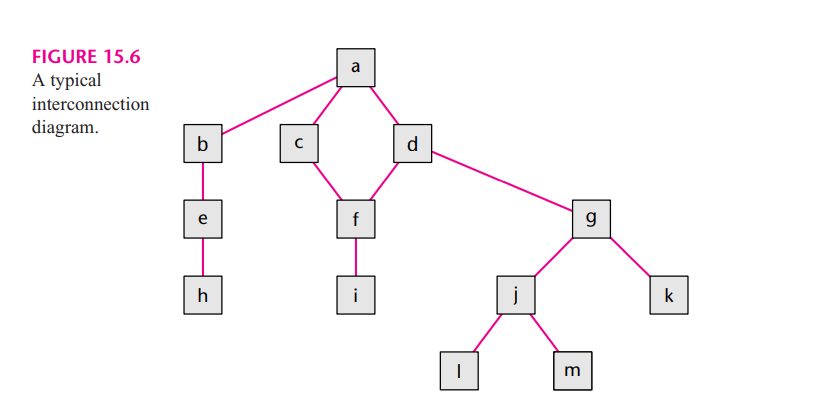
**15.6. Sự tích hợp.**

Hãy xem xét sản phẩm được mô tả trong Hình 15.6. Một trong những cách tiếp cận để tích hợp sản phẩm đó là viết code và test từng mã tạo tác 1 cách riêng biệt, liên kết tất cả 13 mã tạo tác với nhau và kiểm tra toàn bộ sản phẩm. Phương pháp này có 2 vấn đề. Đầu tiên, xét tạo tác a. Nó không thể được test riêng biệt vì nó gọi đến tạo tác b,c,d. Vì thế. Để kiểm thử đơn vị tạo tác a thì tạo tác a, b, c phải được viết dưới dạng **thành phần giả lập(stub)**. Ở trạng thái sơ khai nhất, **stub** là một tạo tác rỗng. Một stub hiệu quả hơn in ra một thông báo giống như cái tạo tác mà phương thức **displayRadarPattern** gọi đến. Hơn hết, một stub nên trả về các giá trị tương ứng với các bộ test được chuẩn bị từ trước.



Xét đến tạo tác h. Để kiểm thử chính nó thì phải dùng đến **Driver(Trình Điều Khiển),** một mã tạo tác gọi đến nó 1 hay nhiều lần, thậm chí kiểm tra các giá trị trả về bởi các tạo tác đang được kiểm thử nếu có thể. Tương tự, kiểm thử tạo tác d cần đến 1 driver và 2 stub. Vì vậy, 1 vấn đề phát sinh với việc triển khai và tích hợp riêng biệt đó là phải tốn công sức vào việc tạo stub và trình điều khiển, những thứ bị vứt bỏ sau khi hoàn thành việc kiểm thử đơn vị.

Khó khăn thứ hai và quan trọng hơn nhiều, sẽ phát sinh khi việc triển khai được hoàn thành quá trình tích hợp bắt đầu đó là việc thiếu sự cách ly lỗi. Nếu toàn bộ sản phẩm được kiểm thử dựa trên các bộ test cụ thể và sản phẩm bị thất bại thì khi đó lỗi có thể nằm ở 13 mã tạo tác(**Code Artifact**) hoặc 13 lớp giao tiếp(**Interface**). Ví dụ trong 1 sản phẩm lớn với 103 mã tạo tác và 108 lớp giao tiếp thì lỗi có thể nằm trong không ít hơn 211 vị trí.

Vì thế mà giải pháp cho cả 2 vấn đề đó là kết hợp cả kiểm thử đơn vị và kiểm thử tích hợp.

**15.6.1 Tích hợp từ trên xuống**

Trong việc tích hợp từ trên xuống, nếu mã tạo tác m Trên(**mAbove**) gửi thông điệp đến mã tạo tác m Dưới(m Below) thì m Trên được triển khai và tích hợp trước m Dưới. Coi sản phẩm trong hình 15.6 được triển khai và tích hợp từ trên xuống thì 1 cách sắp xếp khả thi là **a, b, c, d, e, f, g, h, i, j, k, l,** và **m**. Trước hết tạo tác a được viết và kiểm thử cùng với b, c và được được triển khai theo kiểu thành phần giả lập(**stub**). Thành phần giả lập b tiếp theo được mở rộng thành tạo tác b, dẫn đến tạo tác a, và được kiểm thử với tạo tác e được triển khai theo kiểu thành phần giả lập. Tiếp tục triển khai và tích hợp theo cách này cho tới khi tất cả tạo tác được tích hợp hết vào sản phẩm. Cách sắp xếp từ trên xuống khả thi khác là **a, b, e, h, c, d, f, I, g, j, k, l** và **m**. Với cách sắp xếp này, các phần của việc tích hợp có thể diễn ra 1 cách song song. Sau khi a được viết và kiểm thử, một lập trình viên có thể dùng tạo tác a to triển khai và tích hợp b,r và h trong khi lập trình viên khác có thể dùng a để làm việc song song với c, d, f và i. Một khi d và f được hoàn thiện, một lập trình viên thứ 3 có thể bắt đầu làm việc với g, j, k, l và m.

