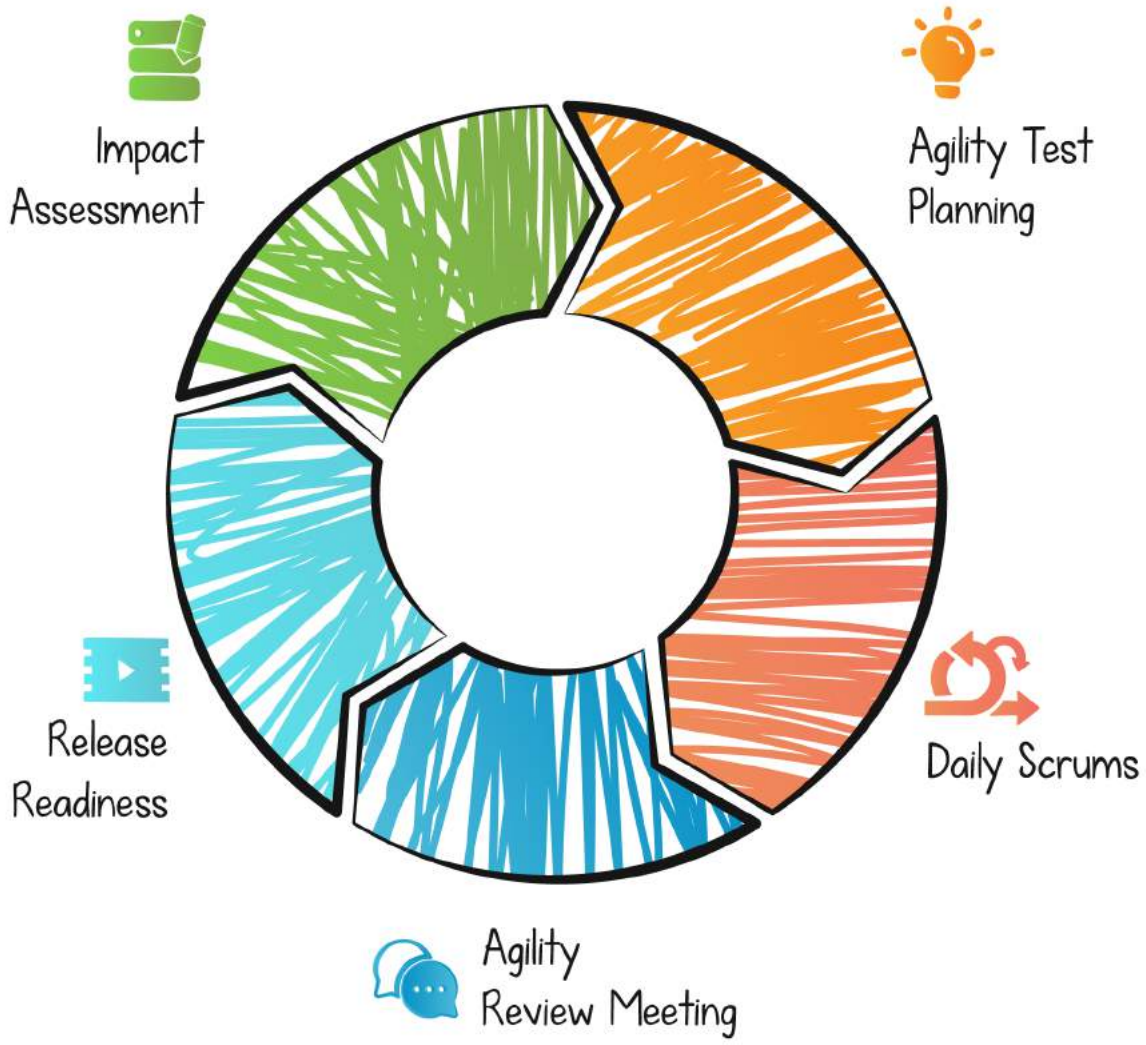
Tạp_Chí
{LậpTrình ():}

Số phát hành Vol.5 06 2020

Tạp chí LẬP TRÌNH

tapchilaptrinh.vn



Agile Testing

SỐ NÀY CÓ GÌ?

Tạp chí lập trình

VOL.5

LỜI MỞ ĐẦU

Quý bạn đọc thân mến,

Phát triển sản phẩm theo tư duy Agile đã và đang là một xu hướng phổ biến ngày nay, chủ đề này đã được đề cập ở trong Vol 4 của Tạp chí Lập trình. Trong Vol 5 này, chúng ta sẽ cùng tiếp nối mạch đó bằng việc tìm hiểu về Agile Testing - một khía cạnh không thể thiếu khi chúng ta muốn triển khai thành công các phương pháp phát triển Agile.

Trong một mô hình phát triển theo tinh thần Agile, chúng ta mong muốn chuyển giao sản phẩm nhanh và thường xuyên mà vẫn đảm bảo được tính ổn định của sản phẩm. Như vậy, với các cách thức làm việc cũ - trong đó khâu kiểm thử được dành riêng cho các tester sau khi các lập trình viên đã hoàn thành nhiệm vụ của mình - đã không còn phù hợp, bởi vì chúng ta sẽ mất rất nhiều thời gian chờ đợi, và đồng thời kéo dài khoảng thời gian từ lúc bắt đầu phát triển một tính năng cho đến lúc bàn giao nó. Agile testing hướng đến việc đưa hoạt động kiểm thử lên song song với hoạt động phát triển các tính năng với mục đích cuối cùng là nhanh chóng phát hành được các tính năng đó và đảm bảo được sự ổn định của sản phẩm.

Vai trò của lập trình viên giờ đây cũng đã khác đi rất nhiều, trong đó nổi bật nhất là để ý nhiều hơn đến tư duy sản phẩm, tư duy phục vụ người dùng, tập trung nhiều hơn đến nghiệp vụ ngay từ sớm, và đồng thời sử dụng được các nền tảng và công cụ hỗ trợ triển khai kiểm thử, nhất là kiểm thử tự động.

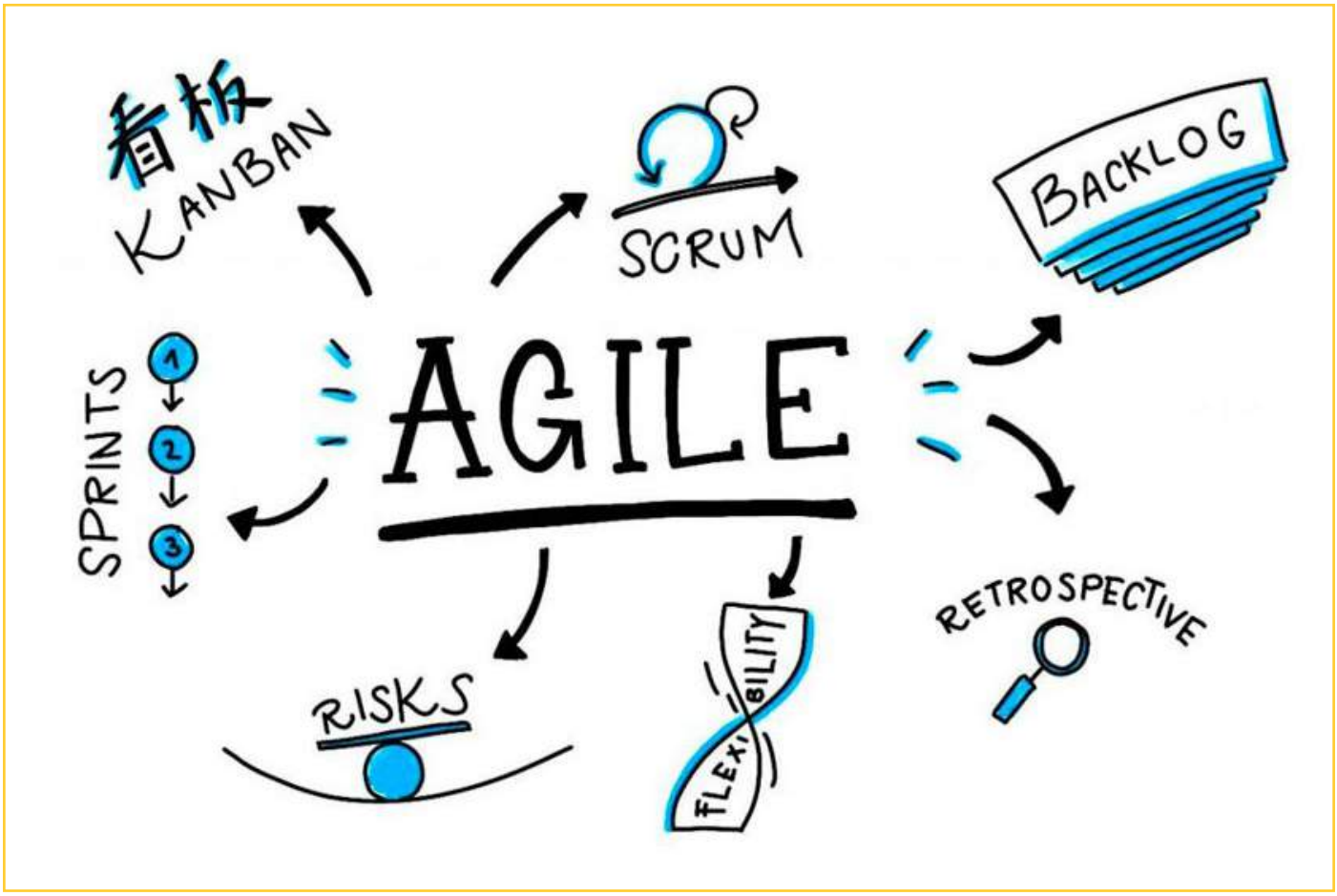
Tạp chí Lập trình kỳ vọng rằng ấn phẩm lần này sẽ cung cấp cho các bạn lập trình viên những góc nhìn đầy đủ nhất về Agile testing, và đồng thời giúp mọi người có những định hướng và lộ trình trong việc bổ sung năng lực của mình để đáp ứng được nhu cầu của thời kỳ mới.

Ban biên tập rất mong nhận được các ý kiến đóng góp của bạn đọc để có thể mở ra nhiều hướng thảo luận sâu hơn về chủ đề này.

Trân trọng.

Ban biên tập Tạp chí Lập trình

KIỂM THỬ AGILE LÀ GÌ?



Mai Công Sơn

Agile Testing là gì?

Agile Testing là một quy trình kiểm thử phần mềm tuân theo các nguyên tắc phát triển phần mềm Agile. Agile testing phù hợp với phương pháp phát triển lặp đi lặp lại, các yêu cầu của khách hàng được phát triển dần dần. Sự phát triển phù hợp với yêu cầu của khách hàng.

Agile Testing là một quá trình liên tục chứ không phải là tuần tự. Việc thử nghiệm bắt đầu khi bắt đầu dự án và có sự tích hợp liên tục giữa thử nghiệm và phát triển. Mục tiêu chung của phát triển và thử nghiệm nhanh là đạt được chất lượng sản phẩm cao.

Agile Testing và WaterFall Testing

Agile Testing được áp dụng trong khi làm việc với Agile trong khi Waterfall Testing được sử dụng trong mô hình Waterfall.

Dưới đây là một số khác biệt chính giữa Agile testing và waterfall tessting.

Agile Testing	*Waterfall Testing*
Agile testing không có cấu trúc so với cách tiếp cận Waterfall và có kế hoạch tối thiểu.	Trong mô hình Waterfall, quy trình thử nghiệm có cấu trúc chặt chẽ hơn và có mô tả chi tiết về giai đoạn thử nghiệm.
Agile testing rất phù hợp cho các dự án nhỏ	Waterfall testing có thể được thông qua cho tất cả các loại dự án.
Kiểm thử bắt đầu khi bắt đầu dự án, lỗi có thể được sửa ở giữa dự án.	Trong waterfall testing, sản phẩm được thử nghiệm vào cuối quá trình phát triển. Đối với bất kỳ thay đổi, dự án phải bắt đầu lại từ đầu.
Tài liệu đơn giản	Các waterfall testing trong cách tiếp cận waterfall đòi hỏi tài liệu phức tạp.
Trong phương pháp này, mỗi lần lặp lại có giai đoạn thử nghiệm riêng. Các bài kiểm tra hồi quy có thể được chạy mỗi khi các chức năng hoặc logic mới được phát hành.	Việc thử nghiệm chỉ bắt đầu sau khi hoàn thành giai đoạn phát triển.
Trong agile testing, các tính năng có thể chuyển của sản phẩm được gửi đến khách hàng khi kết thúc một lần lặp.	Theo cách tiếp cận truyền thống này, tất cả các tính năng được phát triển sẽ được phân phối hoàn toàn sau giai đoạn thực hiện.
Người kiểm thử và nhà phát triển làm việc chặt chẽ trong Agile testing.	Người kiểm thử và nhà phát triển làm việc riêng.
Sự chấp nhận của người dùng được thực hiện vào cuối của mỗi sprint	Sự chấp nhận của người dùng chỉ có thể được thực hiện khi kết thúc dự án.
Tester cần làm việc với các nhà phát triển để phân tích các yêu cầu và lập kế hoạch.	Các nhà phát triển không tham gia vào việc phân tích các yêu cầu và quy trình lập kế hoạch.

Nguyên tắc Agile Testing

Kiểm thử là liên tục: Nhóm Agile kiểm tra liên tục vì đó là cách duy nhất để đảm bảo tiến độ liên tục của sản phẩm.

Phản hồi liên tục – Agile Testing cung cấp phản hồi trên cơ sở liên tục và đây là cách sản phẩm của bạn đáp ứng nhu cầu kinh doanh.

Các thử nghiệm được thực hiện bởi cả nhóm: Trong vòng đời phát triển phần mềm truyền thống, chỉ có nhóm thử nghiệm chịu trách nhiệm thử nghiệm nhưng trong agile testing, các nhà phát triển và nhà phân tích kinh doanh cũng thử nghiệm ứng dụng.

Giảm thời gian phản hồi: Nhóm kinh doanh tham gia vào mỗi lần lặp trong agile testing và phản hồi liên tục rút ngắn thời gian phản hồi.

Mã đơn giản & sạch sẽ: Tất cả các lỗi được đưa ra bởi nhóm agile được sửa trong cùng một lần lặp và nó giúp giữ cho mã sạch và đơn giản hóa.

Tài liệu ít hơn: Các nhóm Agile sử dụng danh sách kiểm tra có thể sử dụng lại, nhóm tập trung vào kiểm tra thay vì các chi tiết ngẫu nhiên.

Thử nghiệm hướng dẫn: Trong các phương pháp agile, thử nghiệm được thực hiện tại thời điểm thực hiện trong khi đó, trong quy trình truyền thống, thử nghiệm được thực hiện sau khi triển khai.

Phương pháp Agile Testing

Có nhiều phương pháp agile tesing như sau:

- Phát triển hướng hành vi
- Phát triển hướng kiểm thử tiếp nhận
- Thử nghiệm thăm dò

Phát triển hướng hành vi (BDD)

Phát triển hướng hành vi (BDD) cải thiện giao tiếp giữa các bên liên quan của dự án để tất cả các thành viên hiểu chính xác từng tính năng trước khi quá trình phát triển bắt đầu. Có sự liên lạc dựa trên ví dụ liên tục giữa các nhà phát triển, người thử nghiệm và nhà phân tích kinh doanh.

