Kỷ yếu Hội nghị Khoa học Quốc gia lần thứ IX "Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR'9)"; Cần Thơ, ngày 4-5/8/2016
DOI: 10.15625/vap.2016.00032

KỸ THUẬT KIỂM THỬ HỒI QUI HIỆU QUẢ CHO PHÁT TRIỂN ỨNG DỤNG DI ĐỘNG

Huỳnh Quyết Thắng[1], Nguyễn Đức Mận[2], Nguyễn Thị Bảo Trang[2], Nguyễn Thị Anh Đào[2]

[1]Viện Công nghệ Thông tin và Truyền thông, Trường Đại học Bách khoa Hà Nội
[2]Khoa Đào tạo Quốc tế, Trường Đại học Duy Tân
thanghq@soict.hust.edu.vn, mannd@duytan.edu.vn

TÓM TẮT— *Kiểm thử hồi qui của các ứng dụng di động là một loại thử nghiệm nhằm kiểm tra xem các hành động như cải tiến, các bản vá lỗi hoặc thay đổi cấu hình đã không mang lại hồi qui mới, hoặc lỗi, trong cả chức năng và phi chức năng của một hệ thống ứng dụng di động. Trong quá trình cải tiến ứng dụng di động của các nhà phát triển, các lỗi mới sẽ phát sinh trong miền chức năng và phi chức năng của hệ thống. Kiểm thử hồi qui được sử dụng để đảm bảo rằng chất lượng của ứng dụng di động vẫn được đảm bảo sau khi có bất kỳ thay đổi trong môi trường phần mềm. Tuy nhiên, kiểm thử hồi qui được cho là tốn kém thời gian và chi phí nhưng không được phép bỏ qua hoạt động kiểm thử này. Do đó, vấn đề lựa chọn các bộ kiểm thử (test suite), lựa chọn ca kiểm thử (test-cases), xác định mức độ ưu tiên lựa chọn ca kiểm thử và tự động hóa kỹ thuật lựa chọn ca kiểm thử được xem là các giải pháp nâng cao hiệu quả của kiểm thử hồi qui. Trong nghiên cứu này, chúng tôi đề xuất một kỹ thuật kết hợp lựa chọn ca kiểm thử, xác định mức độ ưu tiên và tối thiểu hóa số lượng ca kiểm thử bao phủ được mã nguồn sửa đổi của chương trình. Kỹ thuật xác lập ưu tiên và giảm thiểu ca kiểm thử được đề xuất trong nghiên cứu này cùng với qui trình thực hiện kiểm thử hồi qui cho ứng dụng di động trong môi trường phát triển linh hoạt mang lại hiệu quả cao cho hoạt động kiểm thử hồi qui về mặt chi phí và thời gian.*

Từ khóa— *Kiểm thử hồi qui, bao phủ mã nguồn, tối ưu bộ kiểm thử, ứng dụng di động.*

I. GIỚI THIỆU

Kiểm thử phần mềm là một hoạt động kiểm tra được tiến hành để cung cấp cho các bên liên quan thông tin về chất lượng của sản phẩm hoặc dịch vụ được kiểm thử. Kiểm thử cung cấp cho doanh nghiệp một quan điểm, một cách nhìn độc lập về phần mềm để từ đó có thể đánh giá và thấu hiểu được những rủi ro trong quá trình triển khai phần mềm. Trong kỹ thuật kiểm thử không chỉ giới hạn ở việc thực hiện một chương trình hoặc ứng dụng với mục đích đi tìm các lỗi phần mềm (bao gồm các lỗi và các thiếu sót) mà còn là một quá trình phê chuẩn và xác minh một chương trình máy tính/ứng dụng/sản phẩm nhằm: (1) đáp ứng được mọi yêu cầu đặc tả khi thiết kế và phát triển phần mềm; (2) thực hiện công việc đúng như kỳ vọng; (3) có thể triển khai được với những đặc tính tương tự; (4) đáp ứng được mọi nhu cầu của các bên liên quan. Tùy thuộc vào từng phương pháp, (Waterfall process, V-model) thì các hoạt động kiểm thử được tiến hành sau khi các yêu cầu được xác định và việc lập trình được hoàn tất nhưng trong môi trường phát triển linh hoạt (Agile) thì việc kiểm thử được tiến hành liên tục trong suốt quá trình xây dựng phần mềm. Như vậy, mỗi một phương pháp kiểm thử bị chi phối theo một quy trình phát triển phần mềm nhất định. Kiểm thử không thể xác định hoàn toàn được tất cả các lỗi bên trong phần mềm [1]. Thay vào đó, nó so sánh trạng thái và hành vi của sản phẩm với các nguyên tắc hay cơ chế để phát hiện vấn đề. Các nguyên tắc này có thể bao gồm (nhưng không giới hạn) [2] các đặc tả phần mềm, hợp đồng, sản phẩm tương đương, các phiên bản trước của cùng một sản phẩm, phù hợp với mục đích dự kiến nhằm đáp ứng sự kỳ vọng của người dùng, khách hàng, quy định của pháp luật hiện hành và các tiêu chuẩn liên quan khác. Mỗi sản phẩm phần mềm có một đối tượng phục vụ riêng. Ví dụ, đối tượng của phần mềm trò chơi điện tử là hoàn toàn khác với đối tượng của phần mềm ngân hàng. Vì vậy, khi một tổ chức phát triển hoặc đầu tư vào một sản phẩm phần mềm, họ có thể đánh giá liệu các sản phẩm phần mềm có được chấp nhận bởi người dùng cuối, đối tượng phục vụ, người mua, hay những người giữ vai trò quan trọng khác hay không. Và việc kiểm thử phần mềm là một quá trình nỗ lực để đưa ra những đánh giá này.

Việc kiểm thử cho một sản phẩm phần mềm được thực hiện bởi nhiều phương pháp, kỹ thuật và chiến lược khác nhau. Trong phạm vi của nghiên cứu này, chúng tôi chỉ tập trung vào kỹ thuật kiểm thử hồi qui cho ứng dụng di động được phát triển theo phương pháp linh hoạt (Agile Scrum). Kiểm thử hồi qui tập trung vào việc tìm kiếm lỗi sau khi xảy ra một thay đổi mã chính, thay đổi cấu hình, nền tảng, và thiết bị…[3, 4]. Phương pháp phổ biến của kiểm thử hồi qui bao gồm việc chạy lại những ca kiểm thử trước đó để xác định lỗi cố định trước đây tại sao lại xuất hiện. Độ sâu của kiểm thử phụ thuộc vào các nguy cơ và giai đoạn trong quá trình phát hành các tính năng bổ sung. Chúng có thể được hoàn tất khi thay đổi thêm vào đầu hoặc cuối bản phát hành, cũng có thể có mức độ nguy hiểm thấp khi thực hiện kiểm thử tích cực trên mỗi tính năng. Ví dụ, một ứng dụng đang phát triển khi kiểm tra cho thấy nó chạy tốt các chức năng A, B và C. Khi có thay đổi mã của chức năng C, nếu chỉ kiểm tra chức năng C thì chưa đủ, cần phải kiểm tra lại tất cả các chức năng khác liên quan đến chức năng C, bởi khi C thay đổi, nó có thể sẽ làm A và B không còn làm việc đúng nữa. Thực tế cho thấy kiểm thử hồi qui là một trong những loại kiểm thử tốn nhiều thời gian và công sức nhất [3]. Đối với ứng dụng di động, sự thay đổi cấu hình, đa dạng về cấu hình, đa nền tảng,… sẽ là những thách thức cho việc phát triển sản phẩm và kiểm thử sản phẩm để phù hợp với đặc tính thay đổi và sự đa dạng này [5]. Do đó, việc bỏ qua khâu kiểm thử hồi qui là không được phép vì có thể dẫn đến tình trạng phát sinh hoặc tái xuất hiện những lỗi nghiêm trọng, mặc dù chúng ta nghĩ rằng những lỗi đó hoặc không có hoặc đã được kiểm tra và sửa chữa xong. Phương pháp đơn giản để kiểm thử hồi qui là chạy lại tất cả các ca kiểm thử của phiên bản trước. Tuy nhiên, việc thực thi lại toàn bộ

