BÀI TẬP 2 – CÁC KỸ THUẬT KIỂM THỬ VÀ GỠ RỐI CHƯƠNG TRÌNH ÁP DỤNG CHO JAVA VÀ PYTHON

Nhóm 8

March 2019

Bài tập nhóm môn học
IT 3040 - Kỹ thuật lập trình 2018 - 2019
Giảng viên hướng dẫn: PGS.TS Huỳnh Quyết Thắng

STT	Họ và tên	MSSV
1	Trương Ngọc Giang	20170067
2	Nguyễn Mai Phương	20170106
3	Trương Quang Khánh	20170083
4	Trần Minh Hiếu	20170075
Lớp : KSTN - CNTT K62		

Mục lục

1 White Box Testing

1.1 Kiểm thử hàm Merge

Hàm Merge được sử dụng để trộn hai dãy mà hàm MergeSort đã sắp xếp trước đó thành một dãy tăng dần hoặc giảm dần.
Như vậy, để kiểm tra hàm Merge, hai dãy ta nhập vào phải được hàm MergeSort xử lí trước đó. Ta có thể giải quyết điều này bằng cách dùng Driver, hoặc là ta sẽ cung cấp các dãy được sắp xếp sẵn để kiểm tra hàm Merge.
Trước tiên, ta sẽ kiểm tra các đường cơ bản của hàm Merge.
Các trường hợp đó gồm có như:

- Phần tử của dãy thứ nhất được chọn.
- Phần tử của dãy thứ hai được chọn.
- Dãy thứ nhất được chọn hết phần tử trước.
- Dãy thứ hai được chọn hết phần tử trước.

Với các trường hợp trên, ta có các test tương ứng như sau:

- Dãy 1: 1
  Dãy 2: 2
- Dãy 1: 2
  Dãy 2: 1
- Dãy 1: 6 7 8 9 10
  Dãy 2: 1 2 3 4 5
- Dãy 1: 1 2 3 4 5
  Dãy 2: 6 7 8 9 10
- Dãy 1: 1 3 5 7 9
  Dãy 2; 2 4 6 8 10
- Dãy 1: 1 4 9 10 15
  Dãy 2: 5 7 9 10 11

Tiếp theo, ta sẽ xét đến các trường hợp biên có thể xảy ra:

- Cả hai dãy có các phần tử giống nhau.
- Dãy 1 và dãy 2 gồm các phần tử giống nhau
- Phần tử nhỏ nhất của dãy 1 lớn hơn phần tử lớn nhất của dãy 2 hoặc ngược lại.
- Dãy 1 và dãy 2 không thay đổi gì so với thứ tự của dãy sau khi được sắp xếp.

Với các trường hợp trên, ta đưa ra các test tương ứng là:

- Dãy 1: 1 2 3 4 5
  Dãy 2: 1 2 3 4 5
- Dãy 1: 0 0 0 0 0
  Dãy 2: 1 1 1 1 1
- Dãy 1: 1 1 1 1 1
  Dãy 2: 0 0 0 0 0
- Dãy 1: 1 2 3 4 5
  Dãy 2: 6 7 8 9 10
- Dãy 1: 1 3 5 7 9
  Dãy 2: 2 4 6 8 10
- Dãy 1: 0
  Dãy 2: 1

1.2 Kiểm thử hàm Sort

Trong hàm này, ta sẽ kiểm tra xem quá trình đệ quy có dừng lại hay không, các hàm đệ quy con có phải là phân hoạch của hàm gọi đệ quy không. Như vậy, các trường hợp ta phải kiểm tra là:

- Điểm phân hoạch có đúng không.
- Hàm có dừng lại trong một số trường hợp đặc biệt khi mà không phải gọi đến hàm đệ quy.
- Các trường hợp biên.

Với các trường hợp trên, ta đưa ra các test tương ứng là:

- 1
- 1 2 3 4 5 6 7 8 9 10
- 10 9 8 7 6 5 4 3 2 1
- 0 0 0 0 0 0 0 0 0 0
- 1 0 1 0 1 0 1 0 1 0
- 1 0 0 0 0 0 0 0 0 0
- 1 6 3 5 7 4 8 2 9 10

2 Black-Box Testing

2.1 Các Test case

ID Test case	Mô tả nội dung Test case	Kết quả mong muốn	Kết quả của chương trình
1	Một dãy ngẫu nhiên gồm toàn số dương 1 3 2 5 6 4 8 7 9	Dãy được sắp xếp lại 1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
2	Một dãy đã được sắp xếp tăng dần 1 2 3 4 5 6 7	Dãy không đổi 1 2 3 4 5 6 7	1 2 3 4 5 6 7
3	Một dãy được sắp xếp giảm dần 7 6 5 4 3 2 1	Dãy được sắp xếp lại 1 2 3 4 5 6 7	1 2 3 4 5 6 7
4	Một dãy ngẫu nhiên gồm số âm và số dương -1 -4 2 -3 4 5 6	Dãy được sắp xếp lại -4 -3 -1 2 4 5 6	-4 -3 -1 2 4 5 6
5	Một dãy ngẫu nhiên gồm số âm, số dương và số không 4 5 3 0 -1 -4 -10	Dãy được sắp xếp lại -10 -4 -1 0 3 4 5	-10 -4 -1 0 3 4 5
6	Một dãy gồm số âm và số không -6 -3 -1 -2 0 -100000 -1000000000	Dãy được sắp xếp lại -1000000000 -100000 -6 -3 -2 -1 0	-1000000000 -100000 -6 -3 -2 -1 0
7	Một dãy gồm cả số nguyên số thực: 3 1.5 4 -1.5 3	Bạn nhập không đúng kiểu dữ liệu	arr is not a list of int
8	Một dãy gồm số và chữ cái : 3 4 a b abc	Bạn nhập không đúng kiểu dữ liệu	arr is not a list of int