Các ví dụ được gọi là kịch bản được viết theo định dạng đặc biệt gọi là cú pháp Gherkin Given/When/Then. Các kịch bản chứa thông tin về cách một tính năng nhất định sẽ hoạt động trong các tình huống khác nhau với các tham số đầu vào khác nhau. Chúng được gọi là các thông số kỹ thuật có thể thực hiện được bởi vì nó bao gồm cả thông số kỹ thuật và đầu vào cho các thử nghiệm tự động.

Phát triển hướng kiểm thử tiếp nhận (ATDD)

ATDD tập trung vào việc liên quan đến các thành viên trong nhóm với các quan điểm khác nhau như khách hàng, nhà phát triển và người thử nghiệm. Ba cuộc họp của Amigos được tổ chức để hình thành các thử nghiệm chấp nhận kết hợp các quan điểm của khách hàng, phát triển và thử nghiệm. Khách hàng tập trung vào vấn đề cần giải quyết, sự phát triển tập trung vào cách giải quyết vấn đề trong khi thử nghiệm tập trung vào những gì có thể sai. Các thử nghiệm chấp nhận là một đại diện cho quan điểm của người dùng và nó mô tả cách hệ thống sẽ hoạt động. Nó cũng giúp xác minh rằng các chức năng hệ thống như nó được yêu cầu. Trong một số trường hợp kiểm tra chấp nhận được tự động.

Thử nghiệm thăm dò

Trong loại thử nghiệm này, giai đoạn thiết kế thử nghiệm và thực hiện thử nghiệm đi đôi với nhau. Kiểm tra thăm dò nhấn mạnh phần mềm làm việc trên tài liệu toàn diện. Các cá nhân và tương tác quan trọng hơn quá trình và công cụ. Hợp tác khách hàng giữ giá trị lớn hơn so với đàm phán hợp đồng. Thử nghiệm thăm dò là thích ứng hơn với những thay đổi. Trong thử nghiệm này xác định chức năng của một ứng dụng bằng cách khám phá ứng dụng. Những người thử nghiệm cố gắng tìm hiểu ứng dụng, và thiết kế và thực hiện các kế hoạch kiểm tra theo kết quả của họ.

Ưu điểm của Agile Testing

Những lợi ích của phương pháp agile testing như sau:

- Nó tiết kiệm thời gian và tiền bạc
- Kiểm tra Agile làm giảm tài liệu
- Nó linh hoạt và có khả năng thích ứng cao với những thay đổi
- Nó cung cấp một cách để nhận phản hồi thường xuyên từ người dùng cuối

Kế hoạch Agile testing

- Trong agile testing, kế hoạch kiểm tra được viết cũng như cập nhật cho mỗi bản phát hành. Một kế hoạch agile testing bao gồm:
- Kế hoạch agile testing
- Phạm vi thử nghiệm
- Hợp nhất các chức năng mới sẽ được thử nghiệm
- Các loại thử nghiệm / Cấp độ thử nghiệm
- Kiểm tra hiệu suất
- Cân nhắc về kiến trúc
- Kế hoạch cho rủi ro
- Quy hoạch tài nguyên
- Bàn giao sản phẩm & mốc

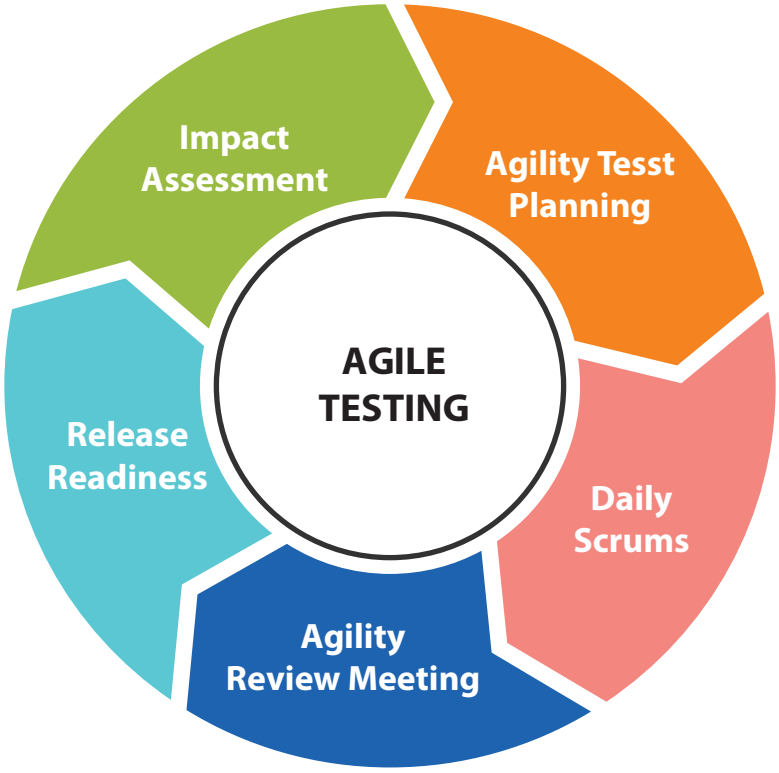
Vòng đời kiểm tra Agile

Vòng đời thử nghiệm nhanh bao gồm 5 giai đoạn sau:

- Đánh giá tác động
- Lập kế hoạch Agile testing
- Sẵn sàng phát hành
- Scrum hàng ngày
- Kiểm thử linh hoạt

Kết luận

Agile testing không chỉ tạo điều kiện phát hiện sớm các lỗi mà còn giảm chi phí lỗi bằng cách sửa chúng sớm. Cách tiếp cận này cũng mang lại một cách tiếp cận lấy khách hàng làm trung tâm bằng cách cung cấp một sản phẩm chất lượng cao càng sớm càng tốt.



Start-Test Plan Meet Up
All stakeholdes come together to plan the schedule of testing process, meeting frequency and deliverables

Approval Meetings for Deployment
At this stage we review the features that have been developed/ implemented are ready to go live or not.

Daily Standup Meetings
This includes the everyday standup morning meeting to catch up on the status of testing and set the goals for whole day.

Impact Assessment
Gather inputs from stake holders and users, this will act as feedback for next deployment cycle..

Agility Review Meeting
Weekly review meetings with stakeholders meet to review and assess the progress against milestones.

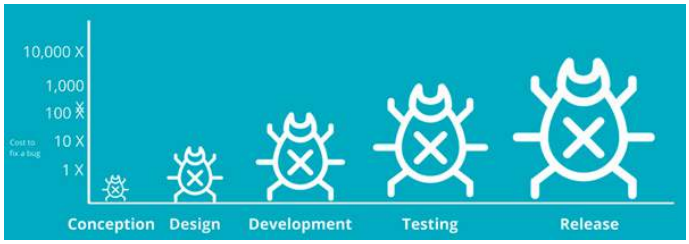
TỰ ĐỘNG HOÁ HOẠT ĐỘNG KIỂM THỬ: NHU CẦU CỦA PHÁT TRIỂN PHẦN MỀM HIỆN ĐẠI

Nguyễn Khắc Nhật

Lỗi trong các sản phẩm dẫn đến thiệt hại vô cùng lớn cho các nhà sản xuất và cho xã hội, không riêng gì trong ngành phần mềm. Hẳn chúng ta dễ dàng tìm thấy các thông tin liên quan đến lỗi trên các dòng xe hơi của các hãng như Toyota, Ford, Honda, Mitsubishi... dẫn đến việc phải triệu hồi và khắc phục hàng triệu chiếc xe. Hoặc sự cố pin gây cháy nổ của dòng điện thoại Galaxy Note 7 của Samsung khi vừa mới ra mắt, mà theo ước tính có thể gây thiệt hại lên đến 17 tỉ USD cho hãng. Hay như lỗi phần mềm của dòng máy bay Boing 737 MAX dẫn đến các tai nạn kinh hoàng ở Indonesia và Ethiopia, không chỉ gây thiệt hại về kinh tế mà còn để lại những hậu quả không thể khắc phục được - hàng trăm người đã thiệt mạng do các sự cố này.

Hình 1: Sự cố của Boing 737 MAX tại Ethiopia

Bởi những hậu quả vô cùng lớn như vậy, cho nên nhiệm vụ đảm bảo chất lượng luôn là ưu tiêu hàng đầu của các nhóm phát triển sản phẩm, trong đó hoạt động kiểm thử được coi là một chốt chặn quan trọng để phát hiện và xử lí các lỗi tiềm ẩn. Cũng bởi vì thế mà trong ngành phần mềm nói riêng, chúng ta luôn cố gắng để đưa ra các phương pháp, kỹ thuật và công cụ hiệu quả nhất để phát hiện và xử lí lỗi càng sớm càng tốt.



Hình 2: Lỗi phát hiện càng muộn thì chi phí sửa chữa càng cao X tại Ethiopia

Có một nguyên tắc mang tính chất hiển nhiên trong phát triển sản phẩm: Các lỗi phát hiện càng muộn thì chi phí sửa chữa càng cao. Chẳng hạn, đối với các lỗi được phát hiện trong giai đoạn thiết kế thì có thể chỉ mất một ít thời gian để điều chỉnh, nếu lỗi đó không được phát hiện và chuyển sang giai đoạn sản xuất thì chi phí sẽ tăng lên nhiều, và cứ như thế, nếu đến khi sản phẩm đã được phát hành mà lại xuất hiện lỗi thì chi phí sẽ cực kỳ lớn. Ngoài những ví dụ như đã được nhắc đến ở phần đầu bài viết, hãy thử tưởng tượng một hệ thống lớn như của Facebook hoặc Google mà bị dừng hoạt động trong dăm phút thì thiệt hại sẽ lên đến chừng nào – chắc chắc sẽ là một con số vô cùng khổng lồ. Do nguyên tắc này, chúng ta luôn cố gắng để các hoạt động kiểm thử được diễn ra càng sớm càng tốt.

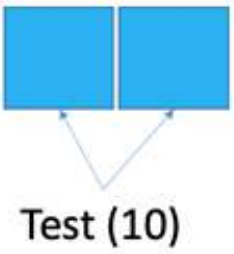
Trước đây, kiểm thử gần như là công đoạn sau cùng của quá trình phát triển. Giờ đây, chúng ta muốn hoạt động này được diễn ra sớm hơn, song song hoặc thậm chí là trước cả việc viết mã. Cũng bởi vì thế, một số thao tác kiểm thử đã được chuyển sang cho lập trình viên, thay vì gán riêng cho các tester như trước. Ngày nay, việc lập trình viên thực hiện kiểm thử đơn vị (unit testing) hoặc kiểm thử tích hợp (integration testing) đã trở nên phổ biến.

Kiểm thử là công việc lặp đi lặp lại

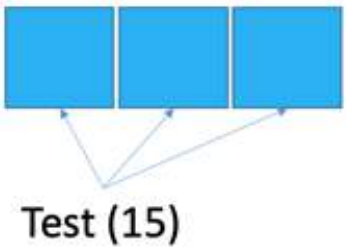
Giả sử quá trình phát triển các tính năng của một phần mềm và kiểm thử nó được diễn ra như minh hoạ sau đây.

Bước 1: Chức năng đầu tiên, với 5 kịch bản kiểm thử

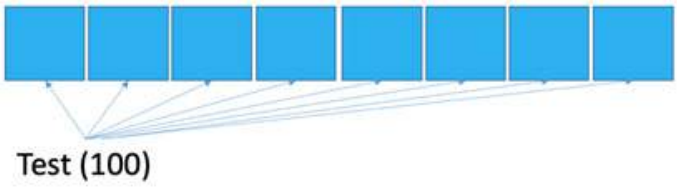
Bước 2: Thêm một tính năng nữa, và chúng ta cần thực hiện 10 kịch bản kiểm thử, bao gồm cả 5 kịch bản kiểm thử của chức năng trước đó, chứ không chỉ là các kịch bản của chức năng mới.



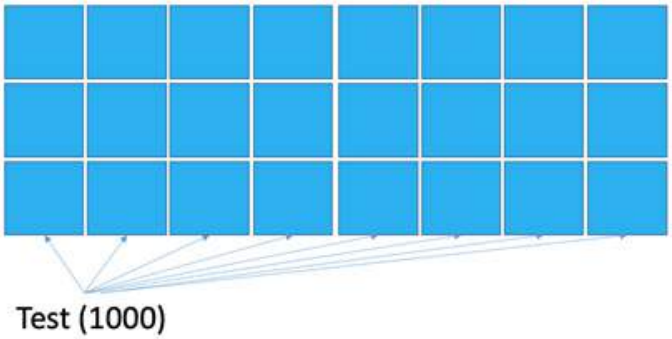
Bước 3: Thêm một tính năng nữa, và chúng ta cần thực hiện 15 kịch bản kiểm thử.



...sau một thời gian, chức năng càng được thêm nhiều vào, chúng ta cần thực hiện 100 kịch bản kiểm thử mỗi khi phát triển một tính năng mới (hoặc là chỉnh sửa một tính năng nào đó)



...và con số cứ thế tăng lên đến hàng nghìn kịch bản kiểm thử



Với một quá trình như thế, điều gì sẽ diễn ra nếu chúng ta phải thực hiện kiểm thử một cách thủ công?

Thứ nhất, chúng ta dễ dàng nhận thấy rằng đây là các thao tác gây hao tốn rất nhiều công sức. Sau khi chỉnh sửa hoặc thêm một chức năng bất kỳ, chúng ta lại phải kiểm thử hàng loạt các kịch bản đã làm trước đó.

Thứ hai, việc phát hành phần mềm sẽ bị kéo dài rất lâu chỉ bởi vì những thao tác kiểm thử này rất mất thời gian. Chúng ta sẽ khó để có thể phát hành phần mềm theo tần suất tuần, ngày hoặc thậm chí là giờ được.

Do vậy, tự động hoá hoạt động kiểm thử là một giải pháp giúp chúng ta rút ngắn được thời gian kiểm thử và giảm thiểu được chi phí nỗ lực rất nhiều so với trước kia.

Hãy thử tính toán, một thao tác kiểm thử thủ công mất vài chục phút thì bây giờ có thể được thực hiện với thời lượng chục giây. Một chuỗi kiểm thử thủ công mất vài giờ thì bây giờ có thể được thực hiện với thời lượng dăm phút. Hơn nữa, sức máy thì không biết mệt, trong khi sức người thì rất không ổn định. Như vậy, với hàng nghìn, thậm chí là chục nghìn kịch bản kiểm thử thì chúng ta đã tiết kiệm được bao nhiêu công sức và thời gian?

Vậy các lập trình viên và kiểm thử viên cần làm gì để bắt đầu với Kiểm thử Tự động? Trước tiên, cần tìm hiểu để thay đổi tư duy về phát triển phần mềm theo hướng hiện đại và linh hoạt. Sau đó, tìm hiểu về một số phương pháp triển khai kiểm thử tự động. Cuối cùng, học cách sử dụng một số phần mềm, nền tảng hoặc công cụ hỗ trợ kiểm thử tự động mà chúng ta dễ dàng tìm thấy được, cho dù chúng ta đang sử dụng bất cứ một ngôn ngữ lập trình nào hoặc nền tảng nào.

QUnit

Những bước chân TDD đầu tiên trên JavaScript

Dư Thanh Hoàng

TDD là gì?