Giả sử rằng tạo tác a tự nó thực thi chính xác với một test case cụ thể. Tuy nhiên, khi dữ liệu kiểm thử tương tự được gửi sau khi b đã được viết và tích hợp vào sản phẩm, sản phẩm bây giờ bao gồm các thành phần a và b được liên kết với nhau, việc kiểm thử lại không thành công. Lỗi có thể ở một trong hai vị trí, trong tạo tác b hoặc lớp giao tiếp giữa tạo tác a và b. Nói chung, mỗi khi một mã tạo tác m Mới(**mNew**) được thêm vào những gì đã được kiểm thử cho đến nay và một test case thành công trước đó mà lại không thành công, thì lỗi gần như chắc chắn nằm ở mNew hoặc trong các lớp giao tiếp giữa mNew và phần còn lại của sản phẩm. Qua thấy được việc tích hợp từ trên xuống hỗ trợ việc cách ly lỗi.

Một điểm mạnh khác của tích hợp từ trên xuống là các lỗi thiết kế chính xuất hiện sớm. Các tạo tác của một sản phẩm có thể được chia thành hai nhóm, tạo tác logic và tạo tác hoạt động được. **Các tạo tác logic** về cơ bản kết hợp luồng công việc ra quyết định với các khía cạnh kiểm soát của sản phẩm. Các tạo tác logic về cơ bản là những thứ nằm gần gốc trong biểu đồ cấu trúc liên kết. Ví dụ, trong Hình 15.6, các tạo tác a, b, c, d, g, j là các tạo tác logic. Mặt khác, các **tạo tác hoạt động** hình thành các hoạt động thực tế của sản phẩm.

* **Ví dụ**: một tạo tác hoạt động có thể được đặt tên getLineFromTerminal(Lấy dòng trên màn hình terminal) hoặc MeasureTemperatureHeatOfReactorCore(Đo nhiệt đọ của lõi lò phản ứng).

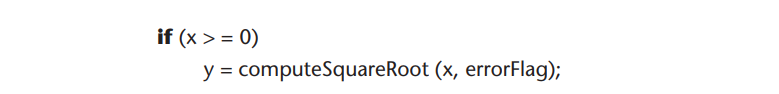
Các tạo tác hoạt động thường được tìm thấy ở các tầng thấp hơn, gần với các lá, của sơ đồ liên kết. Trong hình 15.6, các tạo tác e, f, h, i, k, l và m là các tạo tác hoạt động.

Điều luôn quan trọng là phải viết và kiểm thử các tạo tác logic trước khi viết và kiểm thử hiện vật hoạt động. Điều này đảm bảo rằng bất kỳ lỗi thiết kế lớn nào cũng sẽ xuất hiện sớm. Giả sử toàn bộ sản phẩm được hoàn thành trước khi phát hiện ra lỗi lớn. Các bộ phận lớn của sản phẩm phải được triển khai lại, đặc biệt là các tạo tác logic nắm quyền kiểm soát. Nhiều tạo tác hoạt động có thể được sử dụng lại trong sản phẩm được xây dựng lại; ví dụ, một tạo tác như getLineFromTerminal hoặc MeasureTemperatureHeatOfReactorCore luôn cần thiết bất kể sản phẩm có được tái cấu trúc như thế nào. Tuy nhiên, cách mà các tạo tác hoạt động kết nối với các tạo tác khác trong sản phẩm có thể phải được thay đổi, dẫn đến thực hiện những công việc không cần thiết. Do đó, lỗi thiết kế được phát hiện càng sớm thì công việc sửa chữa và quay lại quá trình phát triển càng nhanh chóng và ít tốn kém. Thứ tự mà các tạo tác được triển khai và tích hợp bằng cách sử dụng chiến lược từ trên xuống về cơ bản đảm bảo rằng các tạo tác logic thực sự được triển khai và tích hợp trước các tạo tác hoạt động, bởi vì các tạo tác logic hầu như luôn luôn là tổ tiên của các tạo tác hoạt động trong biểu đồ liên kết. Đây là một điểm mạnh chính của tích hợp từ trên xuống.

Tuy nhiên, tích hợp từ trên xuống có một điểm yếu: Các tạo tác mã có khả năng tái sử dụng có thể không được kiểm thử đầy đủ, như sẽ được giải thích. Tái sử dụng một tạo tác được cho là đã được kiểm tra kỹ lưỡng có thể ít hiệu quả hơn về chi phí so với việc viết tạo tác đó từ đầu là sai lầm, bởi vì tự cho rằng một tạo tác là đúng có thể dẫn đến kết luận sai khi sản phẩm bị lỗi. Thay vì nghi ngờ một tạo tác tái sử dụng được kiểm thử không tốt, người kiểm tra có thể nghĩ rằng lỗi nằm ở chỗ khác, dẫn đến lãng phí công sức.