2.2 Giải thích

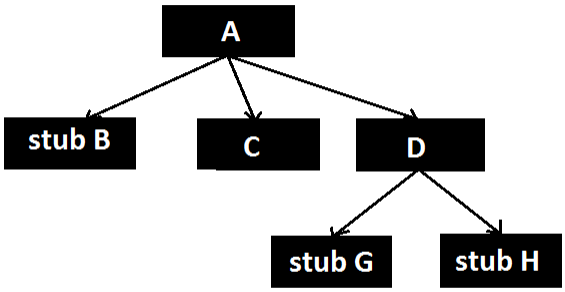
- Test case #1: Kiểm tra tính đúng đắn của chương trình với một dãy ngẫu nhiên.
- Test case #2, #3: Kiểm tra các trường hợp biên của bài toán với 2 dãy suy biến là: Một dãy đã được sắp xếp sẵn xem nó có bị thay đổi gì không, một dãy được sắp xếp giảm dần (ngược hoàn toàn yêu cầu đề bài) để xem tính đúng của chương trình.
- Test case #4, #5, #6: Kiểm tra với dãy gồm các giá trị nằm trải từ âm đến dương (bao phủ miền dữ liệu).
- Test case #6, #7: Kiểm tra chương trình với những input sai (đây là các trường hợp người dùng vô ý nhập sai).

3 Intergration Testing

3.1 Kiểm thử tích hợp dựa trên sự phân tách hệ thống Phân tách hệ thống

hân tách hệ thống được định nghĩa như là một phương pháp mà ở đó hệ thống được chia thành nhiều phần hoặc nhiều hàm độc lập (ở đây ra tạm gọi là units). Chúng có thể chạy tuần tự theo cách trả lời cuộc gọi đồng bộ hoặc chạy đồng thời trên các bộ xử lí khác nhau. Phương pháp này được sử dụng trong quá trình lập kế hoạch, phân tích và thiết kế và tạo ra một hệ thống phân cấp chức năng cho phần mềm.

Một mô hình mà ta dùng để biểu diễn sự phân rã chức năng trong một hệ thống là đồ thị cuộc gọi hoặc cây cuộc gọi. Đồ thị cuộc gọi là đồ thị có hướng trong đó các nút đại diện cho các units và các cạnh thể hiện các sự kiện gọi hoặc yêu cầu tài nguyên cụ thể. Nếu chúng ta loại trừ các cạnh trở lại thì chúng ta có một cây cuộc gọi - cây phân rã

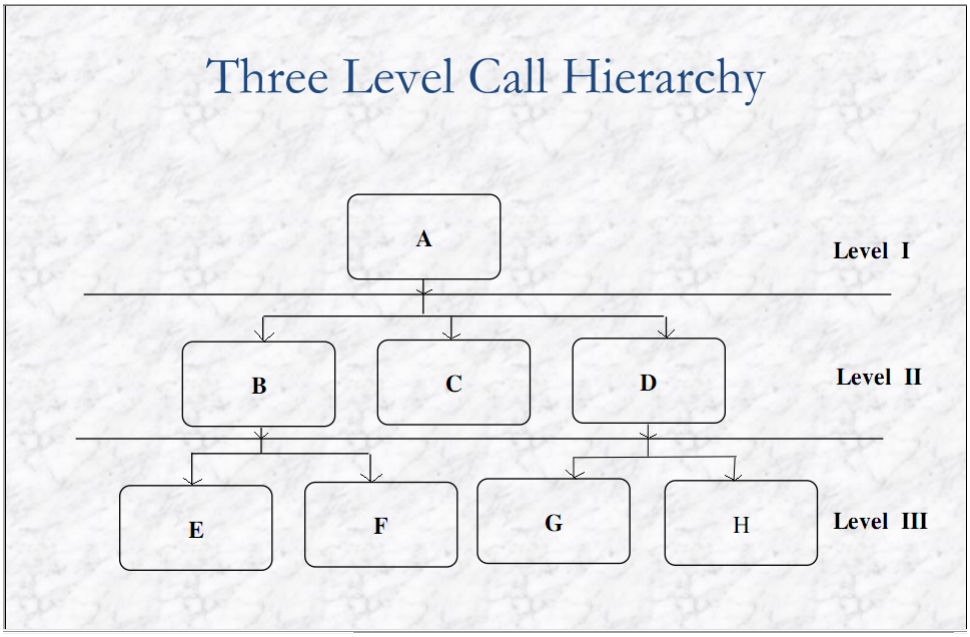


3.1.1 Các chiến lược kiểm thử tích hợp

Toàn bộ hệ thống được xem như một tập hợp các hệ thống con (units) được xác định trong quá trình phân tách hệ thống. Thứ tự mà các hệ thống con được chọn để thử nghiệm và tích hợp xác định chiến lược kiểm thử.