các ca kiểm thử là rất tốn kém. Các ca kiểm thử trước đó có thể không chạy lại được với phiên bản mới của phần mềm mà không có sửa đổi gì. Một bộ kiểm thử chất lượng tốt phải được duy trì xuyên suốt quá trình phát triển các phiên bản của phần mềm. Những thay đổi trong phiên bản phần mềm mới có thể tác động tới khuôn dạng của đầu vào và đầu ra, các ca kiểm thử có thể cần một sự thay đổi tương ứng để thực thi được. Ngay cả việc sửa đổi một cách đơn giản của cấu trúc dữ liệu, chẳng hạn như việc bổ sung các trường hay sự thay đổi nhỏ của các loại dữ liệu cũng có thể ảnh hưởng các ca kiểm thử trước đây không chạy được nữa. Hơn nữa, một số ca kiểm thử có thể bị lỗi thời, khi các tính năng của phần mềm đã thay đổi hoặc bị loại bỏ khỏi phiên bản mới. Các bộ kiểm thử chất lượng cao có thể được duy trì qua các phiên bản của phần mềm bằng việc xác định và loại bỏ các ca kiểm thử lỗi thời và bằng việc phát hiện, đánh dấu thích hợp các ca kiểm thử dư thừa. Ca kiểm thử đi cùng một đường đi trong chương trình là dư thừa với tiêu chuẩn kiểm thử cấu trúc, nhưng ca kiểm thử cùng một phân hoạch lại là dư thừa với kiểm thử dựa trên lớp tương đương. Việc dư thừa là do nhiều người cùng làm hoặc do chương trình bị thay đổi gây ra. Các ca kiểm thử dư thừa không ảnh hưởng đến tính đúng hoặc sai mà chỉ ảnh hưởng đến chi phí kiểm thử. Chúng không phát hiện được lỗi nhưng làm tăng chi phí kiểm thử nên các ca kiểm thử lỗi thời cần phải được loại bỏ [5].

Trong các phần nghiên cứu tiếp theo, chúng tôi tập trung trình bày một số kỹ thuật kiểm thử hồi qui liên quan đã được nghiên cứu và công bố ở phần II, đề xuất quy trình thực hiện, áp dụng chiến lược thực hiện kiểm thử hồi qui hiệu quả và kỹ thuật kiểm thử hồi qui dựa trên các nghiên cứu cho ứng dụng PC, Web để thực hiện cho ứng dụng di động được trình bày ở phần III, trong phần này chúng tôi cũng trình bày thuật toán tối ưu lựa chọn ca kiểm thử hồi qui mức đơn vị dựa vào sự thay đổi mã nguồn. Kết quả thực nghiệm cho lập trình ứng dụng di động bằng Java và thảo luận được trình bày ở phần IV. Phần V là các kết luận, các hạn chế và hướng nghiên cứu tiếp theo.

II. CÁC KỸ THUẬT KIỂM THỬ HỒI QUI ĐÃ ĐƯỢC NGHIÊN CỨU

Kiểm thử hồi qui được định nghĩa là quá trình kiểm tra lại những phần sửa đổi của phần mềm và đảm bảo rằng không có lỗi phát sinh trong mã nguồn chương trình đã thử nghiệm trước đó. Gọi P là một chương trình hoặc một mô đun, P’ là phiên bản mới chỉnh sửa của P và gọi T là một bộ kiểm thử (test suites) của P. Kiểm thử hồi qui bao gồm việc sử dụng lại T trên P’ và xác định xem có cần thêm trường hợp kiểm thử mới để kiểm thử một cách có hiệu quả mã nguồn hay chức năng mới được thay đổi trong P’ [7, 8, 9]. Các kỹ thuật kiểm thử hồi qui khác nhau bao gồm [6]: (1) thực thi lại lại tất cả các ca kiểm thử; (2) lựa chọn ca kiểm thử để hồi qui (Regression Test Selection –RTS); (3) xác lập ưu tiên cho ca kiểm thử (Test case prioritization-TCP); (4) cách tiếp cận kết hợp.

A. Thực thi lại tất cả các ca kiểm thử

Phương pháp thực thi lại tất cả các ca kiểm thử là một trong những phương pháp thông thường để thử nghiệm hồi qui, trong đó tất cả các trường hợp kiểm thử trong các bộ thử nghiệm đều phải được chạy lại. Vì vậy, kỹ thuật này là rất tốn kém để thực hiện đầy đủ vì đòi hỏi nhiều thời gian và ngân sách [16].

B. Chọn lựa ca kiểm thử hồi qui (RTS)

RTS là kỹ thuật chọn lựa một tập các ca kiểm thử từ bộ kiểm thử để thực hiện nếu như chi phí của việc chọn lựa ca kiểm thử này ít hơn chi phí kiểm thử mà kỹ thuật RTS cho phép bỏ qua. RTS chia bộ kiểm thử thành: (1) ca kiểm thử có thể dùng lại được (re-usable); (2) ca kiểm thử có thể kiểm thử lại được (retestable); (3) các ca kiểm thử không còn dùng được (obsolete). Ngoài ra, RTS có thể tạo ra các ca kiểm thử mới để kiểm thử cho các vùng mà không được bao phủ bởi các trường hợp thử nghiệm hiện có. Kỹ thuật RTS được sắp xếp thành ba loại [7, 8, 9]:

- Kỹ thuật bao phủ: áp dụng điều kiện bao phủ mã nguồn để thực hiện chọn lựa ca kiểm thử. Tìm kiếm các phần của chương trình có thể bao phủ được điều kiện mà phần chương trình đó đã thay đổi để tìm chọn lựa ca kiểm thử.
- Kỹ thuật tối thiểu hóa: tương tự như kỹ thuật bao phủ, nhưng kỹ thuật này chọn lựa số ca kiểm thử ít nhất.
- Kỹ thuật an toàn: kỹ thuật này không tập trung vào mức độ bao phủ, ngược lại sẽ chọn tất cả các ca kiểm thử mà cho ra kết quả là khác nhau với cùng một chương trình được chỉnh sửa đó và đem so sánh kết quả đó với phiên bản gốc của nó.

Có nhiều kỹ thuật RTS được các nhà nghiên cứu đưa ra như sau:

- Kỹ thuật thay đổi các hàm không phải là cốt lõi [18]: với kỹ thuật này chỉ chọn lựa các ca kiểm thử thực thi các chức năng tại nơi chương trình thay đổi hay bị xóa bỏ đó.
- Kỹ thuật Modification focused Minimization [20]: sử dụng cách tiếp cận của Fischer’s tìm một tập hợp con của bộ kiểm tra mà nó là tối thiểu bao gồm tất cả các chức năng trong chương trình được xác định là thay đổi.
- Kỹ thuật Coverage focused Minimization [20]: sử dụng kỹ thuật giảm bộ kiểm thử của Gupta, Harrold và Soffa để tìm tập con của bộ kiểm thử mà là nhỏ nhất nhưng có thể phủ hết được tất cả các chức năng của chương trình. Phát triển kỹ thuật này, nhiều tác giả đã có các nghiên cứu liên quan như:

- Sử dụng giải thuật Simulating Annealing (SA): Mansour and El-Faikh [6] đã đề xuất công thức tối ưu hóa việc lựa chọn ca kiểm thử hồi qui.
- Reduction Methodology (RED) do Harrold và cộng sự [12] đề xuất phương pháp giảm thiểu số lượng ca kiểm thử được chọn.

- Slicing Algorithms (SLI): do Agrawal và cộng sự [22] đã đề xuất giải thuật chọn lựa ca kiểm thử mà đầu ra của chúng có thể bị ảnh hưởng bởi chương trình đó.