Các tạo tác logic có thể có một số vấn đề cụ thể và do đó không sử dụng được trong bối cảnh khác. Tuy nhiên, các tạo tác hoạt động, đặc biệt nếu chúng có sự gắn kết thông tin (Phần 7.2.7), có thể được sử dụng lại trong các sản phẩm trong tương lai và do đó, yêu cầu việc kiểm thử kỹ lưỡng. Thật không may, các tạo tác hoạt động thường là các mã tạo tác ở cấp thấp hơn trong sơ đồ kết nối và do đó không được kiểm tra thường xuyên như các tạo tác cấp trên. Ví dụ: nếu có 184 tạo tác, thì tạo tác gốc được kiểm thử184 lần, trong khi tạo tác cuối cùng được tích hợp vào sản phẩm chỉ được kiểm thử một lần. Tích hợp từ trên xuống làm cho việc tái sử dụng trở thành một công việc rủi hệ quả từ việc kiểm thử các tạo tác hoạt động.

Tình hình càng trầm trọng hơn nếu sản phẩm được thiết kế tốt; trên thực tế, thiết kế càng tốt, các tạo tác càng ít phải kiểm thử kỹ lưỡng. Để thấy được điều này, xét một tạo tác computeSquareRoot. Phần tạo tác này có hai tham số, một số thập thực phấy động x có căn bậc hai cần được xác định và một errorFlag được đặt thành true nếu x âm. Giả sử thêm rằng computeSquareRoot được gọi bởi tạo tác a3 và a3 chứa đoạn code



Nói cách khác, computeSquareRoot không bao giờ được gọi trừ khi giá trị của x không âm; do đó, tạo tác không bao giờ có thể được kiểm tra với các giá trị âm của x để xem liệu nó có hoạt động đúng. Kiểu thiết kế mà tạo tác đang gọi có chứa kiểm tra an toàn được gọi **lập trình phòng thủ**. Kết quả của việc lập trình phòng thủ, các tạo tác hoạt động cấp dưới khó có thể được kiểm tra kỹ lưỡng nếu được tích hợp từ trên xuống. Một giải pháp thay thế cho lập trình phòng thủ là sử dụng thiết kế hướng trách nhiệm(Mục 1.9). Ở đây, các bước kiểm tra an toàn cần thiết được tích hợp vào tạo tác được gọi, chứ không phải là vào cái gọi đến tạo tác. Một cách tiếp cận khác là sử dụng các xác nhận trong tạo tác được gọi.

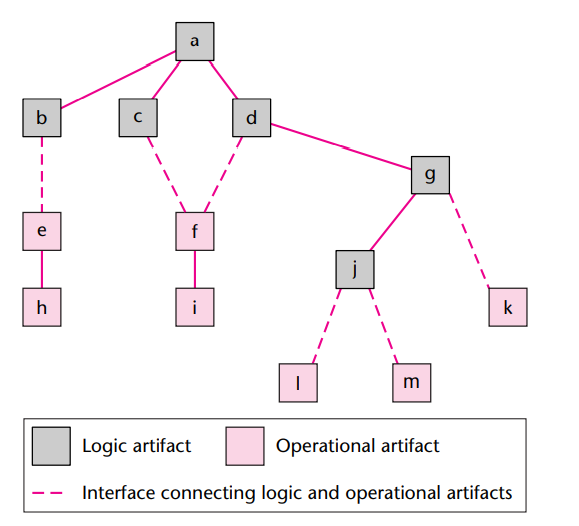
**15.6.2 Tích hợp từ dưới lên**

Trong tích hợp từ dưới lên, nếu tạo tác m Trên(**mAbove**) gửi thông báo đến tạo tác m Dưới(**mBelow**), thì mBelow được thực hiện và tích hợp trước mAbove. Trong Hình 15.6, một thứ tự từ dưới lên khả thi là l, m, h, i, j, k, e, f, g, b, c, d và a. Để có sản phẩm được viết bởi một nhóm, có một thứ tự từ dưới lên tốt hơn như sau: h, e và b được trao cho một lập trình viên và i, f, và c cho một lập trình viên khác. Lập trình viên thứ ba bắt đầu với l, m, j, k và g, sau đó triển khai d và tích hợp phần việc của mình với phần việc của lập trình viên thứ hai. Cuối cùng, khi b, c và d đã được tích hợp thành công, a có thể được triển khai và tích hợp.

Do đó, các tạo tác hoạt động được kiểm tra kỹ lưỡng khi một chiến lược từ dưới lên được sử dụng. Ngoài ra, việc kiểm thử được thực hiện với sự hỗ trợ của trình điều khiển, thay vì bằng cách chống lỗi hay tạo tác được lập trình phòng thủ. Mặc dù tích hợp từ dưới lên giải quyết được khó khăn lớn của tích hợp từ trên xuống và có chung với tích hợp từ trên xuống lợi thế của việc cách ly lỗi, nó lại không may có một nhược điểm khác của riêng nó. Đặc biệt, các lỗi thiết kế lớn được phát hiện muộn trong luồng công việc triển khai. Các tạo tác logic được tích hợp cuối cùng; do đó, nếu có lỗi thiết kế lớn, nó sẽ được xử lý ở cuối luồng công việc triển khai cùng với kết quả là tốn khoản phí lớn cho việc thiết kế lại và viết lại phần lớn sản phẩm.