- Big bang integration (Nonincremental)
- Bottom up integration
- Top down integration
- Sandwich testing
- Variations of the above

Ta có một ví dụ về sự phân cấp hệ thống thành 3 cấp:



3.1.2 Big bang integration Testing

Big Bang intergration là một chiến lược thử nghiệm tích hợp trong đó tất cả các đơn vị được liên kết cùng một lúc, dẫn đến một hệ thống hoàn chỉnh.

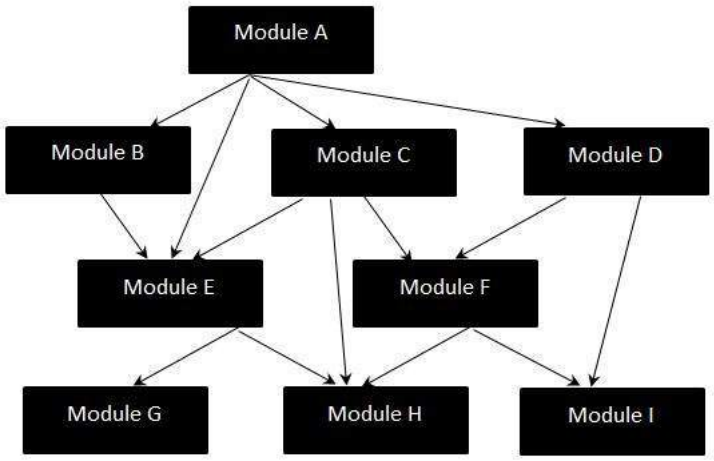
Ưu điểm:

- Tiết kiệm thời gian và công sức.

Nhược điểm:

- Các lỗi xuất hiện ở interfaces của các thành phần được phát hiện ở giai đoạn rất muộn vì tất cả các thành phần được tích hợp trong một lần kiểm thử.
- Rất khó để cô lập các khiếm khuyết được tìm thấy.
- Có khả năng cao sẽ thiếu một số khiếm khuyết quan trọng, có thể xuất hiện trong quá trình đưa vào hoạt động.
- Rất khó để bao quát tất cả các trường hợp để thử nghiệm tích hợp mà không bỏ sót một kịch bản nào.

Ta có một kịch bản kết hợp các units để kiểm thử theo Big Bang intergration:



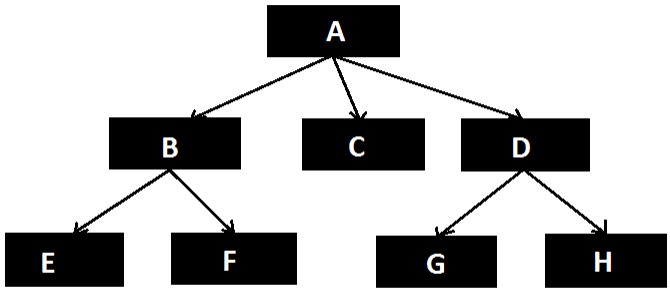
3.1.3 Top-Down Integration Testing

Top-Down intergration testing tập trung vào kiểm tra lớp trên cùng hoặc hệ thống điều khiển trước tiên (main, gốc của cây cuộc gọi).

Quy trình chung trong chiến lược Top-Down intergration testing là dần dần thêm nhiều hệ thống con được gọi bởi các hệ thống con đã được kiểm tra khi thử nghiệm ứng dụng. Làm điều này cho đến khi tất cả các hệ thống con được kết hợp vào thử nghiệm.

Cần có chương trình đặc biệt để thực hiện kiểm tra – stubs. Stub là một chương trình hoặc một hàm mô phỏng chức năng đầu vào-đầu ra của một hệ thống con bị thiếu bằng cách trả lời cho cuộc gọi và trả lại dữ liệu mô phỏng.

Đồ thị ví dụ mình họa về stubs:



Đồ thị trên mô tả rằng các unit A, B, C, D đã có sẵn để tích hợp, trong khi đó B, G, H không có sẵn nên ta thay thế bằng các stub B, stub G, stub H. Stub B, G, H sẽ mô phỏng trả lại kết quả giống với B, G, H cho A và D.

Ưu điểm:

- Các trường hợp thử nghiệm có thể được xác định theo các chức năng của hệ thống (các hàm). Kỹ thuật này cũng có thể được sử dụng cho các đơn vị ở cấp cao nhất.

Nhược điểm:

- Việc tạo ra các stubs có thể khó khăn, đăng biệt là khi tham số được truyền phức tạp.
- Có thể cần một số lượng lớn các stubs, đặc biệt nếu mức thấp nhất của hệ thống chứa nhiều đơn vị chức năng.

Một giải pháp để tránh quá nhiều stubs:

- Kiểm tra từng lớp của hệ thống phân tách riêng lẻ trước khi hợp nhất các lớp.
- Nhược điểm của kiểm tra từ trên xuống sửa đổi: cần cả stubs và drivers.