C. Xác lập ưu tiên cho ca kiểm thử (TCP)

Kỹ thuật lựa chọn ca kiểm thử dựa vào thứ tự ưu tiên của nó giúp cho việc phát hiện lỗi tốt hơn và nhanh hơn nhằm tăng độ tin cậy cho chương trình sau khi sửa đổi. Có 2 loại lựa chọn ca kiểm thử theo thứ tự ưu tiên [7,21]: (1) Ưu tiên tổng quát là phương pháp lựa chọn và sắp xếp thứ tự ca kiểm thử có ảnh hưởng mức trung bình đối với các phiên bản phần mềm tiếp theo, (2) Specific Priorization sẽ liên quan đến việc chọn phiên bản cụ thể nào đó của chương trình. Theo Rothermel và cộng sự [21], R. Beena và S. Sarala [7], vấn đề xác định ưu tiên của ca kiểm thử được phát biểu như sau:

Phát biểu vấn đề: Cho T là một bộ kiểm thử, PT là một tập hợp các hoán vị của T, f là một hàm ánh xạ từ PT đến tập số thực. Tìm $T' \in PT$ với $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$.

Trong đó, PT đại diện cho tập tất cả các thứ tự ưu tiên có thể của T và f là một hàm của T được áp dụng cho bất kỳ thứ tự nào. Có 18 kỹ thuật ưu tiên hóa các ca kiểm thử được xác định ở [23], như các kỹ thuật so sánh, các kỹ thuật mức dòng lệnh và các kỹ thuật mức chức năng. Hiện tại, có nhiều thuật toán tìm độ ưu tiên ca kiểm thử được phát triển bởi nhiều nhà nghiên cứu trong lĩnh vực này. Cụ thể, có thể liệt kê các thuật toán điển hình:

- Thuật toán tham lam (Greedy): đây là thuật toán tìm với chi phí thấp, độ phức tạp O(mn) với m dòng lệnh và n ca kiểm thử.
- Thuật toán Greedy bổ sung [15]: thuật toán này sử dụng phản hồi từ sự chọn lựa trước đó. Nó chọn các thành phần có trọng số lớn nhất trong phần mà chưa được chọn trước đó. Chi phí cho thuật toán này là $O(mn^2)$.
- Thuật toán 2-Optimal [24]: áp dụng bài toán người du lịch. Chi phí cho việc tìm ca kiểm thử ưu tiên là $O(mn^3)$.
- Thuật toán leo đồi (Hill Climbing) [24]: Chi phí cho thực hiện thuật toán này không cao, dễ sử dụng nhưng có hạn chế là thực hiện việc chia O(n2).
- Thuật toán di truyền (GA): đây là thuật toán phổ biến được các nhà nghiên cứu vận dụng với các cải biên khác nhau.

D. Cách tiếp cận kết hợp

Kỹ thuật kết hợp cả RTS và TCP, có nhiều nghiên cứu thực hiện và đưa ra một số đề xuất kết hợp cả RTS và TCP cùng với việc xây dựng các thuật toán tương ứng như: (1) giải thuật lựa chọn ca kiểm thử được đề xuất bởi Aggarwal và cộng sự [26] thực hiện việc lựa chọn và giảm thiểu số ca kiểm thử; (2) kỹ thuật kết hợp được đề xuất bởi Wong và các cộng sự [27] là tìm kết hợp tối thiểu nhất, sửa đổi và lựa chọn ưu tiên dựa trên kết quả kiểm thử lịch sử; (3) Yogesh Singh và cộng sự đề xuất một sự kết hợp giữa RTS và TCP được nghiên cứu chi tiết trong [28].

E. Kiểm thử hồi qui đối với ứng dụng di động trong môi trường phát triển linh hoạt

Khi nhu cầu người dùng chuyển từ việc sử dụng PCs sang thiết bị di động thông minh thì xu hướng phát triển ứng dụng của các công ty phần mềm cũng thay đổi và đầu tư sang lĩnh vực ứng dụng di động. Sự thay đổi này sẽ mang đến những thách thức cho hoạt động kiểm thử bởi sự đa dạng về thiết bị, chu kỳ phát triển của phần cứng và hệ điều hành ngắn, các ứng dụng triệu gọi các dịch vụ back-end trên máy chủ thường xuyên hơn [3, 4, 13]. Vì thế, kiểm thử cho ứng dụng di động trong môi trường phát triển linh hoạt cũng phải điều chỉnh cho phù hợp. Các nghiên cứu của Hagai Cibulski và Amiram Yehudai [32] đã chỉ ra tầm quan trọng và thách thức của kiểm thử hồi qui theo phương pháp phát triển hướng kiểm thử (TDD) đó là việc chọn lựa tập con ca kiểm thử để thực hiện hồi qui mức đơn vị, sự thay đổi mã lệnh trong các hàm, các mô đun khi phát triển. Kỹ thuật được áp dụng là phân tích động và phân tích tĩnh dựa trên mã nguồn và bộ kiểm thử cho mã nguồn đó. Meenakshi và Mr. Rajvir Singh [13] cũng đưa ra một số kỹ thuật RTS và phương pháp thực hiện hồi qui nhằm nâng cao chất lượng của phần mềm trong môi trường phát triển linh hoạt. Nghiên cứu này [13] tập trung đưa ra cách tiếp cận, ứng dụng các kỹ thuật RTS đã được nghiên cứu để vận dụng vào môi trường phát triển linh hoạt một cách hiệu quả và tập trung vào các nghiên cứu các kỹ thuật lựa chọn ca kiểm thử dựa trên phân cụm, dựa trên các kỳ vọng, dựa trên ca sử dụng và các kỹ thuật tính toán thông minh. Abu Wahid Md,. Masud Parvez [4] và Anita, Dr. Naresh Chauhan [14] đã đề xuất một mô hình hiệu quả cho kiểm thử hồi qui ứng dụng di động trong môi trường phát triển ứng dụng linh hoạt. Nghiên cứu này đề xuất cách tiếp cận thực hiện hồi qui cho 6 giai đoạn triển khai kiểm thử hồi qui của một Sprint trong quy trình Scrum.

III. ĐỀ XUẤT PHƯƠNG PHÁP VÀ THUẬT TOÁN RTS

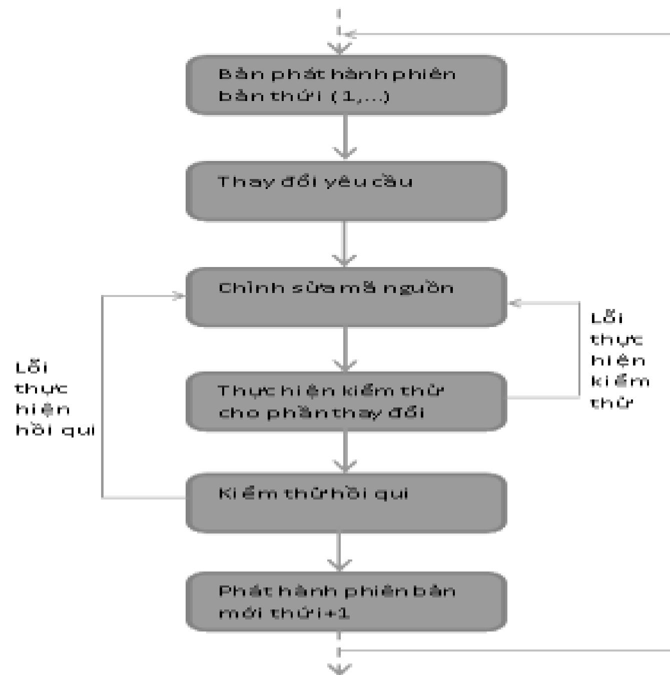
Dựa trên các nghiên cứu đã trình bày ở phần II và đặc biệt là II.E, chúng tôi nghiên cứu và đề xuất quy trình áp dụng kiểm thử hồi qui cho ứng dụng di động và kỹ thuật lựa chọn ca kiểm thử hồi qui RTS dựa trên thay đổi mã lệnh áp dụng ở mức kiểm thử đơn vị và áp dụng thử nghiệm cho phát triển ứng dụng di động mobile Android.