Do đó, cả tích hợp từ trên xuống và từ dưới lên đều có điểm mạnh và điểm yếu. Giải pháp để phát triển sản phẩm là kết hợp hai chiến lược theo cách như trên để tận dụng điểm mạnh và giảm thiểu điểm yếu của chúng. Điều này dẫn đến ý tưởng về tích hợp kiểu bánh kẹp.

**15.6.3 Tích hợp kiểu bánh kẹp**



Xét biểu đồ ở trên, có 6 tạo tác logic vì thế phải tích hợp từ trên xuống. Có 7 tạo tác hoạt động thế nên phải tích hợp từ dưới lên. T thấy được cả 2 phương án đều không phù hợp với tất cả các tạo tác thế nên phương pháp đó là phân vùng cho chúng. 6 tạo tác logic được tích hợp từ trên xuống và các lỗi thiết kế lớn đều có thể được bắt từ sớm. 7 tạo tác họa động được tích hợp từ dưới lên vì thế mà chúng được kiểm thử kỹ càng, không bị chắn bởi các tạo tác lập trình phòng thủ mà gọi chúng vì thế mà chúng có thể được tái sử dụng một cách thoải mái trong các sản phẩm khác. Khi tất cả các tạo tác đã được tích hợp một cách thích hợp, các lớp giao tiếp giữa 2 nhóm tạo tác được kiểm thử từng cái một. Luôn tồn tại cách ly lỗi trong suốt quá trình này gọi là tích hợp kiểu bánh kẹp(**Sandwich Integration**)

**15.6.4 Việc tích hợp của các sản phẩm hướng đối tượng**

Các đối tượng có thể được tích hợp theo kiểu từ trên xuống hoặc từ dưới lên. Nếu kiểu tích hợp từ trên xuống được chọn, **thành phần giả lập(stub)**  được sử dụng cho mỗi phương thức tương tự với kiểu môđun cổ điển.

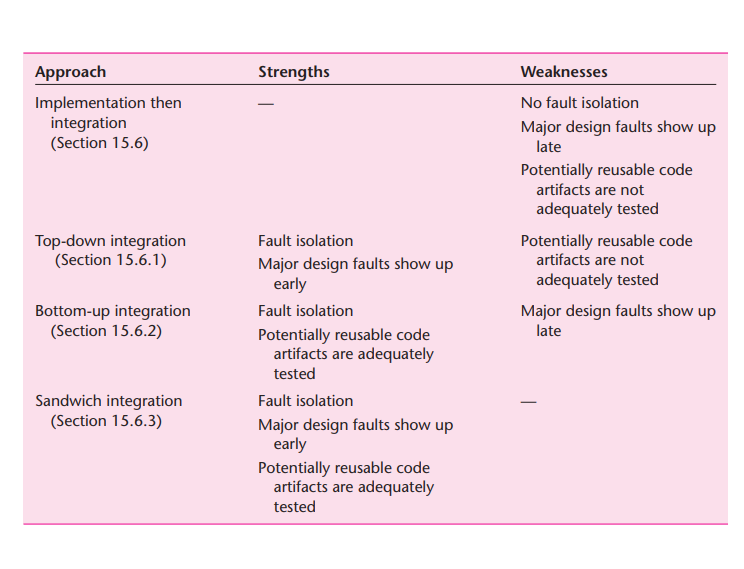
Nếu kiểu tích hợp từ dưới lên được chọn, các đối tượng mà không gửi thông điệp đến các đối tượng khác được triển khai và tích hợp trước. Sau đó, các đối tượng mà có gửi thông điệp đến đối tượng khác mới được triển khai và tích hợp và cứ thế cho đến khi tất cả đối tượng đều được triển khai và tích hợp hết.

Do tích hợp trên xuống và dưới lên đều được hỗ trợ, tích hợp kiểu bánh kẹp cũng có thể được sử dụng. Nếu sản phẩm được triển khai trên ngôn ngữ bán hướng đối tượng như c++, các lớp nói chung đều là tạo tác hoạt động thế nên được triển khai từ dưới lên.

Nhiều hiện vật không phải là lớp là tạo tác logic. Những lớp này được triển khai và được tích hợp theo cách từ trên xuống. Các tạo tác khác là tạo tác hoạt động, vì vậy chúng được triển khai và tích hợp từ dưới lên. cuối cùng, tất cả các tạo tác không phải đối tượng đều được tích hợp với các đối tượng.

Ngay cả khi sản phẩm được triển khai bằng ngôn ngữ hướng đối tượng thuần túy như java, các phương thức của lớp (đôi khi được gọi là phương thức tĩnh) như các phương thức main và phương thức tiện ích thường có cấu trúc tương tự như các mô-đun logic của mô hình cổ điển. Vì vậy, các phương thức của lớp cũng được thực hiện từ trên xuống và sau đó được tích hợp với các đối tượng khác. Nói cách khác, khi triển khai và tích hợp một sản phẩm hướng đối tượng, các biến thể của tích hợp kiểu bánh kẹp được sử dụng.