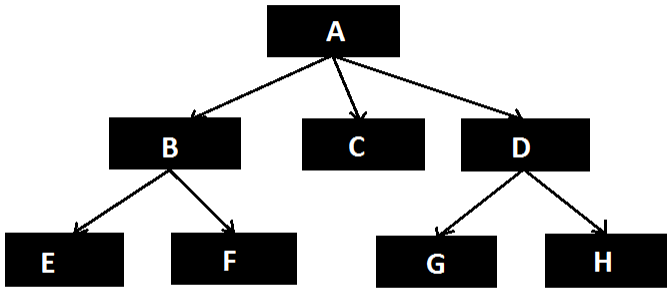
3.1.4 Bottom-Up Integration Testing

Bottom-Up Integration Testing tập trung vào việc thử nghiệm các đơn vị ở mức thấp nhất trước tiên (các đơn vị ở lá của cây phân rã).

Quá trình chung trong Bottom-Up Integration Testing là dần dần tích hợp các hệ thống con điều khiển, yêu cầu các hệ thống con được thử nghiệm trước đó. Điều này được thực hiện lặp đi lặp lại cho đến khi tất cả các hệ thống con được đưa vào thử nghiệm.

Cần có chương trình đặc biệt có tên Test Driver để thực hiện kiểm tra. Test Driver là công việc giả mạo gọi tới hệ thống con đang được thử nghiệm và truyền tham số cho nó.

Ví dụ minh họa một quá trình Bottom-Up Integration Testing cho cây cuộc gọi phía dưới:



- Đầu tiên, kiểm thử lần lượt E, F, G, H sử dụng drivers.
- Kiểm thử B gọi E and F. Nếu xuất hiện lỗi, chúng ra biết rằng lỗi nằm trong B hoặc interface giữa B và E hoặc interface giữa B và F.
- Kiểm thử D gọi G and H. Nếu xuất hiện lỗi, chúng ra biết rằng lỗi nằm trong D hoặc interface giữa D và G hoặc interface giữa D và H.
- Kiểm thử A gọi C. Nếu xuất hiện lỗi, chúng ra biết rằng lỗi nằm trong A hoặc C hoặc interface giữa A và C.
- Kiểm thử A gọi B and D. Nếu xuất hiện lỗi, chúng ra biết rằng lỗi nằm trong interface giữa A và B hoặc interface giữa A và D.

Ưu điểm:

- Hữu ích cho việc tích hợp các hệ thống sau:
  - Các hệ thống định hướng (Object-oriented systems)
  - Hệ thống thời gian thực (Real-time systems)
  - Hệ thống có yêu cầu hoạt động nghiêm ngặt

Nhược điểm:

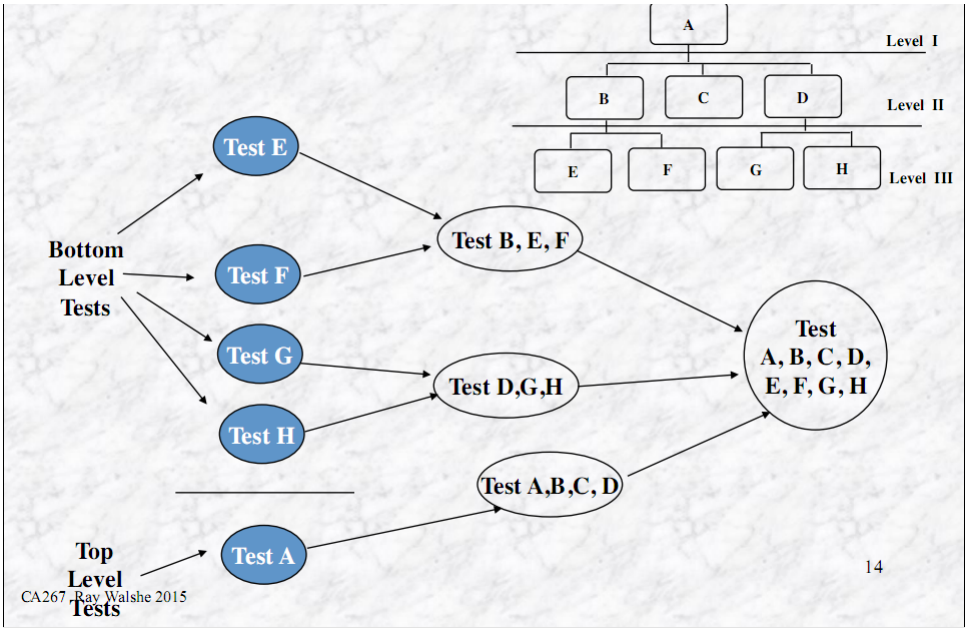
- Chiến lược không tối ưu cho các hệ thống phân rã chức năng: Kiểm tra hệ thống con điều khiển quan trọng nhất cuối cùng.

3.1.5 Sandwich Testing

Sandwich Testing là chiến lược kết hợp top-down strategy và bottom-up strategy. Hệ thống được xem là có ba lớp:

- Lớp giữa (a target layer in the middle)
- Lớp trên mục tiêu (a layer above the target)
- Lớp dưới mục tiêu (a layer below the target)
- Lớp dưới mục tiêu (a layer below the target)

Làm thế nào để bạn chọn lớp mục tiêu nếu có nhiều hơn 3 lớp?
Heuristic: Cố gắng giảm thiểu số lượng stubs và drivers



Ưu điểm:

- Kiểm tra lớp trên cùng và dưới cùng có thể được thực hiện trong song song.