A. Quy trình và phương pháp ứng dụng

Trong thực tế sẽ có một số biến thể về các quy trình phát triển và quy trình kiểm thử (bao gồm kiểm thử hồi qui) tùy thuộc vào từng tổ chức phần mềm. Tuy nhiên, về cơ bản, một qui trình kiểm thử hồi qui được thực hiện qua các

giai đoạn trong Hình 1 được áp dụng cho phát triển ứng dụng desktop-PC, web và ứng dụng di động (mobile application). Quy trình kiểm thử thông thường được cấu trúc cho các mức đơn vị, tích hợp mức đơn vị, kiểm thử mức hệ thống và tích hợp mức hệ thống. Cấu trúc này được áp dụng cho mô hình phát triển V-model bởi việc phát triển được thực hiện tuần tự tuyến tính [3]. Đối với các dự án phát triển theo phương pháp linh hoạt, các mức kiểm thử này được lặp lại và chồng lắp nhiều lần. Trong các phương pháp phát triển linh hoạt hiện nay bao gồm XP (eXtreme programming), Scrum, DSDM (Dynamic Systems Development Method), Lean và FDD (Feature Driven Development) thì quy trình Scrum được ưu tiên lựa chọn như là một phương pháp, một khung phát triển dự án theo phương pháp linh hoạt (agile) phổ biến. Đặc biệt, quy trình này phù hợp với các dự án phát triển ứng dụng di động nói riêng và những ứng dụng có tính thay đổi yêu cầu cao, tính phức tạp trong kiểm thử [5], đòi hỏi sản phẩm xuất xưởng sớm nhưng yêu cầu về chất lượng không hề giảm mà ngày càng cao hơn.

Kiểm thử hồi quy là một phần quan trọng của quá trình kiểm soát chất lượng phần mềm. Trên thực tế, đối với quy trình phát triển ứng dụng truyền thống, quy trình kiểm thử hồi quy được nghiên cứu và đề xuất tương ứng rõ ràng. Tuy nhiên, đối với phương pháp phát triển linh hoạt Scrum thì chưa có nhiều nghiên cứu cho việc vận dụng quy trình kiểm thử hồi quy. Bởi vì đối với quy trình Scrum, phương pháp tiếp cận phát triển và kiểm thử đã thay đổi. Vì vậy, việc nghiên cứu và đề xuất quy trình kiểm thử hồi qui trong quy trình Scrum là cần thiết.



Hình 1. Quy trình tổng quát cho kiểm thử hồi qui

Hình 2 dưới đây là đề xuất quy trình thực hiện kiểm thử và kiểm thử hồi qui trong từng giai đoạn nước rút (sprint) của dự án/sản phẩm; ứng dụng phương pháp phát triển TDD (Test Driven Development), hoạt động kiểm thử ngay từ giai đoạn đầu của mỗi giai đoạn nước rút, việc lặp lại sau khi tái cấu trúc (refactoring) cũng được xem là hoạt động hồi qui và tích hợp liên tục (CI – continuos integration). Kết thúc mỗi một giai đoạn nước rút sẽ thực hiện hồi qui toàn bộ phiên bản thứ i, tiếp tục thay đổi, bổ sung tính năng, yêu cầu và mở ra sprint tiếp theo. Các yêu cầu (stories/backlog) sẽ không được thay đổi mỗi khi Sprint đó đã được thực thi. Tương ứng với từng công đoạn thể hiện ở Hình 2, với mỗi yêu cầu (story) được phát triển sẽ thực hiện hồi qui khi có sự thay đổi tương ứng với từng mức kiểm thử. Nghĩa là từng công đoạn có thể có thực hiện hồi qui ngay ở công đoạn đó và thực hiện hồi qui cho tất cả các công đoạn. Các bước con khi thực hiện kiểm thử hồi qui, cụ thể:

- Bước 1: Kiểm tra lại tính hợp lệ/lựa chọn/tối ưu hóa/ ưu tiên hóa ca kiểm thử. Ở bước này, không phải tất cả các ca kiểm thử cho P đều thực thi được trên P’(P là phiên bản ban đầu của chương trình/mô đun, P’ và phiên bản chỉnh sửa, thay đổi mã lệnh của P), một số ca kiểm thử không hoạt động được, một số dư thừa và cần thiết phải sắp xếp lại theo thứ tự ưu tiên.

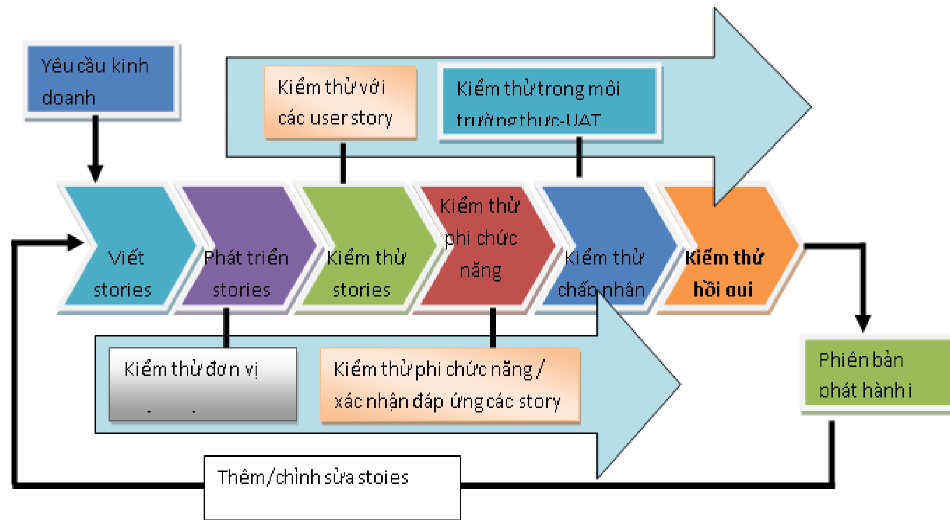
- Bước 2: Thiết lập kiểm thử. P’ được thiết lập để thực hiện kiểm thử ở môi trường thử nghiệm hoặc mô phỏng hoặc giả lập.

- Bước 3: Trình tự kiểm thử. Các ca kiểm thử được thực hiện theo một trình tự xác định hoặc bất kỳ của các dữ liệu điều kiện đầu vào.

- Bước 4: Thực thi kiểm thử

- **Bước 5:** So sánh kết quả đầu ra, mỗi kết quả cần được xác thực.

- **Bước 6:** Giảm thiểu lỗi, bước xử lý, làm gì khi lỗi xảy ra?.



Hình 2. Quy trình kiểm thử hồi qui trong phương pháp phát triển Agile Scrum

Phát triển ứng dụng di động đang đòi hỏi một tốc độ phát triển rất nhanh, chu kỳ phát hành đã được rút ngắn và nếu ứng dụng không thường xuyên cải tiến, người dùng sẽ bắt đầu tìm kiếm giải pháp thay thế. Một điểm yếu trong phát triển và kiểm thử phần mềm là kiểm thử hồi qui. Để thiết kế, phát triển và triển khai các tính năng mới, nhưng điều quan trọng là phải đảm bảo rằng những thay đổi mới cho sản phẩm không phá vỡ chức năng hiện có, không xuất hiện các lỗi mới, hoặc xuất hiện lại các lỗi được giải quyết trước đó. Sau khi thiết lập một chiến lược kiểm thử, kiểm thử hồi qui có thể được xem là hoạt động tối ưu hóa để có được giá trị tốt nhất về công sức, nỗ lực thực hiện kiểm thử. Cụ thể, rất hiếm khi có một yêu cầu để kiểm tra mọi sự kết hợp có thể có của các nền tảng, hệ điều hành và các thiết bị. Vì vậy, chiến lược là để xác định các khu vực có nguy cơ cao và tập trung vào những điểm này. Những khu vực này sẽ phát hiện ra 90% các khiếm khuyết của ứng dụng. Nhiều công cụ phát triển và kiểm thử tự động cho ứng dụng di động đã phát hành và đang phát triển để hỗ trợ cho việc kiểm thử chức năng, kiểm thử giao diện, kiểm thử hiệu suất và kiểm thử hồi qui. Tuy nhiên, mức độ hỗ trợ kiểm thử hồi qui của mỗi công cụ là khác nhau, chủ yếu là thực hiện theo chế độ record-playback, các công cụ được giới thiệu ở Bảng 1. Kiểm thử hồi qui mức đơn vị chưa được thực hiện bởi sự tốn kém về thời gian và chi phí thực hiện.