Bảng so sánh các phương pháp tích hợp



**15.6.5 Quản lý việc tích hợp**

Một vấn đề đối với quản lý là phát hiện ra, tại thời điểm tích hợp, rằng mã tạo tác đơn giản là không khớp với nhau.

**Ví dụ**: giả sử lập trình viên 1 viết đối tượng o1, và lập trình viên 2 viết đối tượng o2. Trong phiên bản của tài liệu thiết kế được sử dụng bởi lập trình viên 1, đối tượng o1 gửi thông báo đến đối tượng o2 truyền bốn đối số, nhưng phiên bản của tài liệu thiết kế được sử dụng bởi lập trình viên 2 nói rõ rằng chỉ có ba đối số được chuyển đến o2. một vấn đề như thế này có thể phát sinh khi một thay đổi được thực hiện chỉ với một bản sao của tài liệu thiết kế mà không thông báo cho tất cả các thành viên của nhóm phát triển. Cả hai lập trình viên đều biết rằng họ đúng; cũng không được chuẩn bị để thỏa hiệp, bởi vì lập trình viên chấp nhận nhượng bộ sẽ phải viết lại phần lớn của sản phẩm.

Để giải quyết những vấn đề này và các vấn đề tương tự về tính không tương thích, toàn bộ quá trình tích hợp nên được điều hành bởi nhóm SQA(**Đảm bảo chất lượng phần mềm**). Hơn nữa, như với thử nghiệm trong các luồng công việc khác, nhóm SQA tốn nhiều sức nhất nếu việc kiểm thử tích hợp được thực hiện không đúng cách. Do đó, nhóm SQA là nhóm phải đảm bảo rằng việc kiểm thử được thực hiện một cách triệt để. Do đó, người quản lý của nhóm SQA phải chịu trách nhiệm về tất cả các khía cạnh của việc kiểm thử tích hợp. Người đó phải quyết định những tạo tác nào được triển khai và tích hợp từ trên xuống và từ dưới lên và chỉ định các nhiệm vụ kiểm thử tích hợp sao cho phù hợp với từng cá nhân. Nhóm SQA, sẽ lập kế hoạch kiểm thử tích hợp trong kế hoạch quản lý dự án phần mềm, chịu trách nhiệm thực hiện kế hoạch đó. Khi kết thúc quá trình tích hợp, tất cả các mã tạo tác sẽ được kiểm tra và kết hợp thành một sản phẩm duy nhất.

**15.7 Luồng công việc triển khai**

Mục đích tổng thể của luồng công việc triển khai đó là triển khai sản phẩm phần mềm đích với ngôn ngữ triển khai đã được lựa chọn. Chính xác hơn thì một phần mềm lớn được phân thành các hệ thống con, sau đó các hệ thống này được triển khai song song bởi các team viết code. Các hệ con này bao gồm các thành phần(**component )** hoặc các mã tạo tác(**code artifact**)

Ngay khi mã tạo tác được viết, lập trình viên kiẻm thử nó, cái này gọi là kiểm thử đơn vị (**unit testing**). Khi lập trình viên đảm bảo được rằng mã tạo tác đã đúng, nó đươc chuyển tới nhóm đảm bảo chất lượng để kiểm thử thêm. VIệc kiểm thử này là một phần của luồn công việc kiểm thử.

**15.8 Case Study quỹ MSG**

**15.9 Luồng công việc kiểm thử: Triển khai**

Nhiều loại kiểm thử khác nhau phải được thực hiện trong suốt luồng công việc triển khai, bao gồm kiểm thử đơn vị, kiểm thử tích hợp, kiểm thử sản phẩm và kiểm thử chấp nhận . Những loại kiểm thử này được thảo luận trong các phần sau.

Như đã chỉ ra trong phần 6.6, các mã tạo tác (mô -đun, lớp) trải qua hai loại kiểm thử: Kiểm tra đơn vị không chính thức được thực hiện bởi lập trình viên trong khi phát triển code, thử nghiệm đơn vị tạo tác và phương pháp được thực hiện bởi nhóm SQA sau khi lập trình viên cảm thấy rằng các tạo tác hoạt động chính xác. Phương pháp kiểm thử này được ghi chép trong các phần 15,10 đến 15,14. Đổi lại, có hai loại cơ bản của phương pháp kiểm thử, kiểm thử không thực hiện, trong đó các tạo tác được xem xét bởi một nhóm và kiểm thử dựa trên thực thi trong đó tạo tác được chạy trên các test case. Kỹ thuật chọn test case bây giờ được mô tả.