Nhược điểm:

- Không kiểm tra kỹ các hệ thống con riêng lẻ trước khi tích hợp
- Giải pháp: Chiến lược sandwich testing sửa đổi.

3.1.6 Các bước trong intergration testing

1. Dựa trên chiến lược kiểm thử tích hợp, chọn một thành phần sẽ được kiểm tra. Kiểm tra đơn vị tất cả các lớp trong thành phần.
2. Kết hợp thành phần được chọn với nhau. Thực hiện bất kỳ sửa chữa sơ bộ nào nếu cần thiết để thực hiện kiểm tra tích hợp hoạt động (drivers, stubs).
3. Thực hiện kiểm tra chức năng: Xác định các trường hợp và thực hiện tất cả các trường hợp này với thành phần đã chọn.
4. Thực hiện kiểm tra cấu trúc: Xác định các trường hợp kiểm tra và kiểm tra trên thành phần đã chọn.
5. Thực hiện kiểm tra hiệu suất.
6. Giữ hồ sơ của các trường hợp thử nghiệm và các hoạt động thử nghiệm.
7. Lặp lại các bước từ 1 đến 7 cho đến khi toàn bộ hệ thống được kiểm tra.

Mục tiêu chính của kiểm thử tích hợp là xác định các lỗi trong cấu hình thành phần (hiện tại).

3.1.7 Vậy chúng ta nên lựa chọn chiến lược kiểm thử tích hợp nào?

Các yếu tố cần xem xét:

- Số lượng stubs và drivers
- Vị trí của các bộ phận quan trọng trong hệ thống
- Tính khả dụng của phần cứng
- Tính có sẵn của từng phần

- Kế hoạch liên quan

Bottom up approach:

- Tốt cho các thiết kế phần mềm hướng đối tượng.
- Giao diện trình điều khiển thử nghiệm phải phù hợp với giao diện thành phần.
- Các thành phần cấp cao nhất thường quan trọng và không thể bị bỏ qua cho đến khi kết thúc thử nghiệm.
- Phát hiện lỗi thiết kế bị hoãn cho đến khi kết thúc thử nghiệm.

Top down approach:

- Các test case có thể được xác định theo các hàm được kiểm tra.
- Cần duy trì tính chính xác của các stubs.
- Thiết lập stubs có thể khó khăn.

3.2 Kiểm thử tích hợp dựa trên biểu đồ cuộc gọi

Ý tưởng cơ bản là sử dụng đồ thị cuộc gọi thay vì cây phân tách. Biểu đồ cuộc gọi là biểu đồ có hướng, có nhãn. Hai loại kiểm tra tích hợp dựa trên biểu đồ cuộc gọi:

- Pair-wise Integration Testing
- Neighborhood Integration Testi

3.2.1 Pair-Wise Integration Testing

Ý tưởng đằng sau thử nghiệm tích hợp Pair-Wise là loại bỏ nhu cầu phát triển stubs và drivers. Mục tiêu là sử dụng mã thực tế thay vì stubs and drivers. Để không làm giảm quá trình big-bang strategy, ta giới hạn phiên thử nghiệm chỉ ở một cặp đơn vị trong đồ thị cuộc gọi.

Kết quả là chúng ta có một phiên kiểm tra tích hợp cho mỗi cạnh trong đồ thị cuộc gọi.

3.2.2 Neighborhood Integration Testing

Ta định nghĩa hàng xóm của một nút trong đồ thị là tập hợp các nút cách một cạnh so với nút đã cho. Trong biểu đồ có hướng, hàng xóm của một nút có nghĩa là tất cả các nút cha trực tiếp và tất cả các nút nhận nút đó làm cha trực tiếp. Neighborhood Integration Testing giảm số lượng phiên kiểm tra.

Số lượng hàng xóm cho một biểu đồ đã cho có thể được tính là:

$$\text{InteriorNodes} = \text{nodes} - (\text{SourceNodes} + \text{SinkNodes})$$
$$\text{Neighborhoods} = \text{InteriorNodes} + \text{SourceNodes}$$
$$\text{or}$$
$$\text{Neighborhoods} = \text{nodes} - \text{SinkNodes}$$

3.2.3 Ưu điểm và nhược điểm của Call-Graph Integration Testing

Ưu điểm:

- Call graph based integration techniques hướng tới các sự đơn giản.
- Nhằm giúp loại bỏ, giảm nhu cầu stubs và drivers
- Trình tự xây dựng gần hơn.
- Các hàng xóm có thể được kết hợp để tạo ra làng.

Nhược điểm:

- Bị vấn đề cách ly lỗi, đặc biệt là với các hàng xóm xa.
- Các nút có thể xuất hiện nhiều hàng xóm.

3.3 Kiểm thử tích hợp dựa trên đường dẫn

Trong phương pháp, chúng ta sẽ tiếp cận kiểm thử tích hợp từ một hướng mới. Ở đây ta sẽ cố gắng kết hợp cách tiếp cận cấu trúc và chức năng. Cuối cùng, thay vì kiểm tra các interfaces (structural), ta sẽ kiểm tra các interaction (behavioural). Phương pháp tiếp cận path-based integration mà ta sẽ thảo luận là MM-Path.