TDD (Test Driven Development) là một phương thức làm việc, hay một quy trình viết mã hiện đại. Lập trình viên sẽ thực hiện thông qua các bước nhỏ (BabyStep) và tiến độ được đảm bảo liên tục bằng cách viết và chạy các bài test tự động (automated tests). Quá trình lập trình trong TDD cực kỳ chú trọng vào các bước liên tục sau:

1. Viết 1 test case cho hàm mới. Đảm bảo rằng test sẽ fail.
2. Chuyển qua viết code sơ khai nhất cho hàm đó để test có thể pass.
3. Tối ưu hóa đoạn code của hàm vừa viết sao cho đảm bảo test vẫn pass và tối ưu nhất cho việc lập trình kế tiếp
4. Lặp lại cho các hàm khác từ bước 1

Phát triển hướng kiểm thử TDD (Test-Driven Development) là một phương pháp phát triển phần mềm trong đó kết hợp phương pháp Phát triển kiểm thử trước (Test First Development) và điều chỉnh lại mã nguồn (Refactoring).

Mục tiêu quan trọng nhất của TDD là hãy nghĩ về kết qủa của bạn trước khi viết mã nguồn cho chức năng. Nhìn chung, mục tiêu của TDD là viết mã nguồn sáng sủa, rõ ràng và hạn chế lỗi, dễ dàng mở rộng.

QUnit là gì?

QUnit là một framework mạnh, miễn phí và dễ sử dụng để triển khai Kiểm thử Đơn vị (Unit Testing) trong JavaScript. Framework này đã được dùng cho các dự án jQuery, jQuery UI và jQuery Mobile cũng như có thể dùng cho tất cả các mã nguồn JavaScript nói chung.

Cài đặt QUnit

QUnit là một thư viện độc lập, chỉ cần một tệp JavaScript (qunit.js) và một tệp CSS (qunit.css). Bạn có thể tải chúng xuống từ trang QUnit hoặc sao chép chúng từ kho lưu trữ QUnit GitHub , như sau:

```
git clone git://github.com/jquery/qunit.git
```

Hoặc đơn giản hơn, chúng ta có thể nhúng CDN của QUnit vào trang HTML của chúng ta:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=”utf-8”>
 <meta name=”viewport” content=”width=de-
vice-width”>
  <title>QUnit Example</title>
  <link rel=”stylesheet” href=”https://
code.jquery.com/qunit/qunit-2.10.0.css”>
</head>
<body>
  <div id=”qunit”></div>
  <div id=”qunit-fixture”></div>
  <script src=”https://code.jquery.com/
qunit/qunit-2.10.0.js”></script>
  <script src=”tests.js”></script>
  </body>
</html>
```

Sau đó, chúng ta tạo 1 file js là “tests.js” và viết những dòng test case đầu tiên vào đó:

```
QUnit.test( “hello test”, function( assert
) {
   assert.ok( 1 == “1”, “Passed!” );
});
```

Cuối cùng, hãy chạy thử file HTML của bạn và xem kết quả nhé:

Yeah, bạn đã chạy thành công 1 test case đơn giản đầu tiên, bây giờ hãy thử nâng cấp 1 chút, viết test cho 1 hàm của bạn nhé. Bạn tạo file “app.js” chứa các hàm mà bạn muốn test, và tạo sẵn vào đó hàm sayHello() như sau:

```
function sayHello(yourName) {
   return yourName;
}
```

Tiếp đó, Viết một test case trong file tests.js dành cho hàm sayHello()

```
Qunit.test( “test sayHello”, function(as-
sert) {
  let expected = “Hello World”;
  let result =  sayHello(“World”);
  assert.ok(expected == result,”Passed!”);
});
```

Chạy thử HTML của bạn sẽ có kết quả như sau

Vậy là bạn đã thành công với bước đầu có được test case “fail” với QUnit, công việc của bạn là hoàn chỉnh hàm sayHello() để “pass” được cái test này.

```
function sayHello(yourName) {
   return “Hello “ + yourName;
}
```

Xong rồi chạy lại trang HTML của bạn, nếu hiển thị như này:

Xanh tức là bạn đã “pass” trường hợp đó rồi. Nếu báo đỏ thì bạn hãy kiểm tra lại hàm sayHello() xem có vấn đề gì không nhé. Sau khi pass rồi thì chúng ta viết tiếp những test case mới.

Đến đây tôi đã hoàn tất việc giới thiệu với các bạn về QUnit và một hướng dẫn nho nhỏ để bạn có thể bắt đầu làm Unit Test hay xa hơn nữa là TDD với JavaScript. Bạn có thể tìm hiểu sâu hơn về QUnit tại trang web http://qunitjs.com.

Chúc bạn thành công và có những mã nguồn JavaScript chất lượng.

Nguồn tham khảo: https://medium.com/@hoangdt/qunit-những-bước-chân-tdd-đầu-tiên-trên-javascript-684a8886820c

BÀI QUYỀN
BOWLING GAME

Nguyễn Bình Sơn

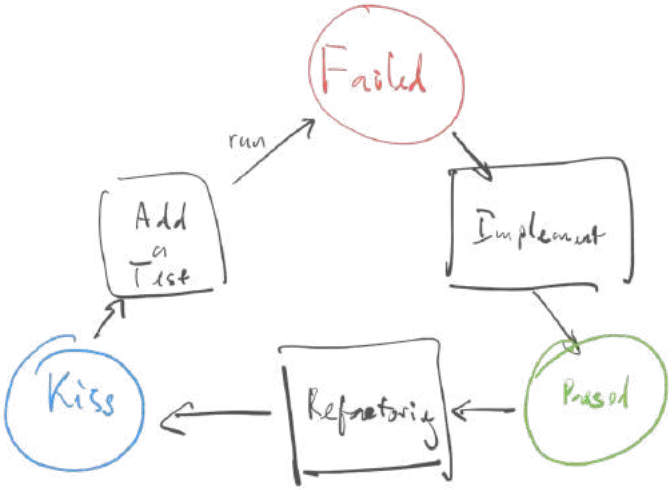
Bowling Game là một bài kata kinh điển của hoạt động Coding Dojo. Bài kata này rất phù hợp để thực hành kỹ thuật TDD, Baby Steps và Refactoring.

Về TDD

TDD (Test Driven Development – Phát triển (mà trong đó việc phát triển) được lái bởi Kiểm thử) là một phương pháp tiếp cận để phát triển phần mềm. Nói cách khác, là một cách để suy nghĩ về requirement (yêu cầu) cũng như thiết kế trước khi viết các mã triển khai.

Có một mô hình giải thích khác về TDD mà trong đó coi TDD là một kỹ thuật lập trình. Tuy vậy trong bài viết này sử dụng thuật ngữ TDD theo mô hình "cách nghĩ".

Cụ thể, "cách nghĩ" theo phương pháp TDD được diễn giải như sau:



TDD explained

1. RED: Quá trình phát triển bắt đầu bằng thao tác Add a Test: bổ sung một ca kiểm thử. Chưa có mã triển khai tương ứng nên ca kiểm thử này sẽ Failed. Nhưng ngay từ thời điểm đó ca kiểm thử đã có tác dụng mô tả yêu cầu cũng như thiết kế của mã triển khai.
2. GREEN: Mã triển khai tương ứng với ca kiểm thử được bổ sung. Mã nguồn chuyển sang giai đoạn Passed (vượt qua kiểm thử). Các mã triển khai nội bộ của chức năng có thể linh hoạt, nhưng thiết kế của mã thì tuân theo mô tả của kiểm thử.
3. BLUE: Mã nguồn được tái cấu trúc. Mục đích của việc tái cấu trúc là đưa mã nguồn tới trạng thái sẵn sàng để bổ sung tính năng mới. Clean Code, SOLID, Clean Architecture, KISS, Simple Design là các quy tắc và tiêu chuẩn thường được áp dụng trong bước này.
4. Quá trình RGB (RedGreenBlue) được lặp đi lặp lại cho đến khi các chức năng được triển khai hoàn thiện.

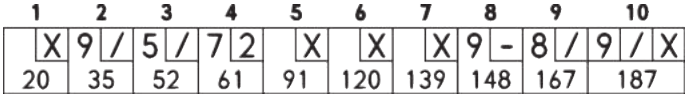
Về các Coding Dojo Kata

Kata (bài quyền) trong một buổi Coding Dojo được định nghĩa là những bài toán được thiết kế cho lập trình viên để luyện tập một kỹ năng trong lập trình, thông qua thực hành và lặp lại.

Các vấn đề (bài toán) được sử dụng làm kata thường đủ nhỏ (không quá khó) để giải quyết nhưng đủ thách thức để người tập không có khả năng hoàn thành trong thời gian cho phép. Đây là thiết kế của Coding Dojo nhằm giúp người tham gia tập trung vào luyện tập kỹ năng thay vì hướng tới mục đích "xong việc".

Về Vấn đề Bowling

Vấn đề Bowling xuất phát từ luật của trò chơi Bowling.



Một ván chơi diễn ra trong 10 frame. Mỗi frame, người chơi có 2 lượt ném để hạ đổ 10 pin. Điểm của frame là tổng số pin bị hạ đổ cộng thêm điểm thưởng từ trike và spare.

Spare (các ký hiệu / trong hình trên)xảy ra khi người chơi hạ thành công cả 10 pin sau hai lượt ném. Điểm thưởng cho spare là số pin bị hạ tại lượt ném ngay sau đó.

Strike (các ký hiệu X trong hình trên) xảy ra khi người chơi hạ thành công cả 10 pin ngay từ lượt ném đầu tiên, và theo đó kết thúc frame trong một roll duy nhất. Điểm thưởng cho strike là tổng số pin bị hạ tại hai lượt ném ngay sau đó.

Người chơi đạt spare hay strike tại frame cuối cùng sẽ được ném thêm các lượt ném phụ để nhận trọn các lượt thưởng. Theo đó frame này sẽ kết thúc sau ba lượt ném.

Bài kata Bowling Game yêu cầu lập trình viên viết chương trình để tính điểm cho trò chơi này.

Bài quyền

Thiết kế ban đầu

Mọi tính thông tin cần thiết để có được điểm của một ván chơi nằm trọn trong một ván chơi.

```
BowlingGame game = new BowlingGame();
```

Rõ ràng, điểm của một game chỉ có thể tính được tại thời điểm mà kết quả của mọi roll đều đã rõ ràng. Giả sử ta thiết kế phương thức getScore dùng để tính về điểm số của game, getScore chỉ làm được điều đó khi nó đã nhận được đầy đủ thông tin về các roll. Chẳng hạn:

```
BowlingGame game = new BowlingGame();
int[] rolls = // a dummy rolls array
int score = game.getScore(rolls);
```

Cách thiết kế này không ổn, hãy để ý game.getScore(rolls), ở đây game là một thực thể (entity) nhưng lại đang được sử dụng như một service cung cấp khả năng tính điểm.

Một thiết kế tốt hơn là đặt một phương thức roll để nhận thông tin về các roll, và đặt một phương thức score được gọi khi các roll đều đã được nhập liệu hoàn chỉnh.

```
BowlingGame game = new BowlingGame();
repeat () {
    int pins = // a dummy pins;
    game.roll(pins);
}
int score = game.score();
```

Đây là thiết kế tối thiểu mà chúng ta có thể có thể bắt đầu phát triển chương trình.

Bắt đầu

Bài viết này dựa trên tiền đề rằng chương trình được viết bằng ngôn ngữ Java và test framework được sử dụng là JUnit.

Tạo một kiểm thử đơn vị

Với JUnit, một kiểm thử đơn vị là một class trong đó có các ca kiểm thử là những phương thức được chú thích @Test. Dưới đây là kiểm thử đơn vị cho chương trình BowlingGame với một ca kiểm thử "dummy" dùng để xác nhận rằng test framework hoạt động ổn định.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Asser-
tions.*;
public class BowlingGameTest {
    @Test
    void testAddition() {
        assertEquals(2, 1 + 1);
    }
}
```

Thực thi kiểm thử cho kết quả

```
+-- JUnit Jupiter [OK]
| '-- BowlingGameTest [OK]
```

Ca kiểm thử đầu tiên

Kiểm thử đầu tiên mô tả rằng "nếu người chơi ném trượt tất cả các roll thì điểm số sẽ là 0".

Mô tả class chức năng

```
@Test
void testAllMissedGame() {
    BowlingGame game = new BowlingGame();
}
```

Mã triển khai để vượt qua kiểm thử:

```
public class BowlingGame {
}
```

Mô tả thao tác nhập thông tin các roll

Để mô tả một ván chơi mà tất cả các roll (tổng cộng 20) đều trượt, ở đây sử dụng một phép lặp:

```
@Test
void testAllMissedGame() {
    BowlingGame game = new BowlingGame();
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
}
```

Mã triển khai để vượt qua kiểm thử (để đơn giản, từ mục này về sau mã triển khai sẽ được viết ra mà không có chú thích gì thêm):

```
public class BowlingGame {
    public void roll(int pins) {
    }
}
```

Mô tả thao tác tính điểm

```
@Test
void testAllMissedGame() {
    BowlingGame game = new BowlingGame();
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    int score = game.score();
}
```

```
public class BowlingGame {
    public void roll(int pins) {
    }
    public int score() {
        return -1;
    }
}
```

Mô tả điểm số mong muốn

```
@Test
void testAllMissedGame() {
    BowlingGame game = new BowlingGame();
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    assertEquals(0, game.score());
}
```

```
public int score() {
    return 0;
}
```

Tới bước này, chương trình đã có khả năng tính đúng số điểm của những ván chơi mà tất cả các roll đều trượt.

Ca kiểm thử thứ hai

Ca kiểm thử thứ hai mô tả trường hợp mà "người chơi ném đổ một số pin nào đó, nhưng không có điểm thưởng", chúng ta chọn một trường hợp điển hình là "người chơi chỉ ném đổ một pin mỗi roll".

Mô tả một game ăn một điểm mỗi roll

```
@Test
void testAllMissedGame() {
    BowlingGame game = new BowlingGame();
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    assertEquals(0, game.score());
}

@Test
void testAllOneGame() {
    BowlingGame game = new BowlingGame();
    for (int i = 0; i < 20; i++) {
        game.roll(1);
    }
    assertEquals(20, game.score());
}
```

```
private int score;
public void roll(int pins) {
    score += pins;
}
public int score() {
    return score;
}
```

Trước khi triển khai chức năng tiếp theo, hãy để ý có dấu hiệu mã xấu

- Trùng lắp mã khởi tạo đối tượng game
- Trùng lắp mã lặp lại thao tác roll

Khử trùng lắp cho mã khởi tạo đối tượng game

Trùng lắp này có thể được khử bằng một phương thức setup mà hầu như mọi test framework đều hỗ trợ. Trong JUnit, phương thức setup được mô tả bởi chú thích @BeforeEach:

```
private BowlingGame game;
@BeforeEach
void setup() {
    game = new BowlingGame();
}
@Test
void testAllMissedGame() {
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    assertEquals(0, game.score());
}
@Test
void testAllOneGame() {
    for (int i = 0; i < 20; i++) {
        game.roll(1);
    }
    assertEquals(20, game.score());
}
```

Khử trùng lắp cho thao tác roll nhiều lần

Trùng lắp này có thể được khử bằng kỹ thuật tách hàm:

```
private BowlingGame game;
@BeforeEach
void setup() {
    game = new BowlingGame();
}
@Test
void testAllMissedGame() {
    rollMany(0, 20);
    assertEquals(0, game.score());
}
@Test
void testAllOneGame() {
    rollMany(1, 20);
    assertEquals(20, game.score());
}
private void rollMany(int pins, int times) {
    for (int i = 0; i < times; i++) {
        game.roll(pins);
    }
}
```

Ca kiểm thử thứ ba

Ca kiểm thử thứ ba hướng đến việc mô tả chức năng tính điểm thưởng cho spare.