Bảng 1. Một số công cụ kiểm thử tự động hỗ trợ kiểm thử hồi qui phổ biến [34]

TT	Công cụ	Năm phát hành
1	AgileLoad	2006
2	Android GUITAR	2011
3	Android SDK	2009
4	Appium	2013
5	Cucumber	2008
6	Googletest	2008
7	Load Testing	2003
8	MobileCloud	2006
9	MonkeyTalk	2008
10	Robolectric	2010
11	Robotium	2010
12	Selenium WebDriver	2012

B. Chiến lược thực hiện

Việc vận dụng quy trình, triển khai phương pháp kiểm thử hồi qui được thực hiện theo chiến lược sau:

a) Xác định chắc chắn những gì cần thực hiện và nên thực hiện như thế nào. Thực hiện hồi qui mỗi khi tính năng mới được thêm vào hay thay đổi. Mỗi tính năng mới thêm vào hay thay đổi đã được thực hiện kiểm thử để đảm bảo nó hoạt động một cách đúng đắn trước khi thực hiện hồi qui.

b) Thiết lập tất cả các yêu cầu và đảm bảo chắc chắn rằng các yêu cầu đã được quyết định với sự tham gia từ phía khách hàng cũng như các bên liên quan. Sự hợp tác chặc chẽ giữa các bên liên quan sẽ giúp cho hoạt động kiểm thử mang lại hiệu quả cao nhất.

c) Xác định điều kiện đầu vào của kiểm thử hồi qui bao gồm việc thiết lập điều kiện tối thiểu cần thiết để bắt đầu thử nghiệm hồi qui như: xác nhận các khiếm khuyết là lặp lại, các khiếm khuyết có đủ tài liệu kèm theo, có ghi nhận các nguồn lực cho kiểm thử hồi qui và xác định mối liên quan của các khiếm khuyết.

d) Kiểm tra các điều kiện, tiêu chuẩn đầu vào – đầu ra tương ứng: tất cả các kiểm thử cần thiết được thông qua với việc đảm bảo cùng điều kiện bao phủ mã lệnh, không có lỗi nghiêm trọng, các rủi ro mức cao phải được bao phủ.

e) Thực hiện theo lịch trình và vận dụng theo phương pháp của Srinivasan Desikan [17] và [33].

Thực hiện khởi tạo kiểm thử Smoke hoặc Sanity:

Một tập hợp các ca kiểm thử hồi qui có thể được thực hiện kiểm thử smoke. Kiểm thử smoke (smoke testing) là một nhóm các trường hợp thử nghiệm chứng minh rằng hệ thống ổn định và tất cả các chức năng chính đang hoạt động dưới điều kiện bình thường. Smoke testing có thể được thực thi trước khi quyết định tiến hành thêm các thử nghiệm khác nhằm mục đích xác định rõ tại sao phải dành nguồn lực để kiểm tra nếu hệ thống là không ổn định. Mục đích của việc kiểm thử smoke là để chứng minh sự ổn định của sản phẩm hay hệ thống, không phải để tìm lỗi của hệ thống. Kiểm thử Sanity (Sanity testing) được thực hiện để kiểm tra các chức năng chính của sản phẩm hay hệ thống có hoạt động hay không. Nếu Sanity testing không thành công thì kiểm thử hồi qui sẽ dừng lại để tiết kiệm thời gian và chi phí liên quan trong một thử nghiệm nghiêm ngặt hơn.

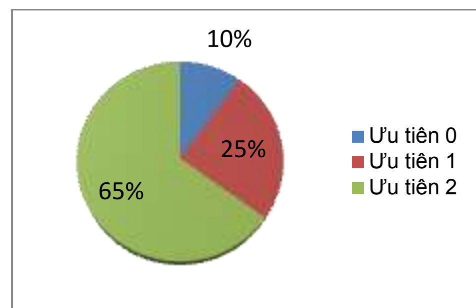
Tiêu chuẩn chọn ca kiểm thử hồi qui:

Tiêu chuẩn để lựa chọn ca kiểm thử hồi qui đã được đúc kết từ dữ liệu công nghiệp dựa trên số lượng các báo cáo lỗi quan trọng. Việc lựa chọn các ca kiểm thử để thực thi hồi qui là một nghệ thuật và không phải một công việc dễ dàng. Việc lựa chọn cần yêu cầu về kiến thức của các bản vá lỗi và sự ảnh hưởng của nó đến hệ thống, các vùng lỗi thường xuyên, vùng thay đổi mã lệnh mới nhất, cùng dễ được phát hiện bởi người dùng, các tính năng chính của phần mềm và các yêu cầu bắt buộc từ khách hàng. Lựa chọn các ca kiểm thử để thực hiện hồi qui phụ thuộc nhiều vào tầm quan trọng của các bản vá lỗi hơn là tầm quan trọng của các khiếm khuyết. Một khiếm khuyết nhỏ có thể dẫn đến tác dụng phụ lớn và một bản vá lỗi cho một khiếm khuyết cực đoan có thể không có hoặc chỉ là một ảnh hưởng nhỏ. Vì vậy, các kỹ sư kiểm thử cần xem xét nhiều góc độ để lựa chọn các ca kiểm thử cho thực hiện hồi qui.

Xác định mức ưu tiên cho các ca kiểm thử:

Xác định thứ tự ưu tiên các ca kiểm thử phụ thuộc vào các tác động nghiệp vụ, chức năng quan trọng và thường xuyên được sử dụng. Lựa chọn các ca kiểm thử dựa trên các mức ưu tiên sẽ giảm được số bộ kiểm thử. Các ca kiểm thử có thể được phân thành ba mức [17]:

- **Ưu tiên 0:** Các test cases có thể được gọi là Sanity test case khi kiểm tra các chức năng cơ bản và thực thi để chấp nhận phiên bản sản phẩm cho kiểm thử sau này. Điều này cũng được thực thi khi dự án có những thay đổi lớn. Những trường hợp thử nghiệm này sẽ cung cấp một giá trị lớn cho dự án đối với cả hai phía: nhóm phát triển và khách hàng.
- **Ưu tiên 1:** Sử dụng các thiết lập cơ bản và bình thường, các trường hợp thử nghiệm cung cấp giá trị cao cho dự án đối với cả hai phía: nhóm phát triển và khách hàng.
- **Ưu tiên 2:** Những trường hợp thử nghiệm cung cấp giá trị dự án vừa phải và được thực hiện như một phần của chu trình thử nghiệm sản phẩm và chọn lựa cho hồi qui trên cơ sở nhu cầu.



Hình 3. Tỷ lệ các mức ưu tiên của Test cases

Phương pháp lựa chọn các ca kiểm thử

Một khi các trường hợp thử nghiệm được ưu tiên có thể được lựa chọn để thực thi. Có thể có nhiều cách tiếp cận để thực hiện kiểm thử hồi qui mà cần phải được quyết định trên cơ sở từng trường hợp [17]. Ví dụ:

- *Trường hợp 1:* Nếu mức độ quan trọng và mức độ ảnh hưởng của việc sửa lỗi là thấp, khi đó chỉ cần chọn một một vài trường hợp thử nghiệm từ Test Case DataBase và thực hiện chúng là đủ. Các ca kiểm thử này có thể thuộc bất kỳ ưu tiên 0, 1, hoặc 2.
- *Trường hợp 2:* Nếu mức độ quan trọng và mức độ tác động của việc sửa lỗi là trung bình, khi đó chúng ta cần phải thực hiện tất cả các trường hợp ưu tiên 0 và ưu tiên 1. Nếu sửa lỗi cần trường hợp thử nghiệm bổ sung từ

ưu tiên 2 cũng có thể lựa chọn và thực hiện hồi qui. Lựa chọn ưu tiên 2 trong trường hợp này là cần nhưng không bắt buộc.