**15.10 Lựa chọn test case**

Cách tệ nhất để kiểm thử một mã tạo tác là sử dụng dữ liệu thử nghiệm thiêu tố chức. Người kiểm thử ngồi trước bàn phím, và bất cứ khi nào tạo tác yêu cầu đầu vào, người kiểm tra đưa vào với dữ liệu tùy ý. Như sẽ được chỉ ra, không bao giờ có thời gian để kiểm thử nhiều hơn phần nhỏ nhất trong tất cả các test case khả thi, số lượng có thể dễ dàng vượt quá 10^100. Số ít test case có thể chạy được (khoảng 1000), quá giá trị để lãng phí vào dữ liệu thiếu tổ chức(**haphazard data**). Tệ hơn, có một xu hướng mà máy yêu cầu đầu vào(**input**) để trả về nhiều lần với dữ liệu giống nhau, gây lãng phí nhiều test case hơn. Rõ ràng là các test case phải được xây dựng một cách có hệ thống.

**15.10.1 Kiểm thử các đặc tả so với kiểm thử vỡi code**

Dữ liệu kiểm thử để kiểm thử đơn vị có thể được xây dựng một cách có hệ thống theo hai cách cơ bản. Đầu tiên là

Kiểm thử với cấu hình. Kỹ thuật này còn được gọi là hộp đen(**Black-box**), hành vi(**Behavioral**), dựa trên dữ liệu(**data-driven**), chức năng(**functional**) và kiểm thử dựa trên input/output(**input/output-driven testing**). Theo cách tiếp cận này, bản thân code bị bỏ qua; Thông tin duy nhất được sử dụng để thiết lập test case là tài liệu đặc tả.

Khó khăn khác là kiểm thử với code và bỏ qua tài liệu đặc tả khi lựa chọn test case. Kỹ thuật này còn được gọi là là hộp kính(**glass-box**), hộp trắng(**white-box**), cấu trúc(**Structural**), dựa logic(**logic-driven**) và kiểm thử hướng đường mòn(**path-oriented testing**)

**15.10.2 Tính khả thi của việc kiểm thử với đặc tả**

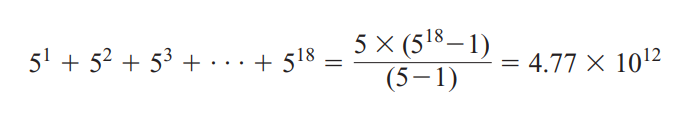
Xét ví dụ sau. Giả sử rằng các đặc tả cho một sản phẩm xử lý dữ liệu nhất định nêu rõ 5 loại hoa hồng và 7 loại chiết khấu phải liên quan chặt chẽ. Kiểm thử mọi sự kết hợp có thể có mỗi hoa hồng và chiết khấu cần đến 35 test case. Sẽ không có lý gì khi nói rằng hoa hồng và chiết khấu được tính trong hai mã tạo tác hoàn toàn riêng biệt và vì thế có thể được kiểm tra độc lập — trong kiểm tra kiểu hộp đen, sản phẩm được coi như một hộp đen và cấu trúc bên trong của nó do đó hoàn toàn không liên quan.

Ví dụ này chỉ chứa hai yếu tố, hoa hồng và chiết khấu, tính đến 5 và 7 giá trị khác nhau. Bất kỳ sản phẩm thực tế nào cũng có hàng trăm, hàng nghìn nhân tố khác nhau. Thậm chí nếu chỉ có 20 nhân tố, mỗi cái lấy 4 giá trị, có tổng cộng 4^20 hay 1.1\*10^12 test case phải được kiểm thử.

Nếu mỗi lập trình viên mỗi 30s có thẻ tạo, chạy và kiểm thử test case thì sẽ mất cả triệu năm.

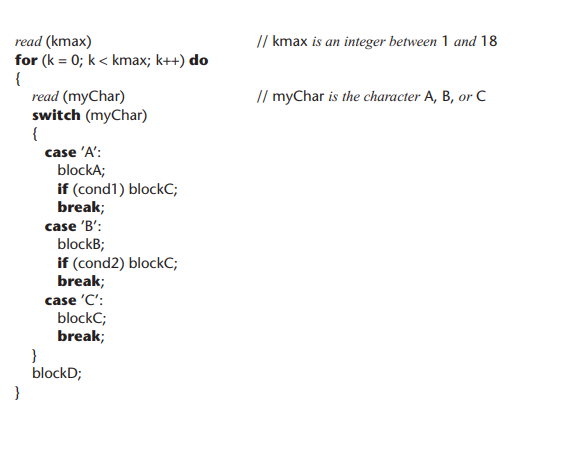
**15.10.3 Tính khả thi của việc kiểm thử với code**