Mô tả một spare

```
@Test
void testSpare() {
    game.roll(3);
    game.roll(7); // spare!
    game.roll(4);
    rollMany(0, 17);
    assertEquals(18, game.score());
}
```

Thiết kế hiện tại không thể vượt qua được kiểm thử này vì những lý do sau đây:

- Score đang được tính toán "lâm thời", trong khi để tính điểm thưởng thì cần dựa vào roll "tương lai" (hoặc "quá khứ", tùy cách hiểu).
- Để giải quyết vấn đề trên thì lịch sử kết quả của các roll cần được lưu lại, nhưng phương thức roll hiện tại không làm điều đó.
- Việc tính toán điểm số cần dựa trên lịch sử, nhưng phương thức score hiện tại không làm điều đó.

Để giải quyết vấn đề trên, cần tái thiết kế mã nguồn. Để có thể tái thiết kế an toàn ta cần giữ lại hai ca kiểm thử đầu tiên. Ca kiểm thử thứ ba tạm thời chưa thể pass được và cần phải đặt sang một bên.

```
// @Test
// void testSpare() {
// ...
```

Lưu giữ lịch sử các roll

Phương thức roll sẽ đảm nhiệm chức năng lưu giữ lịch sử ném:

```
private int score = 0;
private int[] rolls = new int[21];
private int currentRoll = 0;

public void roll(int pins) {
    rolls[currentRoll++] = pins;
    score += pins;
}

public int score() {
    return score;
}
```

Tính điểm số dựa trên lịch sử ném

Các mã đảm nhiệm chức năng tính điểm sẽ được bỏ ra khỏi phương thức roll. Phương thức score sẽ đảm nhiệm tính năng này, và nó thực hiện dựa theo lịch sử ném:

```
private int[] rolls = new int[21];
private int currentRoll = 0;

public void roll(int pins) {
    rolls[currentRoll++] = pins;
}

public int score() {
    int score = 0;
    for (int i = 0; i < rolls.length; i++) {
        score += rolls[i];
    }
    return score;
}
```

Tới lúc này thì ca kiểm thử số ba có thể được gỡ comment.

Phát hiện spare

Thao tác cộng điểm thưởng cho spare bắt đầu bằng việc phát hiện ra sự kiện spare:

```
public int score() {
    // ...
    for (int i = 0; i < rolls.length; i++) {
        if (rolls[i] + rolls[i+1] == 10) //
spare score += ...
        // score += rolls[i];
    }
    // ...
}
```

Giải pháp này không sử dụng được bởi thông tin i không mô tả được roll hiện tại là roll bắt đầu hay kết thúc của một frame. Thiết kế hiện tại vẫn chưa đáp ứng được ca kiểm thử số 3. Chúng ta cần một biến đếm có khả năng đại diện cho một frame, không phải một roll.

Duyệt qua các frame

Ca kiểm thử số 3 cần được comment lại một lần nữa. Phương thức score được tái cấu trúc để duyệt qua từng frame một:

```
public int score() {
    int score = 0;
    int i = 0;
    for (int frame = 0; frame < 10; frame++) {
        score += rolls[i] + rolls[i + 1];
        i += 2;
    }
    return score;
}
```

Tới lúc này thì thiết kế đã sẵn sàng để vượt qua ca kiểm thử thứ ba.

Vượt qua ca kiểm thử thứ 3

Gỡ bỏ comment cho ca kiểm thử thứ ba và bổ sung mã tính điểm thưởng cho spare:

```
public int score() {
  int score = 0;
  int i = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (rolls[i] + rolls[i + 1] == 10) { //
     spare
      score += 10 + rolls[i + 2];
      i += 2;
    } else {
      score += rolls[i] + rolls[i + 1];
      i += 2;
    }
  }
  return score;
}
```

Kiểm thử số ba đã được vượt qua, nhưng những mã xấu sau vẫn hiện diện:

- Tên biến không có tính mô tả (biến i) tại mã triển khai
- Sự tồn tại của comment tại mã triển khai (để giải thích cho phép so sánh magic == 10)
- Sự tồn tại của comment tại mã kiểm thử (để giải thích cho cặp số magic 3-7)

Đặt lại tên biến mặc tả

Biến i của phương thức score đang mô tả vị trí bắt đầu của frame trong lịch sử các roll, có thể được đặt lại tên thành frameIndex.

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (rolls[frameIndex] + rolls[frameIndex
    + 1] == 10) { // spare
      score += 10 + rolls[frameIndex + 2];
      frameIndex += 2;
    } else {
      score += rolls[frameIndex] + rolls[fr
      ameIndex + 1];
      frameIndex += 2;
    }
  }
  return score;
}
```

Khử comment tại mã triển khai

Comment tại mã triển khai có thể được khử bằng cách đặt tên cho biểu thức so sánh magic rolls[frameIndex] + rolls[frameIndex + 1] == 10. Biểu thức này này mô tả dấu hiệu nhận diện một spare. Mục tiêu "đặt tên" có thể được thực hiện bằng kỹ thuật tách phương thức:

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (isSpare(frameIndex)) {
      score += 10 + rolls[frameIndex + 2];
      frameIndex += 2;
    } else {
      score += rolls[frameIndex] + rolls[fr
       ameIndex + 1];
      frameIndex += 2;
    }
  }
  return score;
}

private boolean isSpare(int frameIndex) {
   return rolls[frameIndex] + rolls[frameIn
   dex + 1] == 10;
}
```

Khử comment tại mã kiểm thử

Tương tự, cặp magic roll tại mã kiểm thử có thể được khử bằng kỹ thuật tách phương thức:

```
@Test
void testSpare() {
    rollSpare();
    game.roll(4);
    rollMany(0, 17);
    assertEquals(18, game.score());
}

private void rollSpare() {
    game.roll(3);
    game.roll(7);
}
```

Tới lúc này thì mã nguồn đã sẵn sàng cho kiểm thử tiếp theo.

Ca kiểm thử thứ tư

Mục đích của kiểm thử thứ tư là mô tả khả năng tính điểm thưởng trong trường hợp người chơi ăn strike.

Kiểm thử strike

```
@Test
void testStrike() {
    game.roll(10); // strike
    game.roll(1);
    game.roll(2);
    rollMany(0, 16);
    assertEquals(16, game.score());
}
```

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (rolls[frameIndex] == 10) { // strike
      score += 10
        + rolls[frameIndex + 1]
        + rolls[frameIndex + 2];
      frameIndex++;
    } else if (isSpare(frameIndex)) {
      score += 10 + rolls[frameIndex + 2];
      frameIndex += 2;
    } else {
      score += rolls[frameIndex] + rolls[fr
       ameIndex + 1];
      frameIndex += 2;
    }
  }
  return score;
}
```

Nhờ có các tái cấu trúc trước đó, ca kiểm thử thứ tư được vượt qua rất dễ dàng. Nhưng những mã xấu sau vẫn hiện diện:

- Comment tại mã triển khai
- Comment tại mã kiểm thử

Khử comment tại mã triển khai

Comment tại mã triển khai hiện diện nhằm giải thích cho biểu thức magic rolls[frameIndex] == 10. Biểu thức này mô tả dấu hiệu nhận diện một strike. Dấu hiệu này có thể được đặt tên bằng kỹ thuật tách phương thức:

```
public int score() {
    int score = 0;
    int frameIndex = 0;
   for (int frame = 0; frame < 10; frame++) {
        if (isStrike(frameIndex)) {
            // ...
    }
    return score;
}

private boolean isStrike(int frameIndex) {
    return rolls[frameIndex] == 10;
}
```

Khử comment tại mã kiểm thử

Comment tại mã kiểm thử nhằm mô tả magic roll 10. Roll này có thể được đặt tên bằng kỹ thuật tách phương thức:

```
@Test
void testStrike() {
    rollStrike();
    game.roll(1);
    game.roll(2);
    // ...
}

private void rollStrike() {
    game.roll(10);
}
```

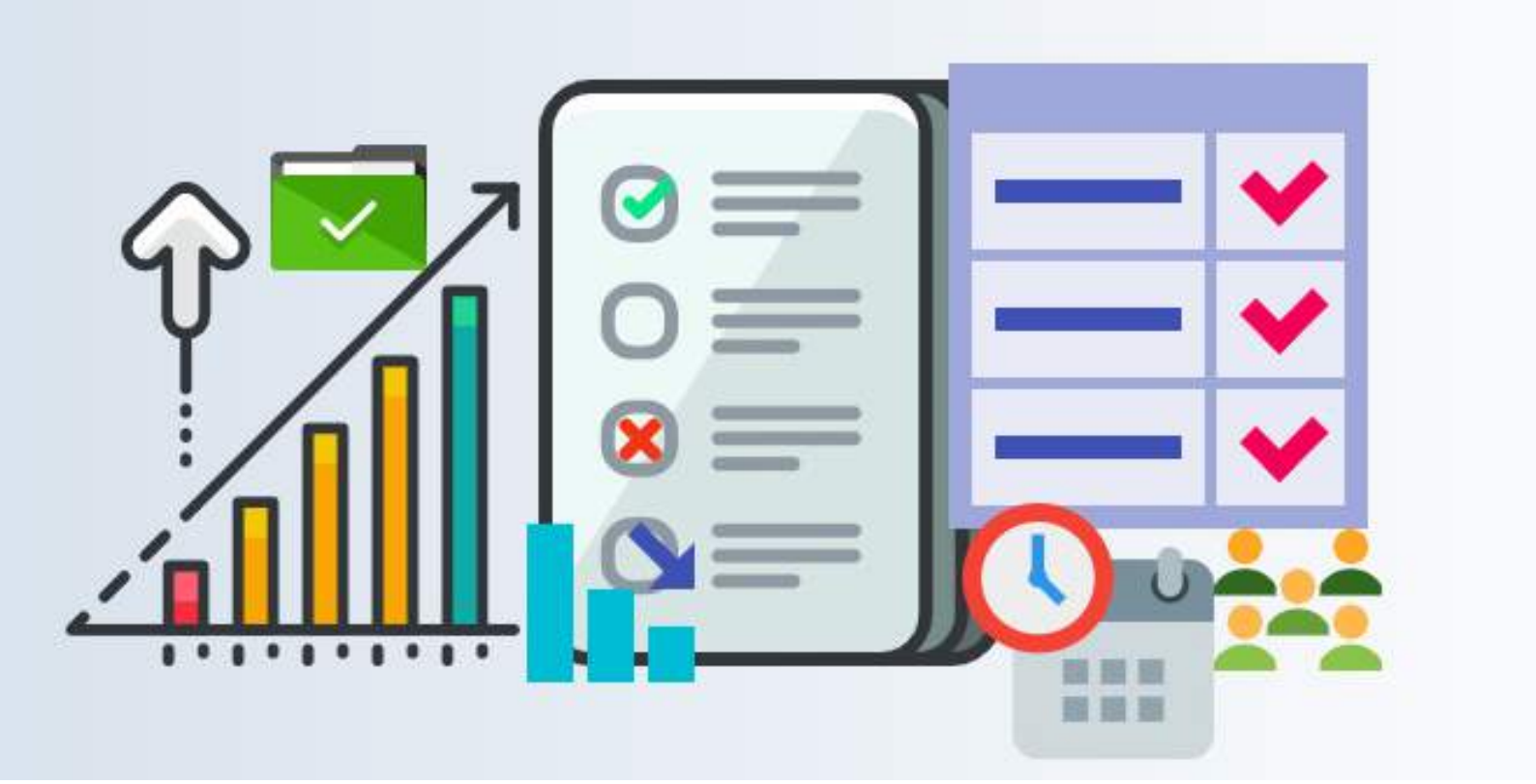
Tới lúc này thì mã triển khai đã có thể tính điểm chính xác trên tất cả các roll có thể xảy ra.

Ca kiểm thử thứ năm

Ca kiểm thử này nhằm kiểm thử trường hợp đặc biệt nhất, khi mà người chơi ăn strike trên tất cả các roll.