- *Trường hợp 3:* Nếu mức độ quan trọng và tác động của việc sửa lỗi là cao, khi đó chúng ta cần phải thực hiện tất cả các ưu tiên 0, ưu tiên 1, ưu tiên 2 cần cẩn trọng khi chọn lựa.

High			P0: All P1: All P2: Subset
Med		P0: All P1: All P2: few	
Low	P0: few P1: few P2: few		
Criticality / Impact	Low	Med	High

P0: Ưu tiên mức 0, mức cao nhất — Few : lựa chọn một số TC
P1: Ưu tiên mức 1 — All : Tất cả test case
P2: Ưu tiên mức 2 — Subset: chọn một phần

Hình 4. Mức độ quan trọng và mức độ ưu tiên ca kiểm thử

Các phương pháp trên đòi hỏi phải phân tích tác động của các bản vá lỗi cho tất cả các khiếm khuyết của sản phẩm. Điều này sẽ tốn khá nhiều thời gian. Nếu không có đủ thời gian và rủi ro của việc không làm phân tích tác động là thấp thì các phương pháp thay thế sau đây có thể được thực hiện:

- Thực thi lại toàn bộ các ca kiểm thử: tất cả các ca kiểm thử ưu tiên 0, 1, và 2 đều được thực thi lại.
- Kiểm thử hồi qui dựa trên mức độ ưu tiên: dựa trên các ưu tiên, tất cả các ca kiểm thử ưu tiên 0, 1, và 2 được thực hiện theo thứ tự, dựa vào thời gian cho phép.
- Phương pháp chọn ngẫu nhiên: trường hợp thử nghiệm ngẫu nhiên được lựa chọn và thực hiện.
- Thay đổi hồi qui: Sự thay đổi mã lệnh được so sánh với chu kỳ cuối cùng của việc kiểm thử và các trường hợp được lựa chọn dựa trên tác động của chúng trên các mã lệnh đó.

Một chiến lược hồi qui hiệu quả thường là sự kết hợp của tất cả các phương pháp trên.

D. Xây dựng thuật toán RTS lựa chọn, giảm thiểu và ưu tiên hóa ca kiểm thử hồi qui

Thuật toán RTS (Regression Test Selection) dựa trên mức độ bao phủ mã lệnh [7, 8, 11, 26, 29, 30] bằng việc xem xét thực thi các ca kiểm thử bao phủ các dòng lệnh được sửa đổi trong kiểm thử hộp trắng mức đơn vị. Gọi P là mã lệnh trước khi sửa đổi của một hàm hoặc phương thức, P' là phiên bản chỉnh sửa của P, T là tập ca kiểm thử của P, TP là một tập bao phủ mã lệnh dựa trên các ca kiểm thử để kiểm thử cho P. Khi P được chỉnh sửa thành P' thì mục tiêu là đi tìm T'; tập con nào của TP mà bao phủ tất cả dòng lệnh được thay đổi trước tiên nhất. Nếu có nhiều hơn 2 các ca kiểm thử có cùng số dòng lệnh sửa đổi và các giá trị của chúng cũng tương thích khi xem xét các trường hợp thử mà có ít số dòng lệnh hơn so với bản sửa đổi. Thuật toán này được áp dụng trong quy trình và các bước thực hiện hồi qui được nêu ở mục III.A.

RTS_based_modified_code_coverage function
Input
 P là mã nguồn trước khi sửa đổi
 P' là mã nguồn được sửa đổi
 T tập các ca kiểm thử (TC- test case) đã sử dụng để kiểm thử P
Output
 T' tập các ca kiểm thử được chọn để thực thi hồi qui.
Begin
Sinh đồ thị CFG cho P
Xác định tập đường đi tối thiểu phủ được đồ thị CFG, mỗi đường đi Ti là một ca kiểm thử của P
Gán tập Ti vào T
T '= ϕ
For each TC t,
 So sánh các dòng lệnh được sửa đổi với các dòng lệnh được bao phủ bởi các ca kiểm thử;
 Tính toán số lượng các ca kiểm thử tương xứng nhau (match) được tìm thấy và;
 Lưu các giá trị của các cặp tương xứng này.
End for
For each TC t,
 Kiểm tra số lượng các cặp tương xứng của t với bất kỳ TC nào khác
 If số lượng các cặp tương xứng của t bằng với bất kỳ TC nào khác **Then**

So sánh từng giá trị của các cặp tương xứng (matched values)
If các giá trị này là bằng tương ứng, **Then**
Kiểm tra TC nào có số dòng lệnh phủ nhỏ nhất.
Lưu lại TC đó và loại bỏ TC kia
If số dòng lệnh bao phủ là như nhau **Then**
Lưu bất kỳ TC nào trong cặp tương xứng này.
End if
End if
End if
If nếu số lượng các cặp tương xứng của t bằng 0 **Then Loại bỏ** TC đó **End if**
End for
Sắp xếp tập TC đã được giảm của T theo thứ tự giảm dần và đánh số 1, 2, 3 … cho đến kích thước của danh sách.
Gán giá trị của TC đầu tiên vào mảng A và T’ = {1^{st} TC}
For each TC t,
So sánh TC t với mảng A
If t có một vài giá trị khác **Then**
Đẩy vào mảng A giá trị khác đó của t.
Ghi lại các giá trị tương xứng mà t có với các phần tử của mảng T’ một cách độc lập
Gán t vào mảng T'
Else
Loại bỏ t.
T'=T' - t
End if
End for
For each TC t in T'
So sánh t các ca kiểm thử còn lại
If tất cả các phần tử của t được tìm thấy trong ca kiểm thử khác **Then**
Loại bỏ t.
T'=T' - t
End if
End for
Output: T'.// *Số lượng tối thiểu bộ test case để thực hiện kiểm thử hồi qui cho mã nguồn đã được chỉnh sửa.*
End.

IV. KẾT QUẢ THỬ NGHIỆM VÀ THẢO LUẬN

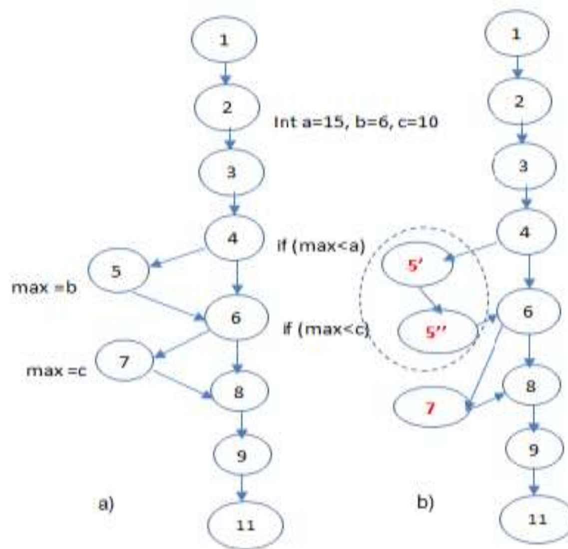
Thuật toán và các bước tiến hành kiểm thử hồi qui ở mục III được triển khai thực nghiệm cho phát triển các ứng dụng di động Android, môi trường phát triển Eclipse Android Development Tool với plug-in “Source Code Visualizer” do Dr. Garbage phát triển [35] để hỗ trợ bước sinh biểu đồ cấu trúc điều khiển CFG cho chương trình hoặc môđun P, ngôn ngữ phát triển và thử nghiệm là Java. Một plugin được phát triển để phục vụ cho việc sinh CFG, tập ca kiểm thử T, và tìm T’ như trình bày trong thuật toán RTS trình bày trong mục III.D. Kết quả triển khai được minh họa bởi chương trình tìm max của 3 số nguyên cho trước ở Hình 5 và đồ thị CFG của mã lệnh thể hiện ở Hình 6a), nốt dòng lệnh được chỉnh sửa ở Hình 6b).

```
public class ControlFlow {
1) public static void main(String[] args) {
// tìm max của 3 số a,b,c
2)     int a=15,b=6,c=10;
3)     int max= a;
4)     if(max<b  )
5)          max=b;
6)     if(max<c)
7)          max=c;
8)          System.out.println(a+"\t"+b+"\t"+c +"\t"+ max );
9)     }
   }
```