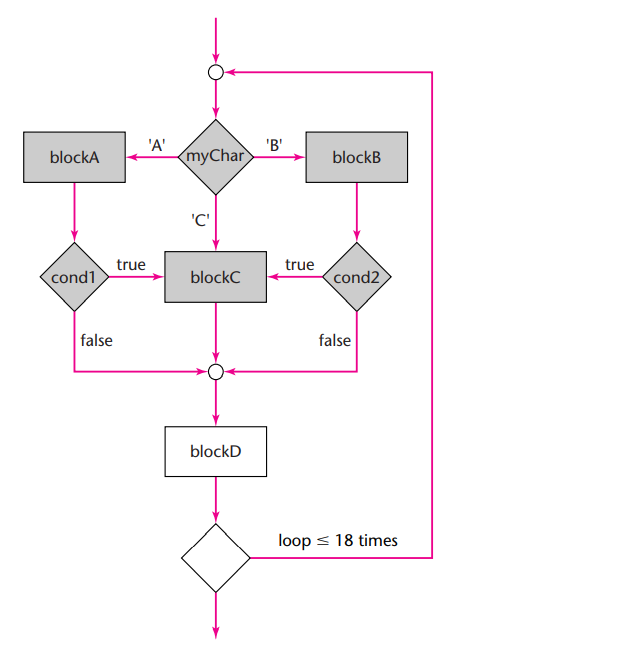
Phương thức chung của kiểm thử code yêu cầu mỗi đường qua mã tạo tác được xử lý ít nhất 1 lần. Flow chart có 10^12 đường. 5 đường khả thi đi quả nhóm 6 hộp màu tối ở trung tâm. Tổng số đường qua flowchart là:



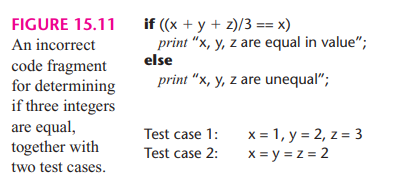
Nếu có từng này đường mà flowchart có chứa một vòng lặp thì số đường qua mã tạo tác sẽ rất nhiều và phức tạp

* Nói chung số đường kiểm thử phương pháp này tương đương với kiểm thử vắt kiệt trong phương pháp kiểm thử đặc tả.

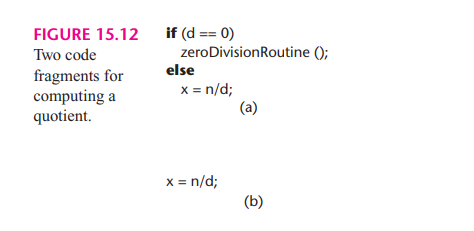




Việc kiểm thử với code không đáng tin ở ví dụ 15.11, đoạn code được viết để kiểm tra xem x, y, z có bằng nhau hay không. Với 2 test case có sẵn thì sẽ ko phát hiện được lỗi, nếu dùng test case như x=2, y = 1, z=3 thì lỗi sẽ hiện ra



Khó khăn thứ 3 với kiẻm thử đường đó là đường chỉ có thể được test nếu nó có sẵn. Ví dụ đoạn code hình 15.12(a) Có 2 đường cần phải được kiêm thử.Với 2 test case d = 0, d != 0 tương ứng. Với đoạn code 15.12(b). Chỉ còn 1 đường và đuòng này chỉ có thể được kiểm thử mà ko tìm ra lỗi.



Các vid dụ này cho thấy rằng việc thử tất cả đường trong sản phẩm là ko đáng tin.

**15.11 Kỹ thuật kiểm thử đơn vị hộp đen**

Kiểm thử vắt kiệt kiểu hộp đen cần đến cả tỷ test case. Để kiểm thử hiệu quả thì phải tạo ra tập các test case nhỏ, quản lý được để tối đa khả năng tìm lỗi đồng thời tối thiểu khả năng test case bị lãng phí khi nhiều test case phát hiện ra cùng 1 lỗi. Mỗi test case phải được chọn để tìm ra lỗi chưa được tìm ra trước đó.

Từ đó chúng ta có kỹ thuật kiểm thử kiểu hộp đen với việc kiểm thử tương đương và phân tích giá trị biên

**15.11.1 Kiểm thử tương đương và phân tích giá trị biên**

Ví dụ: Giả sử các thông số kỹ thuật cho một sản phẩm cơ sở dữ liệu trạng thái rằng sản phẩm phải có khả năng xử lý bất kỳ số lượng bản ghi nào từ 1 đến 16.383 (2^14 - 1). Nếu sản phẩm có thể xử lý 34 bản ghi và 14,870 bản ghi, thì rất có thể là nó sẽ hoạt động tốt, chẳng hạn như 8252 bản ghi. Trên thực tế, cơ hội phát hiện ra lỗi, nếu có, tương đối cao nếu bất kỳ test case nào từ 1 đến 16.383 bản ghi được chọn. Ngược lại, nếu sản phẩm hoạt động chính xác cho bất kỳ 1 test case nào trong phạm vi từ 1 đến 16,383, thì khả năng cao nó sẽ hoạt động cho bất kỳ test case nào khác trong phạm vi đó. Phạm vi từ 1 đến 16,383 tạo thành một **lớp tương đương**, nghĩa là, một tập hợp các test case sao cho bất kỳ một thành phần nào của lớp cũng là một test case tốt như thành phần khác.