Trong MM-Path, ta sẽ theo dõi quá tình thực thi của tất cả các module execution paths và messages trong hệ thống. Ta có 2 loại nút mới:

3.4 MM-Path based Integration

MM-Paths diễn tả các hành động với các inputs và outpust, do đó cho chúng ta thấy hệ thống ở functional level. Cách xây dựng MM-Path graph được dưa trên cách xác định các modules và messages cho phép chúng ta làm việc trên hệ thống ở cấp độ cấu trúc.

MM-Paths được thể hiện bằng một đồ thị có hướng trong có các nút là module execution paths và các cạnh là các messages và returns từ unit này sang unit khác. Các đường dẫn đơn vị chéo hiển thị các đường dẫn thực thi có thể, cho phép chúng ta kiểm tra kĩ hơn hành vi của hệ thống.

Để có thể áp dụng MM-Path based integration, trước tiên ta phải tìm các hướng dẫn cho phép chúng ta giới hạn độ sâu của MM-Path. 2 tiêu chí là:

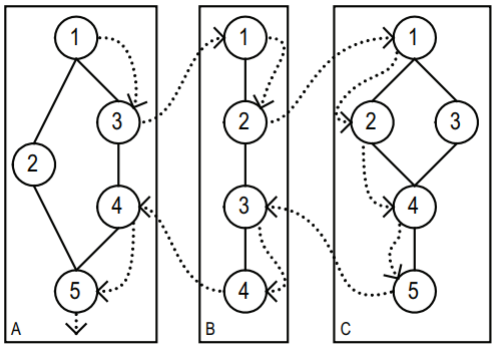
- Message Quiescence
- Data Quiescence

Message quiescence là khi chúng ta đến một unit mà nó không gửi message. Data quiescence đạt được khi chúng ta chấm dứt một tập hợp thực thi với việc tạo dữ liệu được lưu trữ nhưng không được sử dụng ngay lập tức

Với các định nghĩa này, chúng tôi sẽ có thể thực hiện một thử nghiệm tích hợp kết hợp cả phương pháp tiếp cận cấu trúc và chức năng, cho phép kết hợp chặt chẽ hơn với hành vi hệ thống thực tế. Tránh những hạn chế từ cách tiếp cận dựa trên cấu trúc và cho phép chúng ta có sự chuyển đổi suôn sẻ hơn từ kiểm thử tích hợp sang kiểm tra hệ thống (trong đó việc sử dụng các luồng hành vi là mong muốn).

Tuy nhiên, để sử dụng tích hợp dựa trên MM-Path, chúng ta phải nỗ lực nhều hơn để xác định các MM-Path, có thể được bồi thường bằng cách loại stubs và drivers.

Đối với tích hợp dựa trên MM-Path, số lượng phiên kiểm tra tích hợp phụ thuộc vào hệ thống được đề cập.



Một vấn đề chính khi sử dụng phương pháp này là biết có bao nhiêu MM-Paht được yêu cầu để hoàn thành kiểm tra tích hợp. Tập hợp các MM-Path phải đi qua tất cả các đường dẫn source-to-sink paths.

Một số lượng lớn các đường dẫn (hoặc thậm chí vô hạn) gây ra bởi các vòng lặp có thể được giảm bằng các biểu đồ ngưng tụ6 của các biểu đồ chu kỳ có hướng.

Tóm lại, tích hợp MM-Path có các đặc điểm sau:

- Messages được gửi giữa các modules được theo dõi.
- Tập hợp các MM-Path phải bao gồm tất cả source-to-sink paths.
- Điểm không hoạt động là điểm cuối tự nhiên cho MM-path.
- Số lượng integration testing sessions phụ thuộc vào hệ thống được đề cập.

Ưu điểm:

- Kết hợp kiểm tra functional và structural
- Kết hợp chặt chẽ với hành vi hệ thống thực tế
- Không yêu cầu stub hoặc driver

Nhược điểm:

- Cần thêm nỗ lực để xác định MM-path

3.5 Kiểm thử MergeSort với ngôn ngữ py

Chia hệ thống MergeSort thành 2 phần sort - > merge.
Thực hiện Bottom – Up Intergration Testing để kiểm thử hệ thống.

- Bước 1: Kiểm thử merge. Viết 1 driver gọi thực hiện hàm merge cùng với việc cung cấp cho hàm merge đầu vào cần thiết. Chương trình kiểm thử hàm merge sẽ sinh ra các file chứa dữ liệu;
  - ArrLeft, ArrRight : đầu vào hàm merge
  - Output : kết quả trả về của hàm merge
  - Check:
    - trả về true nếu merge thực hiện đúng
    - trả về fale nếu ngược lại

  Nếu xuất hiện lỗi, lỗi sẽ ở trong hàm merge.
- Bước 2: Kiểm thử hàm sort -> merge. Viết 1 driver gọi thực hiện hàm sort cùng với việc cung cấp đầu và cần thiết. Chương trình kiểm thử hàm merge sẽ sinh ra các file chứa dữ liệu:
  - Input: Đầu vào hàm sort là một dãy sinh bất kì
  - Output: Kết quả trả về của hàm sort
  - Check:
    - Trả về true nếu merge thực hiện đúng.
    - rả về false nếu ngược lại

  Nếu xuất hiện lỗi, lỗi sẽ nằm trong hàm sort hoặc trên interface giữa merge và sort.

3.6 Kết luận

- Mặc dù big bang intergration test tiết kiệm thời gian nhưng khó phát hiện các lỗi trong các phần nhỏ.
- Sandwich Testing là một cách hay vì có thế kết hợp các ưu điểm và khắc phục được các nhược điểm của các phương pháp khác.
- Nên kết hợp giữa các phương pháp tùy hệ thống cụ thể.

4 Sử dụng debugger jdb và pdb cho Java và Python

Các ngôn ngữ lập trình thường được cung cấp các bộ công cụ hỗ trợ debug. Các chức năng thường thấy trong các bộ công cụ này bao gồm:

- Breakpoint (điểm dừng): Cho phép lập trình viên dừng chương trình tại điểm xác định và theo dõi các giá trị tại điểm dừng hiện thời, từ đó suy luận ra lỗi. Điểm dừng có thể có hoặc không đi kèm điều kiện kích hoạt (conditional breakpoint).
- Stepping (thực hiện từng bước): Cho phép lập trình viên chạy chương trình theo từng bước một thay vì liên tục, cũng như theo dõi các giá trị ở từng bước.
- Stack Trace (truy vết ngăn xếp): Cho phép lập trình viên theo dõi call stack (ngăn xếp lệnh) để biết được lệnh nào đang gọi lệnh nào vào một thời điểm xác định.

Hai ngôn ngữ Java và Python được cung cấp sẵn hai bộ công cụ debug mặc định là jdb và pdb.

4.1 jdb - Java Debugger

jdb là một bộ công cụ debug sử dụng command line dành cho Java. Nó hỗ trợ khả năng debug code Java chạy trên máy ảo nội bộ hoặc từ xa.

4.1.1 Khởi động một tiến trình jdb

Cú pháp chung để khởi động jdb trên giao diện command line là

```
jdb [ options ] [ class ] [ arguments ]
```

Trong đó:

- options: các cài đặt cho tiến trình hiện tại.
- class: class để thực hiện debug.
- arguments: các tham số truyền vào chương trình, giống như khi khởi tạo máy ảo Java bình thường.

Có hai cách thông dụng để tạo tiến trình jdb. Cách thứ nhất là yêu cầu jdb khởi tạo một máy ảo mới để chạy class cần được debug. Ví dụ:

```
jdb MyClass
```

Khi được khởi tạo theo cách này, máy ảo Java sẽ tạm dừng trước khi bắt đầu thực hiện chỉ thị đầu tiên của chương trình.

Cách thứ hai là kết nối jdb với một máy ảo đang chạy. Máy ảo đang chạy cần phải sử dụng cài đặt:

```
-agentlib:jdwp=transport=dt_shmem,server=y,suspend=n
```

để chuẩn bị các thư viện debug và chỉ thị cụ thể cách thức kết nối cho jdb. Ví dụ: để khởi chạy class MyClass cho việc debug:

```
java -agentlib:jdwp=transport=dt_shmem,
    address=jdbconn,server=y,suspend=n MyClass
```

Khi đó ta có thể kết nối với jdb bằng lệnh:

```
jdb -attach jdbconn
```

Chú ý rằng class MyClass không được truyền vào jdb, vì ta đang không khởi tạo máy ảo mới.

4.1.2 Các lệnh jdb cơ bản

Nếu ta bắt đầu tiến trình jdb bằng cách khởi tạo máy ảo mới, ta có thể thiết lập các breakpoint cần thiết trước khi bắt đầu chạy chương trình bằng lệnh run.

Để sử dụng breakpoint, ta có một số cú pháp:

- stop at MyClass:22 (Đặt breakpoint ở dòng thứ 22 của file MyClass).
- stop in java.lang.String.length (Đặt breakpoint ở đầu của hàm java.lang.String.length).
- stop in MyClass.<init> (Đặt breakpoint tại đầu hàm khởi tạo của class MyClass).
- stop in MyClass.<clinit> (Đặt breakpoint tại đầu hàm khởi tạo static của class MyClass).

Nếu như một hàm được overload (viết đè), ta phải viết rõ kiểu dữ liệu của các biến truyền vào của overload mục tiêu. Ví dụ:

```
stop in MyClass.myMethod(int,java.lang.String)
```

Lệnh clear cho phép ta xóa bỏ một breakpoint cụ thể (khi truyền vào tham số), hoặc xóa tất cả breakpoint (khi không truyền vào tham số).

Khi gặp breakpoint, lệnh cont yêu cầu chương trình tiếp tục chạy. jdb không hỗ trợ conditional breakpoint.

jdb hỗ trợ stepping thông qua hai lệnh step (chạy tiếp dòng lệnh tiếp theo trong chương trình) và next (chạy tới hết hàm hiện tại trong call stack, nhảy sang hàm tiếp theo).

Để theo dõi các giá trị trong chương trình, jdb cung cấp hai lệnh print và dump. Ngoài việc in ra các biến có trong chương trình, lập trình viên có thể kiểm tra giá trị tính toán bất kì giữa chúng.

Để theo dõi các thread đang chạy, jdb cung cấp lệnh threads để liệt kê danh sách, và thread để đổi thread đang thao tác lên.

Để theo dõi call stack, jdb cung cấp lệnh where.