Hình 5. Mã lệnh của chương trình tìm max của ba số nguyên

Tập các ca kiểm thử cơ bản phủ được mã lệnh của chương trình T, gồm: {T1 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 11; T2= 1, 2, 3, 4, 6, 7, 8, 9, 11; T3= 1, 2, 3, 4, 6, 8, 9, 11}, trong đó nốt 1 là START và 11 là EXIT. Các nốt dòng lệnh được sửa đổi M = {4,6} (nốt 4 và 6 trong Hình 6a). Kết quả thực thi của thuật toán trên, tìm được T’ = {}, điều này có nghĩa T = T’, thực hiện kiểm thử hồi qui cho toàn bộ các ca kiểm thử có trong T. Dễ thấy rằng, nốt 4 và 6 chứa dòng lệnh điều kiện if, do đó nếu thay đổi mã ở đây sẽ ảnh hưởng đến các khối lệnh của nó. Nếu nốt có dòng lệnh được sửa đổi M = {5,7} (nốt 5 và 7 trong Hình 6b). Kết quả thực thi thuật toán trên sẽ cho ra T’ = T1. Điều này có nghĩa là chỉ cần dùng T1 để thực hiện kiểm thử hồi qui và có thể kết luận chương trình đảm bảo được tính đúng đắn như khi thực hiện

với bộ T cho P'. Các nốt này không đồng nhất với thứ tự dòng lệnh trong chương trình; việc thay đổi mã lệnh không làm thay đổi CFG của chương trình (đảm bảo tập T ban đầu không thay đổi).



Hình 6. CFG của chương trình a) trước khi thay đổi mã nguồn, b) sau khi thay đổi mã nguồn

Qua thực nghiệm với 8 chương trình (môđun) khác nhau thu được kết quả như Bảng 2 dưới đây.

Bảng 2. Kết quả thử nghiệm thuật toán RTS với các bài toán khác nhau

Chương trình P	Số dòng lệnh	Tập T	Tập M	Tập T'	Tỷ lệ tối ưu	Ghi chú
MaxOf2Integer	10	T={T1,T2,T3}	{5,7} {4,6}	T'={T1} T'={}	66.7% 0%	Dùng If đơn
LoopWhile	11	T={T1,T2,T3,T4 }	{2,5,8} {2,6,7,8}	T'={T3,T4} T'={T2,T3,T4 }	50% 25%	While If else, break, continue
Triangle	30	T={T1,T2,T3,T4,T5}	{5,8,14,16 ,20} {5,8,9,10, 22}	T'={T3} T'={T4}	80%	If-else
SquareEquation	38	T ={T1 – T10}	{6,10,15,1 6,21,22,29 ,30}	T'={T2,T5,T7, T8}	60%	If-else
Gameloop.run()	28	T = {T1- T41}	M bất kỳ	T=''	0%	Unlimited Loop, Try-Catch, jump
LoopFor	25	T={T1,T2,T3,T4,T5,T6}	{7,9,12,15 }	T'={T2}	83.3%	If-Else, For
Calculator	40	T={T1,T2,T3,T4,T5,T6,T 7,T8,T9}	{5,8,9,12, 20,21,22}	T'={T2,T6,T7 }	66.7%	Switch-Case, If-Else
ExRegression	100	T={T1,T2,T3,T4,T5,T6 ,T7,T8,T9}	{2, 6, 10, 15, 20, 21, 22, 29, 31}	T' = {T3, T4, T6, T7}	55,56%.	If, for,

Với số liệu thử nghiệm và kết quả thể hiện ở Bảng 2, chúng tôi nhận thấy rằng việc sửa đổi các mã lệnh (M) tại các dòng điều kiện, rẽ nhánh thì tỷ lệ tối ưu thấp hoặc T' = T. Hay nói cách khác, tỷ lệ tối ưu phụ thuộc vào tập các dòng lệnh được thay đổi M. Nếu việc thay đổi hay loại bỏ các mã lệnh tầm thường, ít tác động đến cấu trúc CFG hay không làm thay đổi đồ thị CFG thì mức độ tối ưu lựa chọn các ca kiểm thử để thực hiện hồi qui là khá hiệu quả. Đối với mã lệnh chứa vòng lặp vô hạn luôn đúng (while (true){..}), đường đi ngắn (đồ thị không sâu nhưng rộng), số trường hợp phủ các giá trị điều kiện đầu vào của bài toán lớn thì hiệu quả của tối ưu ca kiểm thử không cao và có thể là 0% như ở thử nghiệm GamLoop.run(). Kỹ thuật RTS trong nghiên cứu này sẽ giúp bổ sung cho các kỹ thuật RTS, TCP [7,8,30] đã được nghiên cứu. Không có một giải pháp kỹ thuật tốt cho tất cả các trường hợp hay mọi điều kiện mà phụ

thuộc rất nhiều vào từng yêu cầu điều kiện, ràng buộc cụ thể. Giải pháp tốt là sự kết hợp hiệu quả của nhiều phương pháp và kỹ thuật để mang lại lợi ích, hiệu quả cao nhất cho kiểm thử hồi qui.

V. KẾT LUẬN

Kiểm thử hồi qui là hoạt động kiểm thử rất quan trọng trong bất kỳ dự án phát triển phần mềm nào. Kiểm thử hồi qui là hoạt động tốn kém về mặt thời gian và chi phí cho dự án nhưng không thể không thực hiện. Do sự quan trọng và chi phí của hoạt động kiểm thử hồi qui mà cho đến nay có rất nhiều nghiên cứu liên quan nhằm giúp cho các dự án phần mềm, đặc biệt là các dự án phát triển ứng dụng di động theo phương pháp linh hoạt tiết kiệm được chi phí và thời gian, mang lại hiệu quả cho dự án. Việc kết hợp phương pháp lựa chọn ca kiểm thử với phương pháp giảm thiểu hóa số lượng ca kiểm thử và thực hiện ưu tiên hóa ca kiểm thử sẽ mang lại hiệu quả cao cho kiểm thử hồi qui. Đồng thời với đó là chiến lược thực hiện, quy trình thực hiện kiểm thử để nâng cao hiệu quả cho kiểm thử hồi qui của dự án phần mềm ứng dụng di động trong môi trường phát triển linh hoạt.

Thuật toán được đề xuất sẽ cải tiến việc tối ưu lựa chọn ca kiểm thử (tối thiểu hóa) và ưu tiên chọn ca kiểm thử cho kiểm thử hồi qui của phiên bản sửa đổi mã nguồn một chương trình mà cho phép thực thi với ít ca kiểm thử hơn và phát hiện lỗi sớm hơn ngay ở giai đoạn lập trình (mức đơn vị, theo phương pháp TDD). Việc phát hiện lỗi sớm hơn trong thử nghiệm hồi qui sẽ cung cấp thông tin phản hồi sớm về hệ thống được kiểm thử và do đó hoạt động gỡ lỗi có thể bắt đầu sớm hơn, tiết kiệm chi phí nhiều hơn.

Hạn chế của nghiên cứu này là chưa thực nghiệm được một cách đầy đủ quy trình và các bước giải pháp đề xuất để đánh giá hiệu quả của phương pháp. Bên cạnh đó, việc thực nghiệm và đánh giá thuật toán trên các hàm, thủ tục, môđun đơn lẻ. Định hướng phát triển trong thời gian tới là xây dựng thành các plugin cho các IDE phát triển ứng dụng di động hoặc công cụ độc lập để hỗ trợ kỹ sư phần mềm, nhóm dự án Scrum thực hiện có hiệu quả hoạt động kiểm thử hồi qui cho phát triển các dự án phần mềm ứng dụng di động ở mức đơn vi (unit) và tích hợp (CI). Ngoài ra, phương pháp và thuật toán này cũng có thể mở rộng và điều chỉnh để áp dụng cho tất cả các loại ứng dụng PC, hay ứng dụng web và đồng thời kết hợp triển khai các kỹ thuật RTS, TCP của các nghiên cứu [7,8,30] để có được công cụ hỗ trợ đa dạng, mang tính tổng quát và bao phủ rộng nhằm mang lại hiệu quả và nâng cao chất lượng cho phần mềm.

TÀI LIỆU THAM KHẢO

[1] Jiantao Pan, Software Testing, Carnegie Mellon University, 1999, https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

[2] Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development – Writing Test Cases", Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, pp. 425-434, 2007.

[3] Jerry Gao et al, "Mobile Application Testing: a Tutorial", Computer, Vol.47, No. 2, pp.46-55, Feb.2014, doi:10.1109/MC.2013.445.

[4] Parvez, A. W. M. M., "An Efficient Model for Mobile Application Regression Test for Agile Scrum Software Development", Advances in Computer Science and its Applications, *2*(2), 339-344, 2012.

[5] Dehlinger, Josh, and Jeremy Dixon, "Mobile application software engineering: Challenges and research directions", Workshop on Mobile Software Engineering, Vol. 2, 2011.

[6] Mansour, Nashat, and Khalid El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing", Journal of Software Maintenance 11.1, pp.19-34, 1999.

[7] Duggal, G., & Suri, B., " Understanding regression testing techniques", In Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology, 2008.

[8] Beena, R., and S. Sarala, "Code coverage based test case selection and prioritization", arXiv preprint arXiv:1312.2083 ,2013

[9] Yogita Garg, Arun P. Agrawal, "Implementation of Test Case Prioritization Technique Using Static Path", International Journal of Advanced Research in Computer Science and Software Engineering, ISSN: 2277 128X, Volume 3, Issue 8, August 2013, Available online at: www.ijarcsse.com

[10] Spillner, Andreas, Tilo Linz, and Hans Schaefer, Software testing foundations: a study guide for the certified tester exam, Rocky Nook, Inc., 2014.

[11] Nguyễn Đức Mận, Huỳnh Quyết Thắng, Trần Xuân Hoàng, "Một số kỹ thuật áp dụng trong mô hình kiểm thử mã nguồn cho các phương thức của lớp trong Java", Kỷ yếu Hội thảo quốc gia lần thứ XVII: Một số vấn đề chọn lọc của Công nghệ thông tin và truyền thông-Đắk Lắk,30-31/10/2014, trang 167-174, ISBN 978-604-67-0426-3,2014.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of the test suite", ACM Transaction on Software Engineering and Methodology, Volume 2, Issue 3, pp. 270-285, July 1993.

[13] Meenakshi and Rajvir Singh, "Regression Testcase Selection in Agile Environment", International Journal for Scientific Research & Development, ISSN (online): 2321-0613, Vol. 3, Issue 05, pp. 570-576, 2015.

[14] Arora, A., & Chauhan, N., "A regression test selection technique by optimizing user stories in an agile environment", Proceedings of Advance Computing Conference (IACC), 2014 IEEE International, pp. 1454-1458, 2014.

[15] Sebastian Elbaum, Praveen Kallakuri, Alexey G. Malishevsky, Gregg Rothermel, Satya Kanduri, "Understanding the Effects of Changes on the Cost-Effectiveness of Regression Testing Techniques", Journal of Software Testing, Verification, and Reliability, 13(2) pp. 65-83, June 2003.

[16] H. Leung and L. White, "Insights into regression testing", Proceedings of the Conference on Software Maintenance, ICSM '89, pp. 60-69, Oct. 1989.

[17] Srinivasan Desikan, Software Testing: Principles and Practice, Pearson Education India, 2007.

[18] Y. Chen, D. Rosenblum, and K. Vo. TestTube, "A system for selective regression testing", Proceedings of the 16th International Conference on Software Engineering, pp. 211-222, May 1994.

[19] Swarnendu Biswas and Rajib Mall, at el, "Regression Test Selection Techniques: A Survey", Informatica 35(3) pp. 289–321, 2011.

[20] R. Gupta, M. J. Harrold and M. Soffa, "An approach to regression testing using slicing", Proceedings of the Conference on Software Maintenance, ICSM '92, pp. 299-308, Nov. 1992.

[21] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing", IEEE Trans. Software Eng., Vol. 27, No. 10, pp. 929-948, Oct. 2001.

[22] H. Agrawal, J. R. Horgan and E. W. Krauser, "Incremental regression testing", Proceedings of the Conference on Software Maintenance, ICSM '93, pp. 348-357,1993.

[23] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, "Test case prioritization: A family of empirical studies", IEEE Transactions on Software Engineering, Vol. 28, No.2, pp. 159-182, Feb. 2002.

[24] Zheng Li, Mark Harman, and Robert M. Hierons, "Search algorithms for regression test case prioritization", IEEE Trans. On Software Engineering, Vol. 33, No.4, pp. 225-237, April 2007.

[25] Ajay Kumar Jha, "A Risk Catalog of Mobile Applications", Florida Institute of Technology Theses, 2007. http://testingeducation.org/articles/AjayJha_Thesis.pdf (truy cập ngày 10/5/2016)

[26] K. K. Aggrawal, Yogesh Singh, A. Kaur, " Code coverage based technique for prioritizing test cases for regression testing", ACM SIGSOFT Software Engineering Notes, Vol. 29 Issue 5, September 2004.

[27] W. E. Wong, J. R. Horgan, S. London and H.Agrawal, "A study of effective regression testing in practice", In Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering (ISSRE' 97), pages 264-274, November 1997.

[28] Yogesh Singh, Arvinder Kaur, Bharti Suri, "A new technique for version-specific test case selection and prioritization for regression testing", Journal of the CSI, *Vol*. 36 No.4, pp. 23-32, October-December 2006.

[29] Mishra, K. K., Kumar, A., & Misra, A. K., "A novel approach for minimizing and prioritizing test suite", Engineering Science Letters, Vol 2014 (2014), Article ID 1.

[30] Badhera, U., Purohit, G. N., & Biswas, D., "Test case prioritization algorithm based upon modified code coverage in regression testing". Int. J. Soft. Eng. Applic, 3, pp. 29-37, 2012.

[31] Todd l. Graves, Mary Jean Harrold, at el, "An Empirical Study of Regression Test Selection Techniques", ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, Pages 184 –208, April 2001.

[32] Cibulski, H., & Yehudai, A., "Regression test selection techniques for test-driven development", Proceedings of Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011, pp. 115-124.

[33] Beginners Guide to Regression Testing for QA Engineers, http://www.codeproject.com/Articles/717524/Beginners-Guide- (truy cập ngày 17/4/2016).

[34] Mobile testing tools, http://www.qatestingtools.com/code.google/googletest, (truy cập ngày 17/4/2016).

[35] Sourcecode Visualizer, http://www.drgarbage.com/sourcecode-visualizer/.

AN EFFECTIVE REGRESSION TESTING TECHNIQUE FOR MOBILE APPLICATION DEVELOPMENT

Huynh Quyet Thang, Nguyen Duc Man, Nguyen Thi Bao Trang, Nguyen Thi Anh Dao

ABSTRACT— *Regression testing of mobile applications is a kind of test to check whether the action as improvements, patches or configuration change did not bring a new regression, or errors, in both functional and non-functionality of a mobile application system. Many mobile applications are widely used in industries like games, banking, retail, tourism, healthcare, health, sports, news, and so on. Therefore, before releasing a mobile application we need to make sure that it works correctly without errors. During the time of improvement mobile application (enhancements, patches or configuration changes...) the new errors will arise in functional and non-functional features of the system. Regression testing is used to ensure that the quality of mobile applications is still guaranteed after any changes in the software. Therefore, regression testing is part of the testing life cycle and is an important test types. However, regression testing is costly for the time and effort but is not allowed to ignore. The problems of choosing the test (test suite), test-cases selection, test case prioritization and how to automatic test-case selection are the solution to improve the efficiency of regression testing. In this paper, we propose a technique that combines selected test cases, identify priorities and to minimize the number of test cases that coverage modified source code of a program. The proposed technique is apllied for mobile applications regression testing in agile process. Experimental results show efficiency for regression testing on cost and time.*