```
@Test
void testPerfectGame() {
    rollMany(10, 12);
    assertEquals(300, game.score());
}
```

Ca kiểm thử này được vượt qua mà không cần thêm bất kỳ nỗ lực nào. Đây cũng là kiểm thử cuối cùng của bài kata Bowling Game.



TẠI SAO TEST COVERAGE LÀ MỘT PHẦN QUAN TRỌNG CỦA KIỂM THỬ PHẦN MỀM?

Phan Văn Luân

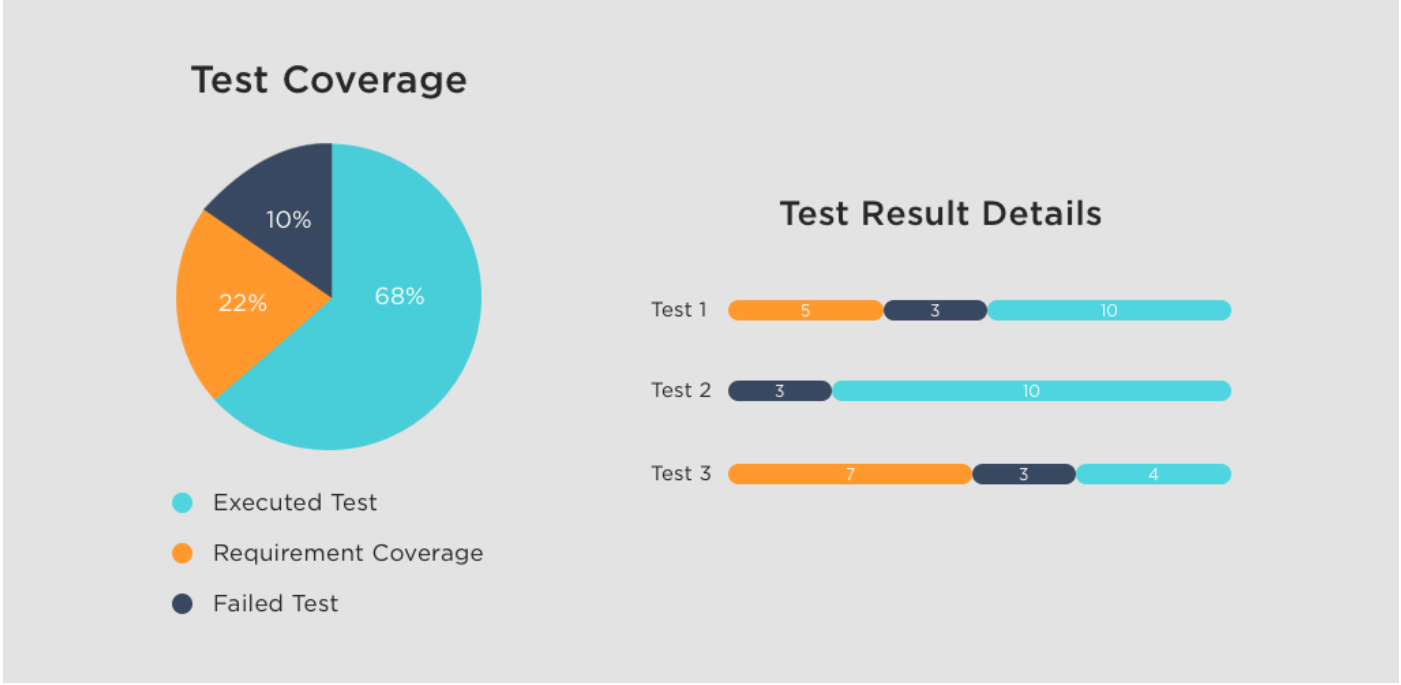
Test coverage là một chỉ số quan trọng trong kiểm thử phần mềm về chất lượng và hiệu quả. Bài viết này chúng ta sẽ tìm hiểu khái niệm test coverage, kỹ thuật, số liệu và cách cải thiện nó.

Thế giới đã chứng kiến một số sự kiện thảm khốc do các lỗi phổ biến trong phần mềm. Một sự kiện như vậy, mà cá nhân tôi nhớ lại, là việc khai trương Heathrow Terminal 5, Vương quốc Anh vào năm 2008.

Các kỹ sư đã khá tự tin về hoạt động của hệ thống xử lý hành lý của khách hàng do đã trải qua quá trình kiểm thử nghiêm ngặt . Tuy vậy hệ thống xử lý hành lý không thể đối phó khi đối mặt với một số tình huống thực tế; dẫn đến việc tắt hoàn toàn hệ thống. Trong 10 ngày sau đó, khoảng 42.000 hành lý không thể đi cùng chủ sở hữu và hơn 500 chuyến bay đã bị hủy.

Tất cả điều này là do các kỹ sư không thực hiện được phạm vi thử nghiệm của các tình huống có thể xảy ra trong thực tế.

Trong bài viết này, chúng ta sẽ thảo luận về tất cả các khía cạnh của test coverage - phạm vi kiểm thử, và cách nó ảnh hưởng trực tiếp đến việc sản xuất, cho dù đó là phát triển phần mềm tùy chỉnh hoặc kiểm thử phần mềm.



Test Coverage là gì?

Test coverage được định nghĩa là một kỹ thuật xác định xem các trường hợp thử nghiệm có thực sự bao trùm mã ứng dụng hay không và bao nhiêu mã được thực hiện khi chạy các trường hợp thử nghiệm đó.

Nếu có 10 yêu cầu và 100 thử nghiệm được tạo và nếu 90 thử nghiệm được thực hiện thì phạm vi thử nghiệm là 90%. Bây giờ, dựa trên số liệu này, người kiểm tra có thể tạo các trường hợp kiểm tra bổ sung cho các kiểm tra còn lại. Dưới đây là một số lợi thế của test coverage.

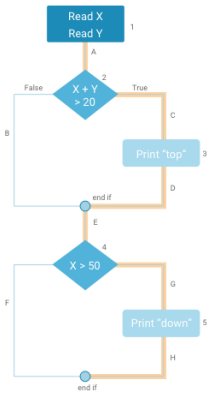
- Bạn có thể xác định các lỗ hổng trong yêu cầu, trường hợp kiểm tra và lỗi ở cấp độ sớm và cấp mã.
- Bạn có thể ngăn ngừa rò rỉ lỗi không mong muốn bằng cách sử dụng phân tích test coverage.
- Test coverage cũng giúp kiểm tra hồi quy, ưu tiên trường hợp kiểm thử, tăng cường bộ kiểm thử và tối thiểu hóa bộ kiểm thử.

Các kỹ thuật Test Coverage

Statement Coverage

Statement Coverage đảm bảo rằng tất cả các dòng lệnh trong mã nguồn đã được kiểm tra ít nhất một lần. Nó cung cấp các chi tiết của cả hai khối mã được thực thi và thất bại trong tổng số các khối mã.

Hãy để hiểu nó với ví dụ về sơ đồ sau. Trong ví dụ đã cho, đường dẫn 1A-2C-3D-E-4G-5H này bao gồm tất cả các câu lệnh và do đó nó chỉ yêu cầu trên một trường hợp thử nghiệm để đáp ứng tất cả các yêu cầu. Một trường hợp thử nghiệm có nghĩa là một Statement Coverage.



Path that covers all the statements in the flowchart:

1A - 2C - 3D - E - 4G - 5H

Trong mã nguồn phức tạp, một đường dẫn không đủ để bao gồm tất cả các câu lệnh. Trong trường hợp đó, bạn cần viết nhiều trường hợp kiểm tra để bao quát tất cả các câu lệnh.

Ưu điểm

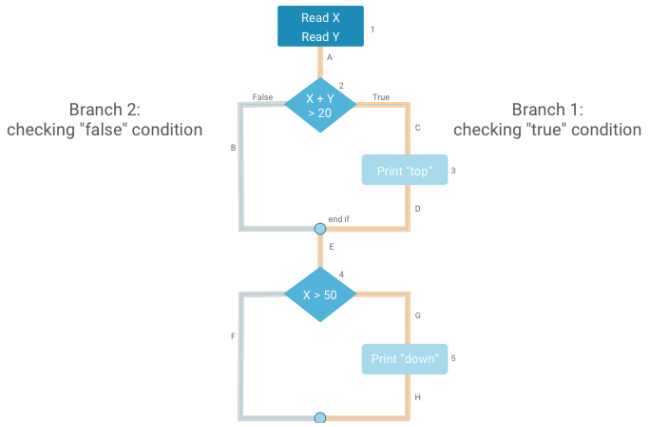
- Nó có thể được áp dụng trực tiếp vào mã đối tượng và không yêu cầu xử lý mã nguồn.
- Nó xác minh những gì mã nguồn viết được dự kiến sẽ thực thi và không thực thi

Nhược điểm

- Nó chỉ bao gồm các điều kiện "true" của mã nguồn.
- Statement Coverage hoàn toàn không quan tâm với các toán tử logic (|| và &&)

Decision/Branch coverage

Các nhà phát triển không thể viết mã trong một chế độ liên tục, tại bất kỳ điểm nào họ cần phân nhánh mã để đáp ứng các yêu cầu chức năng. Sự phân nhánh trong mã thực sự là một bước nhảy từ điểm quyết định này sang điểm khác. Branch coverage kiểm tra mọi đường dẫn có thể hoặc chi nhánh trong mã được kiểm thử.



Branch coverage có thể được tính bằng cách tìm số đường dẫn tối thiểu để đảm bảo rằng tất cả các cạnh đã được che phủ. Trong ví dụ đã cho, không có đường dẫn duy nhất đảm bảo vùng phủ sóng của tất cả các cạnh cùng một lúc.

Ví dụ: nếu bạn đi theo đường dẫn 1A-2C-3D-E-4G-5H này bao gồm số cạnh tối đa (A, C, D, E, G và H), bạn vẫn sẽ bỏ lỡ hai cạnh B và F. Bạn cần đi theo một đường dẫn khác 1A-2B-E-4F để che hai cạnh còn lại. Bằng cách kết hợp hai con đường trên, bạn có thể đảm bảo đi qua tất cả các con đường. Đối với ví dụ này, phạm vi kiểm thử chi nhánh của chúng tôi là 2 vì chúng tôi đang theo hai đường dẫn và nó yêu cầu hai trường hợp thử nghiệm để đáp ứng các yêu cầu.

Ưu điểm

- Nó bao gồm cả các điều kiện đúng và sai không có khả năng được gọi trong statement coverage.
- Nó đảm bảo tất cả các nhánh được kiểm thử.

Nhược điểm

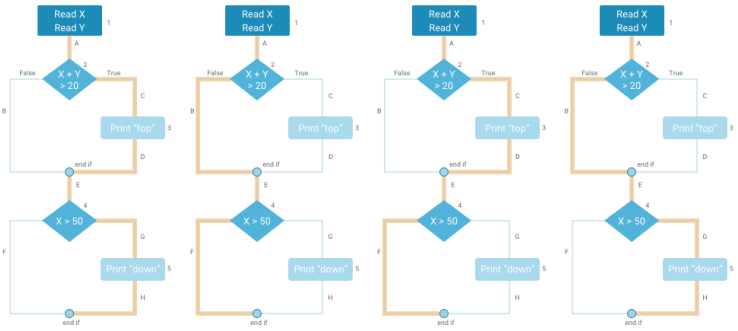
- Nó bỏ qua các nhánh trong các biểu thức Boolean xảy ra do các toán tử ngắn mạch.

Path Coverage

Path Coverage là một phương pháp kiểm tra cấu trúc liên quan đến việc sử dụng mã nguồn của chương trình để tìm mọi đường dẫn thực thi có thể.

Path Coverage đảm bảo phạm vi của tất cả các đường dẫn từ đầu đến cuối. Trong ví dụ này, có bốn đường dẫn có thể kiểm thử:

1. 1A-2B-E-4F
2. 1A-2B-E-4G-5H
3. 1A-2C-3D-E-4G-5H
4. 1A-2C-3D-E-4F



Ưu điểm

- Nó giúp giảm các test thừa.
- Cung cấp phạm vi kiểm tra cao vì nó bao gồm tất cả các câu lệnh và các nhánh trong mã.

Nhược điểm

- Đánh giá mỗi đường dẫn là một thách thức cũng như tốn thời gian vì một số đường dẫn theo cấp số nhân của số nhánh. Ví dụ, một hàm chứa 10 câu lệnh if có 1024 đường dẫn để kiểm tra.
- Đôi khi nhiều đường dẫn không thể thực hiện do mối quan hệ của dữ liệu.

Condition Coverage

Condition Coverage sẽ kiểm tra phạm vi điều kiện nếu cả hai kết quả (có nghĩa là "true" hay "fail") của mọi điều kiện đã được thực hiện. Kết quả của điểm quyết định chỉ liên quan để kiểm tra các điều kiện. Nó đòi hỏi hai trường hợp thử nghiệm cho mỗi điều kiện cho hai kết quả.

Số liệu Test Coverage là gì?

Số liệu test coverage đo lường nỗ lực kiểm thử và giúp trả lời câu hỏi "Bao nhiêu phần của ứng dụng đã được kiểm thử? Số liệu test coverage có thể được chia thành ba phần: số liệu cấp mã, số liệu kiểm tra tính năng và số liệu cấp độ ứng dụng.

Có nhiều công thức khác nhau để tính toán các kết quả này và tạo báo cáo mức độ bao phủ kiểm thử.

Số liệu cấp mã

Tỷ lệ phần trăm testcase được thực hiện

Nó cũng được gọi là các bài kiểm thử được thực hiện được tính bằng tỷ lệ phần trăm của các testcase được thực hiện / thực hiện trong tổng số các testcase.

$$\text{Executed Tests Or Test Execution Coverage Percentage} = \frac{\text{Number of tests run}}{\text{Total number of tests to be run}} \times 100\%$$

Ưu điểm: là bạn có được cái nhìn tổng quan về tiến trình kiểm thử bằng cách đếm số lần testcase đã pass và fail.

Nhược điểm: là việc đếm các testcase pass không liên quan đến chất lượng của các testcase đó.

> ***Ví dụ***
> Một số testcase có thể pass vì nó kiểm tra điều kiện đơn giản hoặc một số lỗi trong mã của testcase đó dẫn đến testcase đó không hoạt động đúng theo yêu cầu.

Số liệu kiểm tra tính năng

Độ bao phủ yêu cầu

Độ bao phủ yêu cầu được sử dụng để xác định các trường hợp kiểm thử bao gồm các yêu cầu phần mềm được xử lý tốt như thế nào. Đối với điều đó, bạn chỉ cần chia số lượng yêu cầu được bao phủ trên tổng số yêu cầu trong phạm vi cho một sprint, phát hành hoặc dự án.

$$\text{Requirements Coverage} = \frac{\text{Number of requirements covered}}{\text{Total number of requirements}} \times 100\%$$

Các trường hợp kiểm thử theo yêu cầu

Số liệu này được sử dụng để xem những tính năng nào đang được kiểm thử và số lượng kiểm thử phù hợp với yêu cầu. Hầu hết các yêu cầu chứa nhiều trường hợp kiểm thử. Điều rất quan trọng là phải biết trường hợp kiểm thử nào bị lỗi đối với một yêu cầu cụ thể để viết lại các trường hợp kiểm thử cho các yêu cầu cụ thể khác.

Request	Test Case	Test Result
Request 1	Test Case 1	Pass
Request 2	Test Case 2	Failed
Request 3	Test Case 3	Incomplete

Số liệu này rất quan trọng đối với các bên liên quan vì nó cho thấy tiến trình phát triển ứng dụng / phần mềm.

Số liệu cấp ứng dụng

Mật độ khiếm khuyết

Mật độ khiếm khuyết là thước đo tổng số khiếm khuyết đã biết chia cho kích thước của thực thể phần mềm được đo.

$$\text{Defect Density} = \frac{\text{Number of known defects}}{\text{Size of software entity}}$$

Nó được sử dụng để xác định các khu vực cần tự động hóa. Nếu mật độ khiếm khuyết cao cho các chức năng cụ thể hơn nó yêu cầu kiểm tra lại. Để giảm các nỗ lực kiểm tra lại, các trường hợp kiểm tra các lỗi đã biết có thể được tự động hóa.

Điều quan trọng là phải xem xét mức độ ưu tiên của khiếm khuyết (thấp hoặc cao) trong khi đánh giá các khiếm khuyết.

> ***Ví dụ***
> Nhiều khiếm khuyết ưu tiên thấp có thể vượt qua vì các tiêu chí chấp nhận đã được thỏa mãn. Mặt khác, chỉ có một khuyết điểm ưu tiên cao có thể ngăn cản các tiêu chí chấp nhận được thỏa mãn.

Các yêu cầu bên ngoài Test Coverage

Sau khi tính toán phạm vi yêu cầu, bạn sẽ tìm thấy một số yêu cầu không được bao phủ. Bây giờ, điều quan trọng là phải biết về từng yêu cầu chưa được đề cập và giai đoạn yêu cầu là gì.

Request ID	Request Name	Request Status
Request 001	Request A	To Do
Request 002	Request B	Done

Số liệu này giúp kiểm tra các kỹ sư và nhà phát triển để xác định và loại bỏ các yêu cầu chưa được khám phá khỏi tổng số yêu cầu trước khi họ gửi chúng đến giai đoạn sản xuất.

Làm thế nào để cải thiện Test Coverage?

Xóa mã "chết"

Test coverage có thể hiểu là tỷ lệ số dòng mã được bao phủ trên tổng số mã trong ứng dụng (cover_code / total_code). Bạn có thể tăng phạm vi test coverage bằng cách giảm mẫu số là tổng mã. Điều này có thể được thực hiện bằng cách xóa mã chết hoặc những đoạn mã thừa. Thông thường, mã "chết" có thể được tìm thấy trong lịch sử phát triển chương trình khi các tính năng đã được thay đổi. Bằng cách này, bạn có thể tăng tổng tỷ lệ bao phủ mã của mình mà không cần viết bất kỳ testcase bổ sung nào.

Mã "chết" có thể được tìm thấy dễ dàng bằng cách kiểm tra thủ công hoặc sử dụng các công cụ tự động hóa. Trước khi loại bỏ mã "chết", bạn cần thực hiện kiểm tra chức năng và đảm bảo nó thực hiện chính xác theo yêu cầu. Bạn cũng có thể sử dụng các công cụ phân tích để xác định mã "chết" không sử dụng trong mã nguồn.

Xoá các đoạn mã trùng lặp

Xóa mã trùng lặp có thể cải thiện tỷ lệ test coverage theo cách tương tự như xóa mã "chết".

Kết luận

Các nhà phát triển ngày nay có hệ thống hơn và các tổ chức tìm kiếm các biện pháp kiểm tra tính đầy đủ và hiệu quả để hiển thị các tiêu chí hoàn thành kiểm thử. Trong đó, test coverage được coi là đặc biệt có giá trị. Dựa vào tỉ lệ test coverage giúp chúng ta giảm thiểu rủi ro tối đa trong phát triển phần mềm.

Nguồn: https://www.simform.com/test-coverage/

THỰC HÀNH BDD VỚI CUCUMBER

Đặng Huy Hòa

Behavior-driven development (BDD) là một quy trình phát triển phần mềm Agile. Quy trình này khuyến khích cộng tác giữa các vai trò kỹ-thuật (như QA, lập trình viên,...) với những vai trò phi-kỹ thuật (như chuyên gia lĩnh vực, người dùng,...) để chia sẻ một cách hiểu chung về những tính năng cần làm trong dự án.

Để áp dụng BDD vào nhóm dự án, bên cạnh việc thực hành quy trình nhuần nhuyễn giữa các vai thì nghiên cứu sử dụng công cụ hỗ trợ cũng là một việc làm không thể thiếu. Trong phạm vi bài viết này, chúng ta sẽ tìm hiểu về Cucumber - một trong số những công cụ như thế.

Bài viết sẽ có những nội dung chính sau:

- Viết kịch bản với cú pháp Gherkin
- Định nghĩa thao tác thực thi với ngôn ngữ JavaScript
- Tự động hoá kiểm thử giao diện với Selenium WebDriver

Tổng quan

Cucumber là một framework hỗ trợ xây dựng các đặc tả với ngôn ngữ tự nhiên và hướng nghiệp vụ, giúp các thành viên có vai trò như QA, BA, Tester, chuyên gia ngành,... có thể dễ dàng viết và cộng tác cùng các thành viên có vai trò xây dựng hệ thống.

Cucumber hỗ trợ nhiều nền tảng công nghệ khác nhau như Java, JavaScript, C#, Ruby, C++,... Vì vậy, nó có thể tích hợp được vào hầu hết các dự án hiện đại.

Việc triển khai các kịch bản kiểm thử với Cucumber được thực hiện gồm 2 phần:

1. QA (hoặc chuyên gia ngành) định nghĩa các kịch bản kiểm thử bằng ngôn ngữ tự nhiên với cú pháp Gherkin. Từ đó các kịch bản kiểm thử được thể hiện thông qua một tập hợp tuần tự các mệnh đề (sẽ được mô tả chi tiết trong mục Cú pháp Gherkin).
2. Cách mệnh đề trong kịch bản kiểm thử sau khi được định nghĩa bằng Gherkin sẽ được cài đặt bằng các đoạn mã bằng ngôn ngữ lập trình tương ứng phù hợp với nền tảng của ứng dụng (xem mục Định nghĩa thao tác).

Cú pháp Gherkin

Cú pháp Gherkin bao gồm những thành phần sau:

- Feature
- Scenario
- Step

Feature (Tính năng)

Thành phần cơ bản trong một dự án là đặc tả tính năng. Các đặc tả này được viết vào một tập tin có đuôi .feature với cú pháp Gherkin.

Dòng đầu tiên của file sẽ bắt đầu với từ khoá `Feature:` và theo sau bởi dòng lùi vào.

> ***Ví dụ:***
>
> ```
> Feature: Là người bận rộn, tôi muốn thêm việc cần làm vào danh sách công việc để lập kế hoạch hằng ngày
> ```

Mỗi feature sẽ bao gồm danh sách các kịch bản kiểm thử liên quan. Các kịch bản này được gọi là `scenario.`

Scenario (Kịch bản)

Kịch bản là nội dung nòng cốt trong cú pháp Gherkin. Đây là nơi đặc tả các thao tác kiểm thử cho một ca kiểm thử.

Kịch bản sẽ được viết bắt đầu với từ khoá `Scenario:` , theo sau là một nội dung tuỳ ý - có thể mô tả rõ ràng nội dung muốn kiểm thử.

> ***Ví dụ:***
>
> ```
> Scenario: Thêm một việc cần làm vào danh sách
> ```

Mỗi kịch bản sẽ có nhiều mệnh đề (chúng ta gọi là `Step`) cùng với các dữ liệu cụ thể thể (được gọi là `Example`). Các mệnh đề và dữ liệu này thường được thu thập từ các yêu cầu sử dụng và mô tả có thể mô tả lại bằng ngôn ngữ tự nhiên.

Step (Mệnh đề)

Mệnh đề những câu mô tả rõ các thao tác thực thi. Các mệnh đề được viết theo sau các từ khoá `Given, When, Then`.

- `Given` được sử dụng để mô tả trạng thái ban đầu của hệ thống (hoặc chức năng). Mục đích của Given là để cung cấp các điều kiện cần thiết trước khi thực hiện các tương tác chính vào hệ thống.
- `When` là từ khoá mô tả sự kiện, hành vi, hoặc tương tác chính của tác nhân lên hệ thống.
- `Then` được sử dụng để mô tả kết quả mong muốn cuối cùng.
- Bên cạnh đó, chúng ta có thể sử dụng `And, But`, hoặc `*` để kết nối mệnh đề có liên quan.

> ***Ví dụ:***
>
> ```
> Scenario: Thêm một việc cần làm vào danh sách
> Given Danh sách việc cần làm ban đầu rỗng
> When Thêm việc ‘Viết bài hướng dẫn Cucumber’ vào danh sách
> Then Danh sách việc cần làm sẽ có ‘Viết bài hướng dẫn Cucumber’
> ```

Các mệnh đề sẽ được định nghĩa với các các mã lệnh kiểm thử dựa trên một ngôn ngữ lập trình cụ thể. Vì trên thực thế, Cucumber không tự biết làm thế nào để thực thi các tính năng, kịch bản để mô tả.

Khi Cucumber thực hiện các mệnh đề trong kịch bản, nó sẽ tìm các Định nghĩa thao tác (Step definitions) phù hợp để thực thi.

Background (Bối cảnh)

Background là từ khoá cho phép thêm ngữ cảnh vào các kịch bản trong một tính năng. Nội dung bối cảnh sẽ được viết giống kịch bản, có thể có các mệnh đề nhưng không có tiêu đề.

Về thứ tực thực hiện, một bối cảnh sẽ được thực hiện trước mỗi kịch bản (và lặp lại nếu có nhiều kịch bản trong một tính năng).

> ***Ví dụ:***
>
> Một số tính năng yêu cầu thực hiện đăng nhập vào hệ thống thì mới có thể thực hiện. Chúng ta sẽ đưa thao tác đăng nhập này vào mục bối cảnh như sau:
>
> ```
> Feature: ...
> Background:
>   Given Truy cập trang quản lý của hệ thống
>   And Đăng nhập vào hệ thống
> ```

Ví dụ hoàn chỉnh

```
Feature: Là người bận rộn, tôi muốn thêm việc cần làm vào danh sách công việc để lập kế hoạch hằng ngày
  Background:
    Given Mở ứng dụng Todos List
    Scenario: Thêm một việc cần làm vào danh sách
    Given Danh sách việc cần làm ban đầu rỗng
    When Thêm việc ‘Viết bài hướng dẫn Cucumber’ vào danh sách
   Then Danh sách việc cần làm sẽ có ‘Viết bài hướng dẫn Cucumber’
```

Chúng ta đã viết đặc tả một tính năng cơ bản trong ứng dụng Todos List. Đây là ứng dụng quản lý danh sách công việc cần làm mỗi ngày. Người dùng có thể thêm, sửa, xoá, cập nhật hoàn thành các công việc hằng ngày. Chúng ta sẽ sử dụng các tính năng này trong các phần tiếp theo của bài viết:

- Link để truy cập ứng dụng là https://cg-to-do-list-demo.netlify.app/
- Mã nguồn ứng dụng được tham khảo và sao chép từ bài viết https://freshman.tech/to-do-list/

Định nghĩa thao tác (JavaScript)

Chúng ta có thể bắt đầu sử dụng Cucumber với JavaScript qua hệ sinh thái Nodejs.

Bước 1 - Cài đặt

Tạo dự án Nodejs mới và cài đặt các gói cần thiết:

```
mkdir automation_testing_project
cd automation_testing_project
npm init
npm install cucumber chai selenium-webdriver
chromedriver --save
```

Mở file package.json và copy mã dưới đây để thực thi kịch bản trong Cucumber:

```
...
“scripts”: {
    “test”: “./node_modules/.bin/cucumber-js
features/*.feature -r step_definitions -r
support”
},
...
```

Tiếp tục, chạy lệnh sau để kiểm tra kết quả:

```
npm test
```

Nếu kết quả hiển thị như sau là cài đặt thành công:

```
0 scenarios
0 steps
0m00.000s
```

Bước 2 - Viết kịch bản với Gherkin

Tạo thư mục features trong dự án. Sau đó tạo file add_todo.feature với nội dung như sau:

```
Feature: Là người bận rộn, tôi muốn thêm việc
cần làm vào danh sách công việc để lập kế
hoạch hằng ngày
  Background:
    Given Mở ứng dụng Todos List
    Scenario: Thêm một việc cần làm vào danh
    sách
    Given Danh sách việc cần làm ban đầu rỗng
    When Thêm việc ‘Viết bài hướng dẫn Cucum-
    ber’ vào danh sách
 Then Danh sách việc cần làm sẽ có ‘Viết bài
ướng dẫn Cucumber’
```

Mở Terminal và thực thi lệnh npm test, kết quả trên terminal sẽ trả về như sau:

```
Warnings:
1) Scenario: Thêm một việc cần làm vào danh
sách # features/add_todo.feature:6
   ? Given Mở ứng dụng Todos List
     Undefined. Implement with the following
     snippet:
       Given(‘Mở ứng dụng Todos List’, func-
       tion () {
         // Write code here that turns the
         phrase above into concrete actions
         return ‘pending’;
       });
#....
#.... kết quả khá dài nên mình đã ẩn bớt đi
#....
1 scenario (1 undefined)
4 steps (4 undefined)
0m00.000s
npm ERR! Test failed.  See above for more
details.
```

Kết quả việc thực hiện kịch bản trên là failed vì chúng ta chưa định nghĩa thao tác để tương tác với ứng dụng.

Bước 3 - Định nghĩa thao tác

Ở bước này, chúng ta sẽ cùng nhau cài đặt các thao tác mô phỏng hành vi người dùng trên ứng dụng web như: gõ phím, click vào button, kiểm tra text,... qua một framework có tên là Selenium WebDriver.

Cấu hình Selenium WebDriver

Tạo thư mục support và file world.js bên trong với nội dung như sau:

```
const seleniumWebdriver = require
(‘selenium-webdriver’);
const { setWorldConstructor } = require
(‘cucumber’);
class CustomWorld {
 constructor() {
   this.driver = new seleniumWebdriver.
   Builder()
   .forBrowser(‘chrome’)
   .build()
   this.waitForElement = function(locator) {
      const condition = seleniumWebdriver.
      until.elementLocated(locator)
      return this.driver.wait(condition)
   }
 }
}
setWorldConstructor(CustomWorld);
```

Bổ sung định nghĩa

Tạo thư mục step_definitions tại thư mục gốc của dự án. Tạo file add_todo.js với nội dung lần lượt theo kịch bản ở file add_todo.feature.

Trước tiên, chúng ta phải import các thư viện cần thiết:

```
const { Given, When, Then, After } =
require('cucumber');
const { Key } = require
('selenium-webdriver');
const { expect } = require('chai');
```

Với mệnh đề Given Mở ứng dụng Todos List, chúng ta định nghĩa như sau để thư viện Selenium WebDriver có thể khởi động trình duyệt và truy cập link ứng dụng:

```
Given('Mở ứng dụng Todos List', async
function () {
  await this.driver.get
('https://cg-todo-list-demo.netlify.app/');
});
```

Mệnh đề Given Danh sách việc cần làm ban đầu rỗng được định nghĩa với các dòng dưới đây. Chúng ta sẽ sử dụng expect thuộc thư viện chai để kiểm tra kết quả thực tế hiển thị trên ứng dụng với mong đợi. Mong muốn của chúng ta là: ở lần đầu tiên truy cập, danh sách công việc phải rỗng (chưa có công việc nào).

```
Given('Danh sách việc cần làm ban đầu
rỗng', async function () {
  const todolist = await this.driver.
  findElement({css: '.todo-list'});
  const items = await todolist.
  findElements({css: 'li'});
  expect(items).to.have.lengthOf(0);
});
```

Dưới đây là mệnh đề When Thêm việc 'Viết bài hướng dẫn Cucumber' vào danh sách. Để thao tác này có thể được tái sử dụng ở kịch bản khác, chúng ta khai báo tham số {string} với tên công việc.

```
When('Thêm việc {string} vào danh sách',
async function (todoText) {
  const todoInput = await this.driver.
  findElement({css: '.js-todo-input'});
  await todoInput.sendKeys(todoText + Key.
  ENTER);
});
```

Cuối cùng, định nghĩa mệnh đề Then Danh sách việc cần làm sẽ có 'Viết bài hướng dẫn Cucumber' như sau:

```
Then('Danh sách việc cần làm sẽ có
{string}', async function (todoText) {
  const todolist = await this.driver.
  findElement({css: '.todo-list'});
  const items = await todolist.
  findElements({css: 'li'});
  const todo = await items[0].getText();
  expect(items).to.have.lengthOf(1);
  expect(todo).to.equal(todoText);
});
```

Sau khi thực hiện sau kịch bản, nếu muốn trình duyệt tự động tắt đi thì chúng ta có thể sử dụng After:

```
After(async function() {
  await this.driver.close();
});
```

Có một lưu ý nhỏ khi cài đặt các định nghĩa thao tác trên JavaScript. Các thao tác của chúng ta được Selenium WebDriver thực hiện bất đồng bộ và kết quả trả về là Promise. Vì vậy, để mã nguồn dễ đọc, chúng ta nên sử dụng async/await như các ví dụ ở trên.

Bạn có thể tham khảo lại mã nguồn tại đây: https://github.com/hoadh/demo-todo-list-cucumber-js

Kinh nghiệm triển khai

Kịch bản tránh phụ thuộc UI

Ví dụ không tốt:

```
# BAD EXAMPLE! Do not copy.
Feature: Google Searching
  Scenario: Google Image search shows
  pictures
     Given the user opens a web browser
     And the user navigates to
     "https://www.google.com/"
     When the user enters "panda" into the
     search bar
     Then links related to "panda" are shown
     on the results page
     When the user clicks on the "Images"
     link at the top of the results page
     Then images related to "panda" are
     shown on the results page
# Nguồn: https://automationpanda.
com/2017/01/30/bdd-101-writing-good-gher-
kin/
```

Với tình huống trên, khi lập trình viên thay đổi mã giao diện ("Images" link) thì đặc tả kịch bản phải cập nhật lại. Vì vậy, để viết kịch bản thì chúng ta nên sử dụng ngôn ngữ hướng người dùng, đặc tả ở mức cao (high-level), tránh quá chi tiết và phụ thuộc vào thành phần giao diện như ví dụ trên.

Áp dụng Page Object Model

Page Object Model (POM) là một design pattern phổ biến trong Automation testing. Việc sử dụng POM đảm bảo tính tổ chức và tái sử dụng mã kiểm thử, nhờ đó mã nguồn dễ mở rộng và dễ bảo trì hơn.

Thay vì truy cập trực tiếp các DOM thông qua WebDriver như đã làm ở các ví dụ trên, chúng ta tạo các class bao đóng thành phần giao diện bên trong. Vì thế:

- Thay đổi trong giao diện không ảnh hưởng quá nhiều đến định nghĩa thao tác
- Có thể tái sử dụng giao diện khi có kịch bản mới
- Các trang phức tạp có thể được cấu trúc thành nhiều page object nhỏ hơn

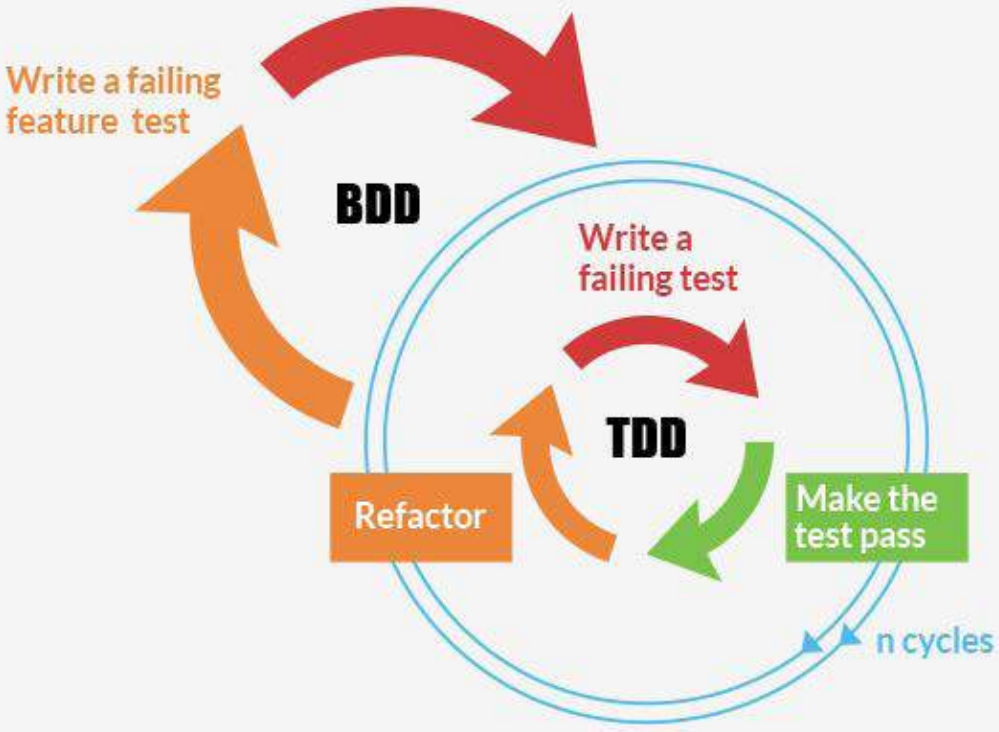
Bạn có thể tham khảo mã nguồn được refactor theo POM tại đây: https://github.com/hoadh/demo-todo-list-cucumber-js/tree/refactor

Chặng đường tiếp theo

Cảm ơn bạn đã đồng hành cùng bài viết đến đây. Hy vọng nội dung bài viết này có thể giúp bạn có được cái nhìn tổng quan về áp dụng BDD với Cucumber.

Tiếp theo, để tăng thêm hiểu biết về BDD và Cucumber, tác giả bài viết gợi ý một số tài nguyên dưới đây:

- Cucumber School: https://cucumber.io/school/
- Tài liệu Cucumber framework https://cucumber.io/docs/guides/
- Cú pháp Gherkin https://cucumber.io/docs/gherkin/
- Thư viện Chaijs https://www.chaijs.com/
- Tài liệu sử dụng Selenium WebDriver: https://www.selenium.dev/documentation/en/



UNIT TESTING TRONG ANGULAR

Đặng Huy Hòa

Nếu bạn chưa biết Unit Testing và những khái niệm cơ bản liên quan, hãy tìm đọc bài viết Unit Test - Những bước chân đầu tiên trong Tạp Chí Lập Trình Vol.4.

Kiểm thử giúp đảm bảo những thay đổi hoặc bổ sung không gây ảnh hưởng đến các tính năng cũ của ứng dụng hoặc phát sinh lỗi mới. Chúng ta tiếp tục tìm hiểu cách viết unit test cho các ứng dụng được xây dựng với framework Angular. Cụ thể, qua bài viết này, chúng ta sẽ đạt được các mục tiêu sau:

- Hiểu biết cơ bản về testing trong Angular
- Có thể viết unit test cho component
- Có thể viết unit test cho service
- Có thể giả lập service phụ thuộc trong các component

Để đọc hiểu bài viết này, bạn cần những kiến thức cơ bản về lập trình với ngôn ngữ JavaScript và sử dụng được framework Angular.

Tổng quan

Angular hỗ trợ hai loại kiểm thử: Unit testing (kiểm thử đơn vị) và End-to-end testing (e2e). Với kiểm thử đơn vị, Jasmine là framework được sử dụng mặc định. Karma là thành phần thực thi các bộ test. Sau khi thực thi các bộ test, Karma sẽ tạo file báo cáo kết quả với định dạng HTML. Với kiểm thử e2e, nhóm phát triển Angular gợi ý sử dụng Protractor. Đây là thư viện được xây dựng dựa trên Selenium WebDriver.

Để thực thi kiểm thử đơn vị, chúng ta sử dụng lệnh `ng serve`. Để thực thi kiểm thử e2e, sử dụng lệnh `ng e2e`.

Tìm hiểu Jasmine

Jasmine là framework kiểm thử cho các ứng dụng được xây dựng trên JavaScript. Với cú pháp đơn giản, rõ ràng, đây là một trong những framework mạnh mẽ để viết các unit test cho mã JavaScript. Nó có thể chạy độc lập, không phụ thuộc vào các thư viện khác và không yêu cầu DOM.

Ví dụ:

```
describe('TestSuiteName', () => {
  // suite of tests here
  it('should do some stuff', () => {
    // this is the body of the test
  });
});
```

describe

Hàm describe được sử dụng để nhóm các đặc tả hoặc kiểm thử có liên quan. Đầu vào là 2 tham số:

- một chuỗi mô tả mục đích của nhóm kiểm thử
- một hàm callback chứa các đặc tả, hoặc các bộ kiểm thử

Ví dụ:

```
describe("TestSuiteName", function() {
  ...
});
```

Các khối describe có thể lồng nhau.

```
describe("A suite", function() {
  ...
  describe("Another suite inside",
function() {
  ...
 });
 ...
});
```

Để tạm thời dừng thực thi một khối `describe` nào đó, chúng ta có thể thay thế hàm thành `xdescribe`. Để chỉ thực thi một khối `describe` nào đó, chúng ta sử dụng `fdescribe`.

It

Đây là hàm chứa các đặc tả hoặc kiểm thử cụ thể.

```
it("contains spec with an expectation",
function() {
  expect(myOrg).toBe('CodeGym');
});
```

Để tạm thời dừng một test case nào đó, chúng ta sử dụng hàm `xit` thay thế `it`. Và để chỉ thực thi một test case nào đó, chúng ta sử dụng `fit` thay thế.

Expect

Hàm expect được sử dụng để đánh giá kết quả một đợi và kết quả thực tế của một kiểm thử.

```
expect(myOrg).toBe('CodeGym');
```

Như ví dụ trên, hàm `expect` mong đợi giá trị của biến `my_variable` có bằng `true` hay không. Nếu giá trị thực tế của biến `my_variable` là `true` thì kết quả của test case này là `PASS`; ngược lại, kết quả là `FAIL`.

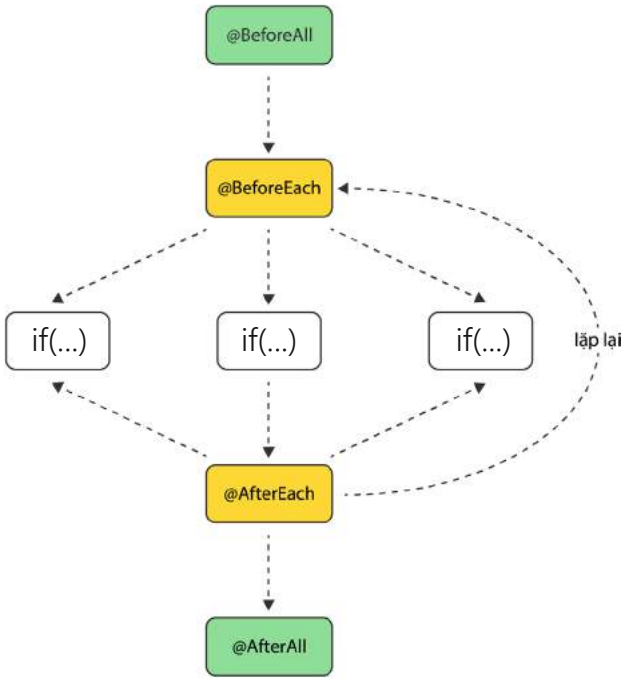
Setup và Teardown

- Setup là thành phần được thực thi trước tất cả (hoặc mỗi) test case.
- Teardown là thành phần được thực thi sau tất cả (hoặc mỗi) test case.

Trong Jasmine, để thiết lập Setup và Teardown, chúng ta sử dụng các hàm sau:

- beforeEach
- beforeAll
- afterEach
- afterAll

Thứ tự thực hiện của các hàm trên trong Jasmine được thể hiện như khối hình dưới đây.



Done

Xét tình huống chúng ta muốn kiểm tra một hàm xử lý bất đồng bộ (async function). Việc thực thi hàm này có thể mất thời gian chờ đợi. Hàm `done()` được sử dụng trong trường hợp này để báo Jasmine biết thời điểm kết thúc việc thực thi.

Ví dụ

```
it('should wait 3 seconds', (done) => {
  const weAre = 'CodeGym';
  setTimeout( () => {
    expect(weAre).toBe('CodeGym');
    done();
  }, 3000);
});
```

Chúng ta muốn jasmine đợi 5 giây trước khi đánh giá kết quả của biến `weAre`.

Với một hàm bất đồng bộ trả về kiểu Promise, chúng ta có thể viết kiểm thử như sau:

```
it('should wait until getting result',
(done) => {
    doSomething()
    .then( result => {
        expect(result).toBe('CodeGym');
        done();
    })
    .catch(error => {
      fail();
      done();
  });
});
```

Testing trong Angular

Tiện ích TestBed

`TestBed` là một tiện ích được Angular cung cấp để tạo môi trường kiểm thử phù hợp cho các thành phần của Angular như: Component, Service, Pipe, Directive,...

Với TestBed, lập trình viên có thể khởi tạo một module kiểm thử với phương thức `configureTestingModule`. Tham số cung cấp cho `configureTestingModule` khi khởi tạo module là những metadata cần thiết cho một module như imports, providers, declarations,...

Chúng ta thường tạo mới một module kiểm thử trước khi thực hiện các test case liên quan tới component trong hàm beforeEach (như đã giới thiệu ở trên).

Ví dụ:

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    imports: [
      RouterTestingModule
    ],
    declarations: [
      AppComponent
    ],
  }).compileComponents();
}));
```

Nếu bộ kiểm thử đang thực thi phụ thuộc vào một service nào đó, chúng ta khai báo tên service vào metadata providersnhư sau:

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    ...
    providers: [OneService]
  }).compileComponents();
}));
```

Component Fixtures

Fixture là đối tượng đại diện component root trong Angular. Với fixture, chúng ta có thể sử dụng debugElement để truy cập các thuộc tính bên trong component.

Component fixture được tạo bằng phương thức createComponent của TestBed.

Hãy xem qua ví dụ dưới đây. Để lấy được giá trị text nằm trong thẻ <h1> đầu tiên trên template, chúng ta viết đoạn mã như sau:

```
fixture = TestBed.createComponent
(MyComponent); // (1)
debugElement = fixture.debugElement; // (2)
let el = debugElement.query(By.css('h1'));
// (3)
let value = el.nativeElement.innerHTML; //
(4)
```

Giải thích các dòng mã trên:

1. Tạo một component fixture
2. Truy cập đối tượng debugElement từ fixture
3. Truy cập thẻ <h1> trên template bằng phương thức query. Chúng ta có thể kết hợp By.css để truy cập các phần tử với phương pháp tương tự selector của css.
4. Biến value chứa giá trị text trong thẻ <h1> trên template.

Tình huống viết test trong Angular

Unit test cho component

Bước 1: Tạo mới một component có tên là CodeGym từ Angular/CLI với lệnh sau:

```
ng g c codegym
```

Cấu trúc thư mục của component CodeGym được tạo như sau:

```
src/
-- app/
-- -- codegym/
-- -- -- codegym.component.css
-- -- -- codegym.component.html
-- -- -- codegym.component.spec.ts
-- -- -- codegym.component.ts
```

File codegym.component.spec.ts là nơi chứa mã unit test của component CodeGym. Chúng ta sẽ bổ sung mã test case vào đây sau khi bổ sung mã cho template và component.

Bước 2: Sửa nội dung file componentcodegym.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-codegym',
  templateUrl: './codegym.component.html',
  styleUrls: ['./codegym.component.css']
})
export class CodegymComponent {
  myOrg = 'CodeGym';
  changeMyText() {
    this.myOrg = 'CodeGym MonCity';
  }
}
```

Sửa nội dung file template codegym.component.html:

```
<p>{{myOrg}}</p>
<button (click)="changeMyText()">Change
   Text</button>
```

Ở component này, khi người dùng click vào button Change Text thì giá trị của myOrg thay đổi, và chuỗi trong thẻtrên template sẽ được cập nhật.

Bước 3: Bổ sung test case

Chúng ta muốn kiểm tra chuỗi được cập nhật sau khi click button có như mong muốn. Test case được bổ sung vào file codegym.component.spec.ts như sau:

```
it('should change text after clicking on
`Change Text` button', () => {
  // Arrange (1)
  const buttonElement =
  debugElement.query(By.css('button'));
  const pElement = debugElement.query(By.
  css('p'));
  const expected = 'CodeGym MonCity';
  // Act (2)
  buttonElement.triggerEventHandler('click',
  null);
  fixture.detectChanges();
  // Assert (3)
  const actual = pElement.nativeElement.
  innerText;
  expect(actual).toEqual(expected);
});
```

Giải thích mã test case trên:

1. Arrange (1) là khu vực chứa mã chuẩn bị cho dự án. Bao gồm: ánh xạ đến hai thẻ và trên template thông qua hàm query thuộc DebugElement, kết hợp By.css.
2. Act (2) là nơi chứa mã thực hiện thao tác click vào button. Với phương thức triggerEventHandler, chúng ta có thể kích hoạt một sự kiện trên template. Trong test case, Angular không tự động phát hiện các thay đổi. Vì vậy cần gọi hàm detectChanges từ fixture để yêu cầu Angular chờ đến khi template được cập nhật.
3. Assert(3) làm hai nhiệm vụ sau:

- Truy cập nội dung cập nhật trên template nhờ sử dụng thuộc tính .nativeElement.innerText, giá giá trị vào biến actual.
- So sánh với giá trị mong đợi (biến expected) thông qua hàm expect.