4.2 pdb - Python Debugger

pdb là một bộ công cụ debug trên nền command line dành cho Python. pdb có thể được chạy trên nền command line giống như jdb, hoặc import trực tiếp vào mã nguồn Python để sử dụng và mở rộng.

4.2.1 Khởi động một tiến trình pdb

Có một số cách để sử dụng pdb:

1) Thông qua Python shell:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
```

2) Bằng cách chạy script pdb.py:

```
python3 -m pdb myscript.py
```

3) Bằng cách kêu gọi từ trong mã nguồn cần debug:

```
import pdb; pdb.set_trace()
```

Hoặc ở trong Python phiên bản 3.7 trở lên:

```
breakpoint()
```

4.2.2 Các lệnh pdb cơ bản

Khi sử dụng pdb dưới dạng command line (cách 1 hoặc cách 2), tương tự như với jdb, chương trình sẽ được tạm dừng trước khi thực hiện chỉ thị đầu tiên, và ta có thể sắp xếp các breakpoint cần thiết trước khi bắt đầu chạy bằng lệnh run.

Cú pháp để thiết lập breakpoint trong pdb là:

```
b(reak) [([filename:]lineno | function) [, condition]]
```

Nếu ta sử dụng tham số lineno, breakpoint được đặt tại dòng được chỉ định. Nếu ta sử dụng tham số function, breakpoint được đặt trước chỉ thị đầu tiên của hàm tương ứng. Filename có thể được sử dụng để chỉ thị rõ ràng tên file đặt breakpoint.

Nếu condition được truyền vào, breakpoint sẽ chỉ được kích hoạt nếu condition trả về true.

Nếu như không có tham số nào được truyền vào, lệnh này sẽ liệt kê tất cả các breakpoint, cùng với số lần breakpoint này đã được kích hoạt, số lần bỏ qua và điều kiện đi kèm (nếu có).

pdb còn hỗ trợ một loại breakpoint đặc biệt là temporary breakpoint (breakpoint tạm thời). Temporary breakpoint sẽ bị loại bỏ ngay vào lần đầu tiên nó được kích hoạt. Cú pháp của temporary breakpoint là:

```
tbreak [([filename:]lineno | function) [, condition]]
```

Các breakpoint có thể bị vô hiệu hóa bằng lệnh disable, hoạt hiệu hóa trở lại bằng lệnh enable, hoặc bị loại bỏ hoàn toàn bằng lệnh clear. Chúng cũng có thể được chỉ thị để bỏ qua một số lần nhất định bằng lệnh ignore, hoặc thay đổi điều kiện kích hoạt bằng lệnh condition.

Tương tự như jdb, khi gặp một breakpoint, pdb sẽ dừng chương trình lại cho tới khi gặp lệnh continue.

Cũng tương tự như jdb, pdb hỗ trợ các lệnh step, next và where để thực hiện chương trình theo từng bước một và để theo dõi call stack.

Để in ra một giá trị trong quá trình debug, ta sử dụng cú pháp:

```
p expression
```

expression ở đây có thể là một giá trị, một biến, một biểu thức hoặc một chương trình con.

4.3 Làm rõ các khái niệm Exception và Assertion

Các ngôn ngữ lập trình hiện đại thường cung cấp hai tính năng bắt lỗi trong quá trình chạy là Exception và Assertion.

- Exception cho phép chương trình báo lỗi (throw exception) trong quá trình chạy. Các exception được throw có thể được bắt lại (catch) bởi chương trình để xử lý tương ứng.
- Assertion cho phép chương trình kiểm tra điều kiện trong quá trình chạy. Nếu điều kiện kiểm tra không được thỏa mãn, chương trình sẽ bị dừng lại và một thông báo lỗi được in ra.

Tuy đều cho phép lập trình viên kiểm tra hoạt động của chương trình và báo lại khi xảy ra lỗi, exception và assertion có vài điểm khác nhau:

- Assertion được sử dụng để kiểm tra trong quá trình viết code và bắt các lỗi không được phép xuất hiện trong chương trình thành phẩm (Ví dụ như lỗi chia cho 0, lỗi tràn mảng, vân vân...). Các ngôn ngữ lập trình thường hỗ trợ việc tự động loại bỏ assertion khi dịch chương trình thành phẩm để tăng hiệu năng.
- Exception được sử dụng để đối phó với các trường hợp ngoại lệ trong hoạt động thực tế của chương trình, do các yếu tố ngoại cảnh gây ra (như lỗi đọc - viết phần cứng, kết nối mạng, input của người dùng không hợp lệ, vân vân...). Exception là một bộ phận của chương trình hoàn chỉnh, và do đó không thể/không cần phải loại bỏ ra khỏi chương trình thành phẩm.

Để sử dụng assertion trong Java, ta sử dụng cú pháp:

```
assert expression1 [: expression2];
```

Trong đó expression1 là điều kiện cần kiểm tra. Nếu như điều kiện này là false, chương trình sẽ báo lỗi và dừng lại. expression2 (nếu có) là thông tin được trả về cùng với thông báo lỗi.

Để sử dụng exception trong Java, ta sử dụng cú pháp

```
throw new (subclass của java.lang.Exception)();
```

để throw exception, và sử dụng cụm try/catch để bắt các exception đã throw.

Cú pháp assertion và exception của Python cũng gần giống với Java, tuy nhiên từ khóa catch của Java được đổi thành except trong Python.