Phạm vi được chỉ định số lượng bản ghi mà sản phẩm phải xử lý tạo thành ba lớp tương đương:

-Lớp tương đương 1. Ít hơn 1 bản ghi

-Lớp tương đương 2. Từ 1 đến 16383 bản ghi

-Lớp tương đương 3. Nhiều hơn 16383 bản ghi

Kiểm tra sản phẩm cơ sở dữ liệu bằng cách sử dụng kỹ thuật của lớp tương đương sau đó yêu cầu một test case từ mỗi lớp tương đương được chọn. Test case từ lớp tương đương 2 nên được xử lý đúng cách, trong khi các thông báo lỗi sẽ được in ra với các test case từ lớp 1 và 3.

1 test case gọi là thành công nếu phát hiện ra những lỗi chưa đc tìm ra trc đó. Để tối đa khả năng tìm lỗi đó, ta có kỹ thuật **phân tích giá trị biên.**

Thực nghiệm chỉ ra rằng, khi 1 test case đang ở trên hoặc ở 1 bên của một lớp trương đương được chọn thì khả năng tìm ra lỗi tăng lên. Thế nên khi kiểm thử một sản phẩm lquan đến CSDL, 7 test case sau nên được chọn.

Test case 1. 0 bản ghi: là thành phần của lớp tương đương 1 và kề với giá trị biên

Test case 2. 1 bản ghi: Giá trị biên.

Test case 3. 2 bản ghi: Kề với giá trị biên.

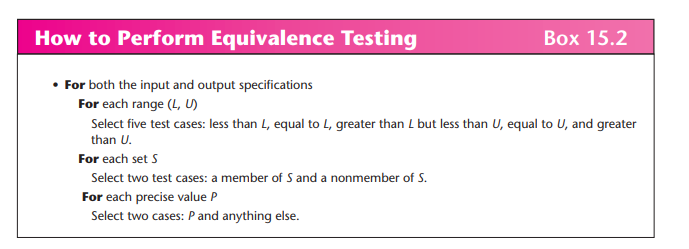
Test case 4. 723 bản ghi: Là thành phần của lớp tương đương 2.

Test case 5. 16,382 bản ghi: Kề với giá trị biên.

Test case 6. 16,383 records: giá trị biên.

Test case 7. 16,384 records: là thành phần của lớp tương đương 3 và kề với giá trị biên

Cách thực hiện kiểm thử tương đương



Ví dụ:

Vào 2008, khấu trừ An sinh xã hội tối thiểu từ bất kỳ thanh toán lương nào được mã thuế Hoa Kỳ cho phép là 0 đô la và tối đa là 6324 đô la, về sau tương ứng với tổng thu nhập là 102.000 đô la. Vì vậy, khi kiểm thử một sản phẩm liên quan đến bảng lương, các test case cho khoản khấu trừ ASXH từ khoản thanh toán nên bao gồm dữ liệu đầu vào mà dẫn đến các khoản khấu từ của 0$ và 6324$. Thêm vào đó, dữ liệu kiểm thử nên được cài đặt sao cho có thể dẫn đến khấu trừ ít hơn 0$ và nhiều hơn 6234$.

Việc sử dụng các lớp tương đương, cùng với phân tích giá trị biên, để kiểm thử cả các đặc tả đầu vào và đặc tả đầu ra là một kỹ thuật sáng giá để tạo ra một tập hợp dữ liệu kiểm thử tương đối nhỏ với khả năng phát hiện ra các lỗi có thể vẫn bị ẩn với các kỹ thuật lựa chọn dữ liệu kiểm thử ít mạnh mẽ hơn được sử dụng.

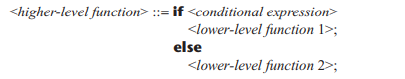
15.11.2 Kiểm thử chức năng

Một cách kiểm thử hộp đen khác là kiểm thử dữ liệu dựa trên chức năng của một mã tạo tác.

Mỗi mục của chức năng được triển khai trong mã tạo tác được xác định. Các hàm bình thường trong các mô đun cổ điển dành cho một kho hàng được vi tính hóa có thể là lấy bản ghi csdl tiếp theo(**get\_next\_database\_record**).

Sau khi xác định được tất cả các hàm trong 1 mã tạo tác, dữ liệu kiểm thử được chia ra để kiểm thử các hàm 1 cách tách biệt. Nếu mã tạo tác chứa cá hàm cấp thấp theo cấp bậc, được liên kết bởi các cấu trúc điều khiển việc lập trình có cấu trúc thì việc kiểm thử chức năng được thực hiện 1 cách đệ quy.

Ví dụ hàm cấp cao có mẫu như sau:



Vì < conditional expression >, < lower-level function 1>, và < lower-level

function 2> được gắn với kiểm thử chức năng thì < higher-level function > có thể được kiểm thử sử dụng phương pháp bao phủ nhánh(**một dạng kỹ thuật hộp kính**). Hàm cấp thấp được kiểm thử sử dụng kỹ thuật hộp đen, hàm cấp cao thì sử dụng kỹ thuật hộp kính.

Thực tế thì các hàm cấp cao không được cấu trúc từ hàm cấp dưới như trên mà các hàm cấp dưới thường được đan xen. Để tìm lỗi thì phải cần đến phân tích chức năng

15.12 Test case hộp đen: Case Study quỹ MSG

Các test case hộp đen:

