสอบในระดับนี้มักจะใช้วิธีการทดสอบเชิงฟังก์ชัน (Black box testing)

ระดับของการทดสอบ

การทดสอบรวมหน่วย (Integration testing)

- จะสอดคล้องกับผลลัพธ์ที่ได้จากขั้นตอนการออกแบบขั้นต้น (Preliminary design) ของซอฟต์แวร์ซึ่งเป็นการออกแบบโครงสร้างของระบบ เพื่อให้ทราบว่าซอฟต์แวร์แบ่งออกเป็นระบบย่อยใดบ้าง และในแต่ละระบบย่อยประกอบด้วยฟังก์ชันต่างๆ ที่สัมพันธ์กันอย่างไร
- การทดสอบระดับนี้จะทดสอบการทำงานในแต่ละระบบย่อย
- เน้นทดสอบการทำงานร่วมกันระหว่างฟังก์ชัน เพื่อให้แน่ใจว่าแต่ละระบบย่อยทำงานได้อย่างมีประสิทธิภาพ โดยอ้างอิงเอกสารการออกแบบระบบในส่วนที่เป็นการออกแบบโครงสร้างของ ซอฟต์แวร์

ระดับของการทดสอบ

การทดสอบหน่วย (Unit testing)

- จะสอดคล้องกับผลลัพธ์ที่ได้จากขั้นตอนการออกแบบโดยละเอียด (Detailed design) ซึ่งเป็นการออกแบบขั้นตอนวิธีการทำงานของแต่ละฟังก์ชันโดยละเอียด
- การทดสอบระดับนี้จะทดสอบว่าแต่ละฟังก์ชันทำงานได้อย่างถูกต้องหรือไม่
- จะทำการทดสอบแต่ละฟังก์ชันแยกจากกันโดยอิสระ โดยอ้างอิงเอกสารการออกแบบระบบในรายละเอียดส่วนที่เป็นการออกแบบขั้นตอนวิธีการทำงานของฟังก์ชัน
- การทดสอบในระดับนี้มักจะใช้วิธีการทดสอบเชิงโครงสร้าง (White box testing)

การทดสอบซอฟต์แวร์แบบดั้งเดิม

- การทดสอบซอฟต์แวร์มีหลายระดับ การทดสอบแต่ละระดับมีวัตถุประสงค์ในการทดสอบที่แตกต่างกัน
- วัตถุประสงค์การทดสอบ เช่น ทดสอบเพื่อค้นหาข้อผิดพลาด ทดสอบเทียบกับข้อกำหนดคุณลักษณะระบบ ทดสอบประสิทธิภาพของระบบ ทดสอบเพื่อให้มั่นใจว่าระบบพร้อมใช้งาน เป็นต้น
- การทดสอบแต่ละระดับมีส่วนประกอบของซอฟต์แวร์ที่นำเข้าสู่การทดสอบแตกต่างกัน ในเนื้อหา slide หน้าถัดไปจะกล่าวถึงการทดสอบซอฟต์แวร์ในบริบทของซอฟต์แวร์แบบดั้งเดิม

การทดสอบหน่วย (Unit Test)

- การทดสอบหน่วยจะเน้นการทดสอบหน่วยย่อยที่สุดของซอฟต์แวร์ เช่น ฟังก์ชัน
- วิธีการทดสอบที่เหมาะสมคือวิธีการทดสอบเชิงโครงสร้าง เช่น การทดสอบเส้นทาง และการทดสอบกระแสข้อมูล เป็นต้น
- กรณีทดสอบสำหรับการทดสอบระดับหน่วยสร้างมาจากรายละเอียดขั้นตอนการทำงานหรือขั้นตอนวิธีของแต่ละฟังก์ชันที่ได้ออกแบบไว้
- การทดสอบหน่วยย่อยนี้สามารถทำการทดสอบหน่วยย่อยหลายหน่วยไปพร้อมกันได้

การทดสอบหน่วย (Unit Test)

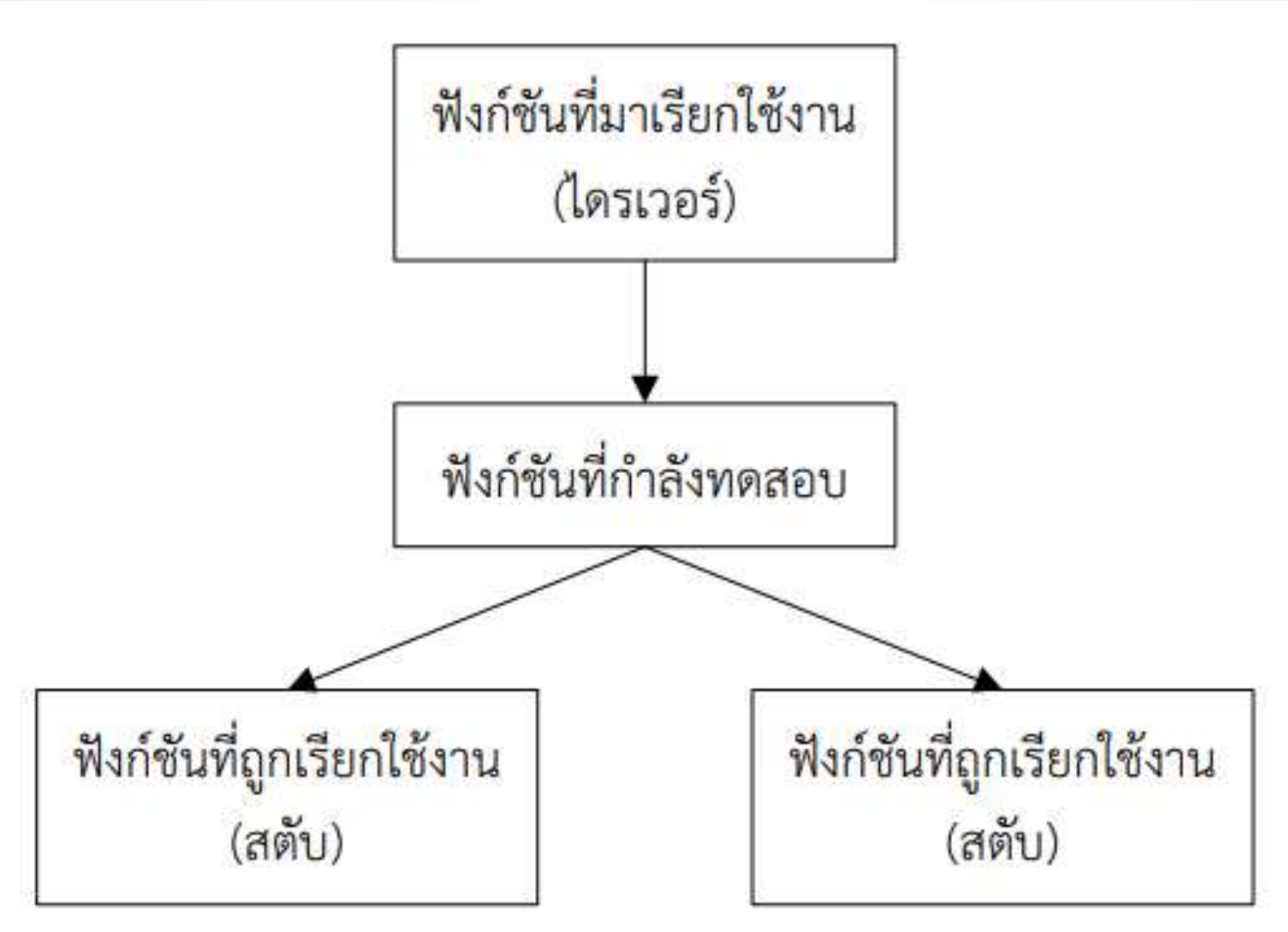
ในการสร้างกรณีทดสอบสำหรับการทดสอบหน่วยมีสิ่งที่ต้องพิจารณาดังต่อไปนี้

1. **ทดสอบช่องทางเชื่อมต่อเข้าสู่ฟังก์ชัน (Interface) เพื่อทดสอบดูว่าสามารถรับข้อมูลเข้าและส่งข้อมูลออกจากฟังก์ชันได้หรือไม่**
2. **ทดสอบโครงสร้างข้อมูลที่ซับซ้อน เช่น ทดสอบการจัดเก็บข้อมูลแบบแถวลำดับและดรรชนีว่าสามารถจัดเก็บข้อมูลได้อย่างถูกต้องตลอดช่วงเวลาที่โปรแกรมยังต้องใช้งานข้อมูลเหล่านั้น**
3. **ทดสอบขอบเขตของเงื่อนไขในการทำงาน เช่น เลือกทดสอบค่าขอบจากแต่ละช่วงของค่าของตัวแปรที่นำมาใช้ในการตัดสินใจ เนื่องจากโปรแกรมมักทำงานผิดพลาดตรงขอบเขตของค่าของตัวแปร**
4. **ทดสอบเส้นทางในการทำงาน และทดสอบการทำงานแบบวนซ้ำ โดยพิจารณาจากกราฟสายงานควบคุมของโปรแกรม**
5. **ทดสอบการทำงานของโปรแกรมในกรณีไม่ปกติ เช่น การรับข้อมูลนำเข้าที่มีชนิดของข้อมูลไม่ถูกต้อง**

การทดสอบหน่วย (Unit Test)

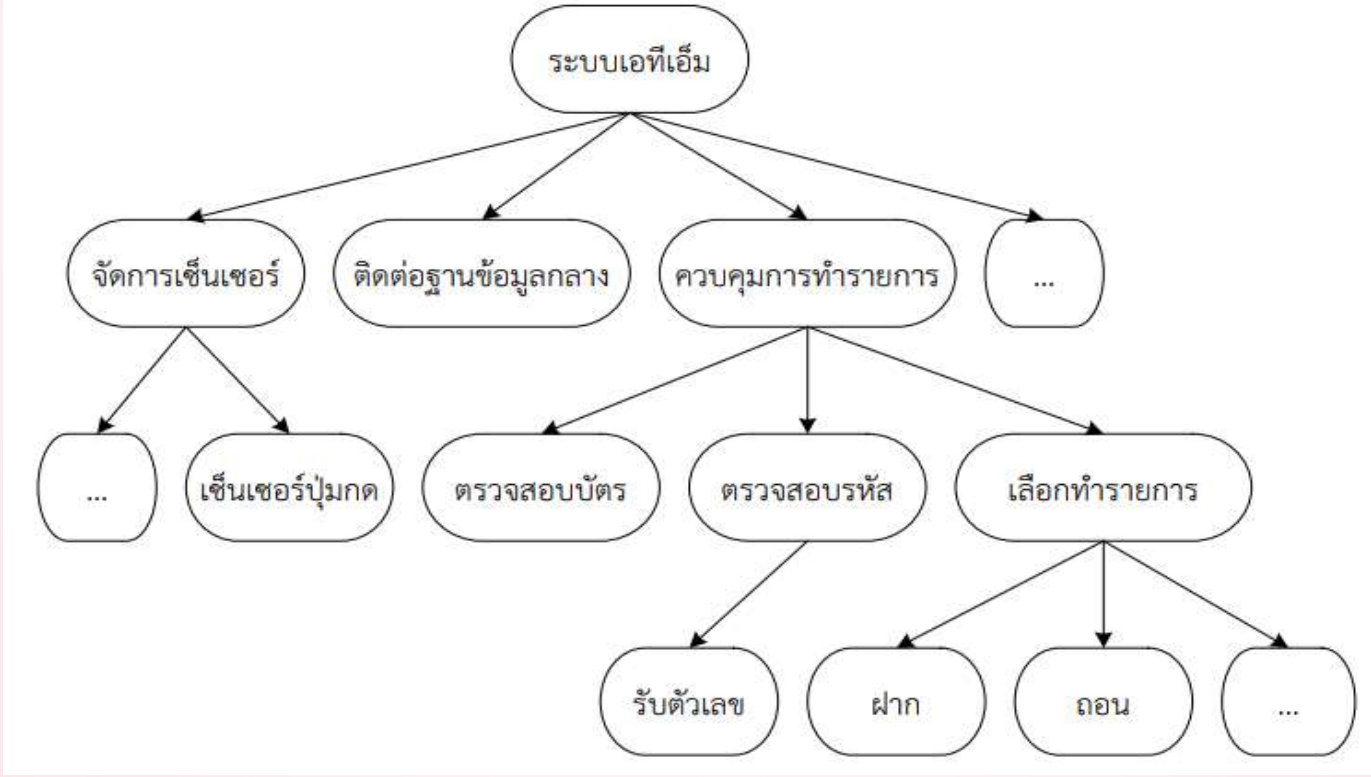
- เนื่องจากฟังก์ชันใดๆ มักจะต้องทำงานร่วมกับฟังก์ชันอื่น โดยในการทำงานอาจเรียกใช้ฟังก์ชันอื่นหรือถูกเรียกใช้โดยฟังก์ชันอื่น
- สภาพแวดล้อมในการทดสอบหน่วยจึงต้องมีตัวแทนของฟังก์ชันที่มาเรียกใช้ฟังก์ชันที่กำลังทดสอบ เรียกว่าไดรเวอร์ (Driver) และมีตัวแทนของฟังก์ชันที่ถูกฟังก์ชันที่กำลังทดสอบเรียกใช้ เรียกว่าสตับ (Stub)
- ไดรเวอร์เป็นโปรแกรมที่ต้องสร้างขึ้นมาให้ทำหน้าที่สร้างตัวแปร กำหนดค่าให้กับตัวแปร และสั่งคำสั่งเรียกใช้ฟังก์ชันที่กำลังทดสอบ
- สตับเป็นโปรแกรมที่ต้องสร้างขึ้นมาให้ทำหน้าที่รับคำสั่งที่ฟังก์ชันที่กำลังทดสอบเรียกใช้ แล้วอาจแสดงผลให้เห็นว่าได้รับค่าข้อมูลอะไรมาหรืออาจส่งค่าบางอย่างกลับไปยังฟังก์ชันที่กำลังทดสอบ

การทดสอบหน่วย (Unit Test)



การทดสอบรวมหน่วย (Integration testing)

- ในการออกแบบโครงสร้างของระบบอาจใช้วิธีการแตกระบบงานออกเป็นฟังก์ชันย่อย (Functional decomposition) เป็นการอธิบายความสัมพันธ์ระหว่างฟังก์ชันย่อยของระบบ
- ลักษณะโครงสร้างของระบบสามารถแสดงโดยใช้โครงสร้างต้นไม้ ดังภาพแสดงตัวอย่างการแตกระบบงานออกเป็นฟังก์ชันย่อยของระบบเอทีเอ็ม



การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยจากบนลงล่าง (Top-down integration)

- การทดสอบจะเริ่มจากฟังก์ชันหลักของโปรแกรมและค่อยๆ เพิ่มฟังก์ชันในระดับล่างของโครงสร้างต้นไม้เข้าสู่กระบวนการทดสอบ
- เริ่มจากฟังก์ชันระบบเอทีเอ็ม ทดสอบการเรียกใช้ฟังก์ชันในระดับล่าง คือจัดการเซ็นเซอร์ ติดต่อฐานข้อมูลกลาง และควบคุมการทำรายการ
- แล้วทดสอบการเรียกใช้ฟังก์ชันในระดับล่างลงไปเรื่อยๆ เช่น ทดสอบฟังก์ชันควบคุมการทำรายการให้เรียกใช้ฟังก์ชันในระดับล่างคือ ตรวจสอบบัตร ตรวจสอบรหัส และเลือกทำรายการ เป็นต้น

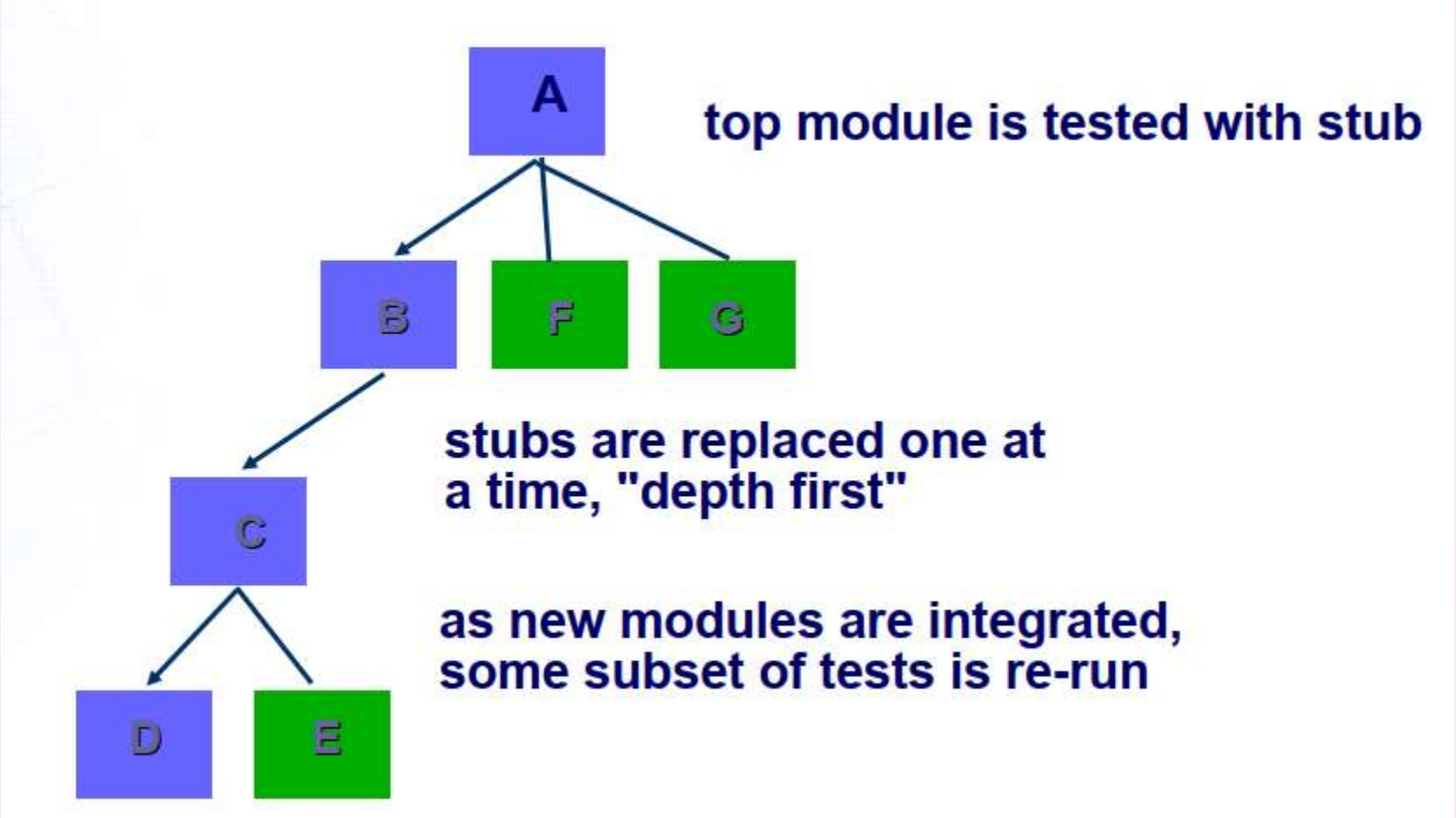
การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยจากบนลงล่าง (Top-down integration)

- การเลือกฟังก์ชันมาทดสอบขึ้นอยู่กับวิธีการในการเลือก หากใช้วิธีค้นหาแนวกว้างก่อน (Breadth-first search) หมายถึงทดสอบหน่วยที่อยู่ในระดับเดียวกันให้หมดก่อนแล้วจึงเลื่อนลงไปยังระดับล่าง
- หากใช้วิธีค้นหาแนวลึกก่อน (Depth-first search) ในแต่ละเส้นทางจะเลือกหน่วยในแนวลึกลงไปจนถึงระดับล่างสุดก่อนแล้วจึงเริ่มเส้นทางใหม่
- ในการทดสอบแต่ละครั้งจะแทนที่รหัสต้นฉบับจริงของฟังก์ชันในระดับล่างด้วยสตับ ซึ่งทำหน้าที่เพียงแสดงผลลัพธ์ให้เห็นว่าฟังก์ชันที่ถูกเรียกใช้งานในระดับล่างได้รับพารามิเตอร์อะไรมาบ้าง ฟังก์ชันในระดับบนที่ทำหน้าที่เรียกใช้งานได้ส่งพารามิเตอร์มาถูกต้องครบถ้วนหรือไม่
- เมื่อทดสอบจนไม่พบข้อผิดพลาดอีกแล้วจึงนำรหัสต้นฉบับจริงมาแทนที่สตับแล้ว ทดสอบซ้ำอีกครั้งทีละฟังก์ชัน

การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยจากบนลงล่าง (Top-down integration) แบบ Depth-first search



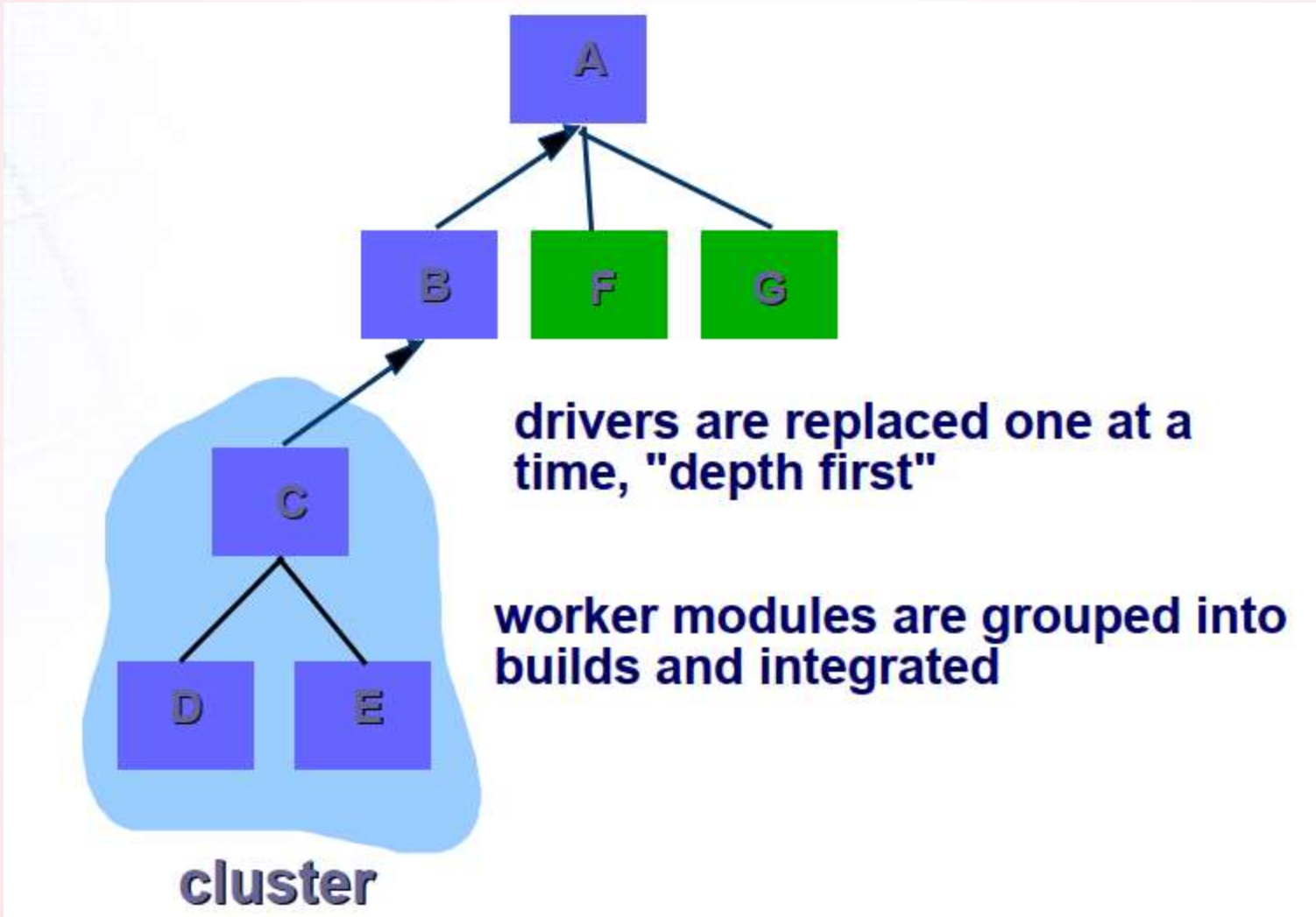
การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยจากล่างขึ้นบน (Bottom-up integration)

- การทดสอบจะเริ่มจากฟังก์ชันระดับล่างของโครงสร้างต้นไม้และค่อยๆ เพิ่มฟังก์ชันในระดับบนเข้าสู่กระบวนการทดสอบ
- เริ่มจากทดสอบว่าฟังก์ชันฝากและถอนถูกเรียกใช้โดยฟังก์ชันเลือกทำรายการได้อย่างถูกต้องหรือไม่
- แล้วเลื่อนการทดสอบขึ้นไปในระดับบนเรื่อยๆ เช่น ทดสอบว่าฟังก์ชันตรวจสอบบัตร ตรวจสอบรหัส และเลือกทำรายการถูกเรียกใช้โดยฟังก์ชันควบคุมการทำรายการได้อย่างถูกต้องหรือไม่
- ในการทดสอบแต่ละครั้งจะแทนที่รหัสต้นฉบับจริงของฟังก์ชันในระดับบนด้วยไดรเวอร์ซึ่งทำหน้าที่กำหนดค่าพารามิเตอร์ที่เกี่ยวข้องและสร้างคำสั่งเพื่อเรียกใช้งานฟังก์ชันในระดับล่าง
- เมื่อทดสอบจนไม่พบข้อผิดพลาดอีกแล้วจึงนำรหัสต้นฉบับจริงมาแทนที่ไดรเวอร์แล้วทดสอบการเรียกใช้งานฟังก์ชันระดับล่างซ้ำอีกครั้งทีละฟังก์ชัน

การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยจากล่างขึ้นบน (Bottom-up integration)



การทดสอบรวมหน่วย (Integration testing)

การทดสอบรวมหน่วยแบบรวมทั้งหมด (Big-bang integration)

- จะนำทุกฟังก์ชันมารวมกันและทดสอบร่วมกันในคราวเดียว
- ข้อเสียของการทดสอบรวมหน่วยแบบรวมทั้งหมดคือเมื่อพบเหตุการณ์ผิดปกติหรือมีความขัดข้องเกิดขึ้นในโปรแกรม จะไม่มีเบาะแสชี้ให้เห็นว่าข้อผิดพลาดน่าจะอยู่ที่ตรงจุดใด
- ต่างจากวิธีการทดสอบรวมหน่วยแบบล่างขึ้นบนหรือบนลงล่าง ที่หากมีความขัดข้องเกิดขึ้นจะสามารถคาดเดาได้ว่าเกิดจากหน่วยที่เพิ่มเติมเข้ามาสู่การทดสอบ

การทดสอบรวมระบบ (System testing)

- การทดสอบรวมระบบจะมีลักษณะใกล้เคียงกับประสบการณ์ในการทดสอบสิ่งต่างๆ ของมนุษย์มากที่สุด
- เช่น การทดลองขับรถมือสองเพื่อทดสอบประสิทธิภาพในการทำงานก่อนที่จะตัดสินใจซื้อ จะเห็นได้ว่ามนุษย์จะใช้วิธีการประเมินว่าเป็นไปตามที่คาดหวังหรือไม่ โดยอาจจะไม่ได้สนใจข้อกำหนดหรือมาตรฐานใดๆ
- ดังนั้นเป้าหมายของการทดสอบรวมระบบคือต้องการแสดงให้เห็นว่าระบบมีพฤติกรรม (Behavior) ในการทำงานที่ถูกต้อง ไม่ใช่ทดสอบเพื่อค้นหาข้อผิดพลาด
- วิธีการในการทดสอบจึงเป็นการทดสอบเชิงฟังก์ชันโดยไม่สนใจโครงสร้างภายในของโปรแกรมว่ามีขั้นตอนการทำงานเป็นอย่างไร
- เน้นทดสอบการทำงานของทั้งระบบเทียบกับเอกสารข้อกำหนดคุณลักษณะระบบ ซึ่งเป็นการทดสอบที่ลูกค้าสามารถเข้าใจและให้ข้อคิดเห็นต่อการทดสอบได้

การทดสอบซอฟต์แวร์เชิงวัตถุ

- ซอฟต์แวร์เชิงวัตถุมีความแตกต่างจากซอฟต์แวร์แบบโครงสร้างจึงต้องพิจารณารูปแบบการเขียนโปรแกรมและปรับเปลี่ยนวิธีการทดสอบใหม่
- เช่น เมท็อด (Method) ภายในซอฟต์แวร์เชิงวัตถุมักจะมีชุดคำสั่งที่สั้นกว่าฟังก์ชันหรือกระบวนงาน (Procedure) ภายในซอฟต์แวร์แบบโครงสร้าง ดังนั้นความผิดพร่องที่เกิดขึ้นกับตรรกะที่ซับซ้อนและเส้นทางควบคุมของโปรแกรมจะไม่ค่อยเกิดขึ้น จึงเป็นข้อดีที่ไม่ต้องลงแรงในการทดสอบมากนัก
- ในทางกลับกันเมท็อดที่มีชุดคำสั่งสั้นๆ นั้นถูกห่อหุ้ม (Encapsulation) ไว้ร่วมกับวัตถุ (Object) ซึ่งจะต้องให้ความสนใจกับการทดสอบปฏิสัมพันธ์ในการเรียกใช้งานเมท็อด
- นอกจากนี้ยังมีคุณสมบัติอื่นๆ เช่น โพลิมอร์ฟิซึม(Polymorphism) ไดนามิกไบดิง (Dynamic binding) เจเนอริก (Generic) และมีการจัดการกับกรณีไม่ปกติ (Exception handling) เป็นจำนวนมาก ทำให้เกิดความผิดพร่องรูปแบบใหม่ที่ต้องให้ความสนใจ

ประเด็นสำคัญในการทดสอบซอฟต์แวร์เชิงวัตถุ

- แนวทางในการทดสอบซอฟต์แวร์เชิงวัตถุมีพื้นฐานคล้ายกันกับการทดสอบซอฟต์แวร์แบบดั้งเดิมหรือที่เรียกอีกอย่างว่าซอฟต์แวร์แบบโครงสร้าง
- การทดสอบจะเริ่มจากหน่วยย่อยที่เล็กที่สุดก่อน จากนั้นรวมหน่วยที่เกี่ยวข้องกันมาทดสอบเป็นกลุ่มๆ แล้วขยายขอบเขตเป็นกลุ่มของหน่วยที่ครอบคลุมมากขึ้นจนกระทั่งรวมทั้งระบบเข้ามาสู่การทดสอบ
- อย่างไรก็ตามซอฟต์แวร์เชิงวัตถุมีความแตกต่างจากซอฟต์แวร์แบบดั้งเดิมหรือแบบโครงสร้างอยู่มากจึงต้องปรับเทคนิคในการทดสอบให้เหมาะสม

คุณสมบัติของซอฟต์แวร์เชิงวัตถุที่ส่งผลกระทบต่อการออกแบบการทดสอบ

1. **พฤติกรรมขึ้นอยู่กับสถานะของวัตถุ:** พฤติกรรมของเมท็อดไม่เพียงขึ้นอยู่กับพารามิเตอร์ที่ส่งมายังเมท็อดแต่ยังขึ้นอยู่กับสถานะของวัตถุอีกด้วย เทคนิคในการทดสอบคือจะต้องพิจารณาสถานะในขณะที่เมท็อดถูกเรียกใช้ เทคนิคการทดสอบแบบเดิมที่ไม่สนใจสถานะของวัตถุจะไม่สามารถค้นพบข้อผิดพลาดที่ขึ้นอยู่กับสถานะได้ นอกจากนี้หากเมท็อดไม่แสดงข้อมูลของคลาสออกมาจะทำให้ทราบสถานะของคลาสได้ยาก
2. **การห่อหุ้ม:** ส่วนประกอบของคลาส (Class) แบ่งเป็นไพรเวท (Private) และพับลิก (Public) สถานะของวัตถุและเมท็อดที่เป็นไพรเวทจะไม่สามารถเข้าถึงได้จากภายนอกคลาส หากต้องการเข้าถึงข้อมูลที่เป็นไพรเวทของคลาสจะต้องกระทำผ่านเมท็อดที่เป็นพับลิกของคลาสเท่านั้น ผลของการรันรหัสต้นฉบับเชิงวัตถุอาจประกอบด้วย ผลลัพธ์ หรือการเปลี่ยนแปลงสถานะของวัตถุ หรือทั้งสองอย่าง ในการทดสอบอาจจำเป็นต้องเข้าไปดูข้อมูลที่เป็นไพรเวทที่ถูกห่อหุ้มไว้เพื่อช่วยในการประเมินว่าเป็นพฤติกรรมที่ถูกต้องหรือไม่

คุณสมบัติของซอฟต์แวร์เชิงวัตถุที่ส่งผลกระทบต่อการออกแบบการทดสอบ

3. การสืบทอด (Inheritance): คลาสลูก (Child class) สามารถสืบทอดตัวแปรและเมท็อดของคลาสบรรพบุรุษมาได้ สามารถเขียนทับ (Overwrite) เพื่อเปลี่ยนแปลงการทำงานของเมท็อดให้แตกต่างจากเมท็อดของคลาสบรรพบุรุษได้ และสามารถสร้างตัวแปรและเมท็อดที่เป็นของคลาสตัวเองได้ การทดสอบพฤติกรรมของเมท็อดที่สืบทอดมาจะต้องพิจารณาผลของการสร้างเมท็อดใหม่กับการโอเวอร์ไรด์ (Override) เมท็อด และต้องแยกการทดสอบเป็น 3 กรณี ได้แก่

3.1 การทดสอบเมท็อดบรรพบุรุษ (Ancestor method) สามารถนำกรณีทดสอบเดิมมารันการทดสอบได้

3.2 การทดสอบเมท็อดของคลาสลูกที่ต้องสร้างกรณีทดสอบขึ้นมาใหม่

3.3 การทดสอบเมท็อดที่ไม่มีความจำเป็นต้องทำการทดสอบซ้ำ

นอกจากนี้ในกรณีที่มีการสืบทอดคุณสมบัติหลายอย่างจะเพิ่มความซับซ้อนในการทดสอบเพราะจะมีสถานการณ์การทดสอบที่เพิ่มขึ้นด้วย

คุณสมบัติของซอฟต์แวร์เชิงวัตถุที่ส่งผลกระทบต่อการออกแบบการทดสอบ

4. **โพลิมอร์ฟิซึมและไดนามิกไบดิง:** การเรียกใช้งานเมท็อดใดๆ ในแต่ละครั้งอาจนำไปสู่เมท็อดที่แตกต่างกันขึ้นอยู่กับสถานะของวัตถุ ดังนั้นในการทดสอบจะต้องทดสอบการไบดิงในรูปแบบ ต่างๆ เพื่อค้นหาความขัดข้องที่ขึ้นอยู่กับการไบดิงโดยเฉพาะ ผู้ทดสอบจะต้องเลือกเซตย่อยจากรูปแบบการไบดิงที่เป็นไปได้ที่ครอบคลุมเพียงพอที่จะค้นพบความผิดพร่องได้

5. **แอบสแตรกคลาส (Abstract class):** แอบสแตรกคลาสไม่สามารถทดสอบได้โดยตรง อาจสร้างขึ้นเพื่อเป็นช่องทางการเชื่อมต่อไปยังคอมโพเนนต์ (Component) ดังนั้นจึงทำการทดสอบโดยอาจไม่ต้องสนใจว่ามีการนำไปใช้งานจริงอย่างไร หรือใช้วิธีการทดสอบคลาสลูกทั้งหมดที่มีแทน แต่ในบางกรณีอาจจำเป็นต้องทดสอบแอบสแตรกคลาสก่อนที่จะสร้างคลาสลูก เช่น ในกรณีที่คลาสลูกไม่ได้สร้างขึ้นมาพร้อมกันทั้งหมดในคราวเดียวหรือไม่ได้สร้างขึ้นโดยวิศวกรซอฟต์แวร์คนเดียวกัน

คุณสมบัติของซอฟต์แวร์เชิงวัตถุที่ส่งผลกระทบต่อการออกแบบการทดสอบ

6. **การจัดการกับกรณีไม่ปกติ:** ในภาษาโปรแกรมเชิงวัตถุมีการใช้การจัดการกับกรณีไม่ปกติเป็นจำนวนมากในการทำงานของโปรแกรม จะมีจุดที่แตกต่างกันที่โปรแกรมเปลี่ยนเส้นทางควบคุมไปเป็นการจัดการกับกรณีไม่ปกติ รวมทั้งการมีคุณสมบัติไดนามิกไบดิง จึงทำให้ต้องทำการทดสอบการจัดการกับกรณีไม่ปกติเสมือนเป็นเส้นทางควบคุมปกติของโปรแกรม

7. **การทำงานแบบภาวะพร้อมกัน (Concurrency):** ภาษาโปรแกรมเชิงวัตถุสามารถสนับสนุนการทำงานของโปรแกรมแบบภาวะพร้อมกัน โดยการสร้างเทร็ด (Thread) ขึ้นมาหลายตัวเพื่อให้ทำงานพร้อมกันได้ ซึ่งการทำงานแบบภาวะพร้อมกันจะก่อให้เกิดความขัดข้องรูปแบบใหม่ขึ้นได้ เช่น การติดตาย (Deadlock) เป็นต้น และทำให้พฤติกรรมของระบบขึ้นอยู่กับการทำงานของโปรแกรม ณ เวลาใดๆ ซึ่งอยู่นอกเหนือการควบคุมของผู้ทดสอบ