Unit test cho service

Giả sử, ứng dụng của chúng ta cần service phục vụ xử lý văn bản.

Bước 1: Tạo mới service

```
ng g s text-transform
```

Cấu trúc thư mục của service text-transform được tạo như sau:

```
src/
-- app/
-- -- text-transform.service.spec.ts
-- -- text-transform.service.ts
```

File `text-transform.service.spec.ts` là nơi chứa mã unit test của service TextTransform.

Bước 2: Bổ sung mã vào text-transform.service.ts

```
removeSpaces(text: string) {
    return text.replace(/\s/g, '');
}
```

Phương thức removeSpaces này có nhiệm vụ xoá tất cả khoảng trắng trong chuỗi đầu vào.

Bước 3: Bổ sung mã test case vào text-transform.service.spec.ts

```
it('should remove all space characters', () => {
  // Arrange
  const service: TextTransformService = TestBed.get(TextTransformService); // (1)
  const text = 'Code Gym Moncity';
  const expected = 'CodeGymMoncity';
  // Act
  const actual = service.removeSpaces(text); // (2)
  // Assert
  expect(actual).toEqual(expected); // (3)
});
```

Ở đoạn mã trên, chúng ta sử dụng TestBed.get để lấy đối tượng service được tạo ra từ testing module. (1)

Sau khi thực thi phương thức removeSpaces (2), chúng ta kiểm tra kết quả trả về với giá trị mong đợi qua dòng lệnh expect(actual).toEqual(expected); (3)

Unit test cho component có service phụ thuộc

Với hai ví dụ unit test cho component và service ở trên, chúng ta đã có component và service. Nếu component muốn sử dụng service TextTransform để để chuyển xoá khoảng trắng trước khi hiển thị thì chúng ta phải tiêm (inject) service vào component qua constructor như sau:

```
...
export class CodegymComponent {
  ..
   constructor(private textTransform: TextTransformService) {}
  ...
}
```

Ở phía test case, chúng ta bổ sung metadata providers cho phương thức createTestingModule. Đây là nơi khai báo các phụ thuộc cho module testing.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ CodegymComponent ],
    providers: [TextTransformService]
  })
  .compileComponents();
}));
```

Unit test cho service sử dụng HttpClient

Với tình huống cần test các service có sử dụng giao thức HTTP để giao tiếp với API Backend, Angular cung cấp HttpTestingClientModule và công cụ giả lập HTTP bằng HttpTestingController.

Ví dụ, chúng ta xây dựng tính năng hiển thị các thông tin Github của account codegym-vn. Như vậy sẽ cần service gọi API của Github để lấy các thông tin này.

Bước 1: Tạo mới service

```
ng g s github-api
```

Cấu trúc thư mục của service text-transform được tạo như sau:

```
src/
-- app/
-- -- github-api.service.spec.ts
-- -- github-api.service.ts
```

File github-api.service.spec.ts là nơi chứa mã unit test của service GithubApi.

Bước 2: Bổ sung mã vào app.module.ts và github-api.service.ts

Vì chúng ta cần tạo các request HTTP, nên ứng dụng Angular được import module HttpClientModule:

```
...
import {HttpClientModule} from '@angular/common/http';
@NgModule({
  ...
  imports: [
    ...
    HttpClientModule
  ],
  ...
})
export class AppModule { }
```

Tiếp tục bổ sung mã vào file github-api.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/
http';
@Injectable({
  providedIn: 'root'
})
export class GithubApiService {
  apiUrl = 'https://api.github.com/users/co-
degym-vn';
    constructor(private httpClient: HttpCli-
ent) { }
  fetchUser() {
    return this.httpClient.get(this.apiUrl);
  }
}
```

Phương thức fetchUser có nhiệm vụ lấy thông tin của codegym-vn từ API của Github qua đường link https://api.github.com/users/codegym-vn.

Bước 3: Bổ sung mã test case vào github-api.service.spec.ts

```
import { TestBed } from '@angular/core/
testing';
import { GithubApiService }
from './github-api.service';
import { HttpClientTestingModule,
HttpTestingController }
from '@angular/common/http/testing';
fdescribe('GithubApiService', () => {
  let httpMock: HttpTestingController; //
(1)
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule]
    });
    httpMock = TestBed.get(HttpTestingCon-
troller);
  });
  it('should fetch user from Github API',
() => {
    const service: GithubApiService = Test-
Bed.get(GithubApiService);
    // (2)
    service.fetchUser().subscribe( res => {
      expect(res['login']).toEqual('code-
gym-vn'); // (3)
    });
    const req = httpMock.expectOne(service.
apiUrl); // (4)
    expect(req.request.method).toBe('GET');
    httpMock.verify(); // (5)
  });
});
```

Ở đoạn mã trên, chúng ta sử dụng HttpTestingController để xác minh các HTTP request. (1)

Sau khi thực thi phương thức fetchUser() (2), chúng ta kiểm tra kết quả API trả về với giá trị mong đợi qua mã lệnh expect(res['login']).toEqual('codegym-vn'); (3)

Đồng thời, chúng ta có thể kiểm chứng service đã tạo một HTTP request với method 'GET' tới đường link được lưu trong thuộc tính apiUrl . (4)

Cuối cùng, httpMock.verify() (5) giúp xác minh rằng không còn request nào được tạo ra sau khi thực thi các đoạn mã mà chưa kiểm chứng.

Mô phỏng service khi test component

Khi component phụ thuộc vào một service, và service này lại phụ thuộc vào một thành phần bên ngoài như API hoặc dịch vụ bên thứ ba,... chúng ta có thể giả lập các service này bằng cách tạo đối tượng mô phỏng.

Tiếp tục từ ví dụ ở mục Unit test cho service sử dụng HttpClient trên, chúng ta tạo component hiển thị các thông tin lấy từ GitHub API qua GithubApiService.

Bước 1: Tạo mới một component có tên là Repo từ Angular/CLI với lệnh sau:

```
ng g c repo
```

Cấu trúc thư mục của component Repo được tạo như sau:

```
src/
-- app/
-- -- repo/
-- -- -- repo.component.css
-- -- -- repo.component.html
-- -- -- repo.component.spec.ts
-- -- -- repo.component.ts
```

File repo.component.spec.ts là nơi chứa mã unit test của component Repo. Chúng ta sẽ bổ sung mã test case vào đây sau khi bổ sung mã cho template và component.

Bước 2: Sửa nội dung component repo.component.ts

```
...
export class RepoComponent {
  user: any;
     constructor(private  githubApiService: GithubApiService) { }
  fetchGithubUser() {
      this.githubApiService.fetchUser().subscribe( res => {
      this.user = res;
    });
  }
}
```

Sửa nội dung file template repo.component.html:

```
<p *ngIf="user">{{ user.login }}</p>
<button (click)="fetchGithubUser()">Fetch Github User</button>
```

Ở component này, khi người dùng click vào button Fetch Github User thì giá trị user thay đổi, và chuỗi trong thẻ

Trên template sẽ được cập nhật lại.

Bước 3: Bổ sung test case

Để kiểm tra template được cập nhật như mong đợi sau khi click button, file repo.component.spec.ts sẽ được bổ sung như sau:

```
...
fdescribe('RepoComponent', () => {
  let component: RepoComponent;
  let fixture: ComponentFixture<RepoComponent>;
  // (1)
  const mockGithubApiService = {
    fetchUser: () => of({ login: 'codegym-vn' }) // (2)
  };
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ RepoComponent ],
      // (3)
      providers: [{
        provide: GithubApiService,
        useValue: mockGithubApiService
      }],
      imports: [HttpClientTestingModule]
    })
    .compileComponents();
  }));
  ...
  it('should display GitHub repository name after click button', () => {
    // Arrange
    const debugElement = fixture.debugElement;
    const expected = 'codegym-vn';
    // Act
    const buttonElement = debugElement.query(By.css('button'));
    buttonElement.triggerEventHandler('click', null);
    fixture.detectChanges();
    // Assert
    const pElement = debugElement.query(By.css('p'));
    const actual = pElement.nativeElement.innerText;
    expect(actual).toEqual(expected);
  });
});
```

Giải thích mã test case trên:

1. Khởi tạo một đối tượng mô phỏng cho GithubApiService có tên là mockGithubApiService.
2. Component chỉ phụ thuộc vào hàm fetchUser của service thật, nên chúng ta khai báo hàm fetchUser trả về giá trị là một đối tượng cố định. Hàm of (thuộc gói 'rxjs') sẽ trả về một Observable (giống kết quả được trả về từ HttpClient).
3. providers (3) là khu vực khai báo các service phụ thuộc. Chúng ta sử dụng useValue để tiêm vào đối tượng mô phỏng được khởi tạo ở dòng (1).

Chặng đường tiếp theo

Hy vọng những tình huống viết test ở trên sẽ phù hợp với quá trình triển khai dự án Angular trên thực tế của bạn.

Tiếp theo, tác giả gợi ý một số tài nguyên về testing trong Angular/JavaScript dưới đây:

- Mục Testing trên trang tài liệu chính thức của Angular - https://angular.io/guide/testing
- Trang chủ Jasmine - https://jasmine.github.io/
- Các bài hướng dẫn Unit Testing cho Angular trên CodeCraft.tv - https://codecraft.tv/courses/angular/unit-testing/overview/
- Khoá học có trả phí trên PluralSight - https://www.pluralsight.com/courses/unit-testing-angular

NHỮNG ĐIỀU NÊN LƯU Ý
KHI THỰC HÀNH KIỂM THỬ TỰ ĐỘNG

Nguyễn Bình Sơn

Tầm quan trọng của hoạt động Kiểm thử Tự động trong phát triển phần mềm nằm ở khả năng cung cấp nhanh chóng các phản hồi cho nhóm phát triển khi xây dựng tính năng mới.

Nó cũng loại bỏ gánh nặng của thao tác kiểm thử hồi quy, giúp QA có thể tập trung vào các hoạt động kiểm thử khác.

Bài viết này cung cấp các lưu ý cũng như những cạm bẫy cần tránh khi triển khai kiểm thử tự động nhằm giúp mang lại giá trị cao nhất cho nhóm phát triển.

Thủ công và Tự động; Kiểm thử và Rà soát

Tránh đặt bài toán so sánh giữa kiểm thử tự động và thủ công. Cả hai hoạt động đều cần thiết và chúng phục vụ cho những mục đích khác nhau. Kiểm thử tự động là một tập hợp các hướng dẫn được viết bởi con người để thực hiện một phép kiểm thử cụ thể. Mỗi khi một kiểm thử tự động được khởi động, nó làm theo một cách chính xác những gì được mô tả và chỉ kiểm tra chính xác những gì được yêu cầu.

Kiểm thử thủ công thì khác, khi thực hiện, người tester có thể liên kết nhiều dấu hiệu, hay thay đổi các bước kiểm thử theo nhiều cách khác nhau, để phát hiện ra các vấn đề ngoài kịch bản. Điều này thể hiện đặc biệt rõ trong các kiểm thử thăm dò.

Tự động hóa những Kiểm thử Hồi quy

Lý do chính cho việc tự động hóa một bài kiểm thử là để thực hiện những kiểm thử mang tính lặp lại. Nếu bài kiểm thử chỉ được yêu cầu để thực thi một vài lần duy nhất thì nỗ lực cần bỏ ra để tự động hóa có thể lớn hơn nhiều so với lợi ích mang lại.

Kiểm thử hồi quy là những bài kiểm thử được yêu cầu phải lặp lại mỗi khi phát hành phiên bản mới của sản phẩm. Đây là những bài kiểm thử cần thiết, nhưng tốn thời gian, và nhàm chán. Đây là những bài kiểm thử nên được tự động hóa.

Thiết kế bài kiểm thử trước khi tự động hóa chúng

Viết trước các test case với một chiến lược rõ ràng và cụ thể trước khi tiến hành tự động hóa kiểm thử. Như thế thiết kế kiểm thử sẽ tốt hơn và giúp dễ nhận diện các lỗi. Việc "tự động hóa" chỉ đơn thuần là triển khai thiết kế của kiểm thử .

Sẽ rất nguy hiểm nếu nhảy ngay vào tự động hóa bởi bạn sẽ chỉ chú ý tới việc làm cho các mã chỉ dẫn hoạt động mà quên mất toàn cảnh. Bạn cũng sẽ thường chỉ chú ý tới các phép kiểm thử "dương" và "trôi chảy" và cố ý bỏ qua các phép kiểm thử cần thiết khác.

Và, đừng giảm phạm vi của kiểm thử chỉ để kiểm thử xanh hết.

Loại bỏ sự thiếu nhất quán khỏi các kiểm thử tự động

Một trong những điểm cốt yếu của kiểm thử tự động là cung cấp kết quả nhất quán mà từ đó chúng ta có thể khẳng định được có điều gì đó không đúng khi xuất hiện kiểm thử thất bại.

Nếu một kiểm thử tự động mới vừa xanh và rồi đỏ ngay trong lần chạy tiếp theo, mà không có bất kỳ thay đổi nào trên phần mềm, chúng ta không thể khẳng định được kết quả thất bại đó là do bản thân phần mềm hay là từ những yếu tố khác như môi trường thực thi, hay chính bản thân mã kiểm thử.

Khi kiểm thử thất bại, chúng ta luôn phải phân tích nguyên nhân, và nếu có nhiều kết quả thất bại thiếu nhất quán như vậy thì chi phí phải bỏ ra để phân tích sẽ rất lớn.

Vậy nên hãy loại bỏ những kiểm thử không ổn định khỏi bộ kiểm thử tự động; để có được bộ kết quả nhất quán nhất mà bạn có thể sử dụng.

Xem xét lại những kiểm thử tính hợp lệ tự động

Hạn chế sử dụng kiểm thử tự động để kiểm thử tính hợp lệ (validation), nếu không bạn sẽ thường xuyên phải cập nhật một lượng lớn các kiểm thử tự động. Nếu cần thiết, chỉ đặt kiểm thử tự động cho những trường hợp quan trọng nhất.

Việc kiểm thử tính hợp lệ tự động một cách bừa bãi có thể là một triệu chứng của việc triển khai kiểm thử tự động mà không dành đủ nỗ lực cho việc lên kế hoạch cũng như thiết kế.

Hãy luôn nhờ một người khác xem xét cẩn trọng những kiểm thử tính hợp lệ. Luôn đảm bảo các kiểm thử này được cập nhật sát với luật hợp lệ.

Đừng kiểm thử tự động những chức năng chưa ổn định

Khi một tính năng mới được phát triển, nó thường không ổn định. Có thể chức năng đó sai, thiếu hợp lý, và có thể sẽ không được phát triển trong tương lai. Cũng có thể tính năng đó sẽ sớm được thay đổi và cập nhật nhiều lần.

Trong cả hai trường hợp, những nỗ lực bỏ ra cho kiểm thử tự động sẽ đều bị lãng phí.

Do đó, tốt nhất là nên tự động hóa việc kiểm thử một tính năng khi tính năng đó đã đi vào ổn định và ít có khả năng thay đổi trong một sớm một chiều.

Đừng trông đợi Kiểm thử Tự động là phép thuật toàn năng

Động lực chính để tự động hóa kiểm thử là để giải phóng thời gian QA dành cho kiểm thử thaw dò, và tạo chỗ dựa để nhóm phát triển có thể tin rằng phần mềm vẫn hoạt động tốt sau khi cập nhật tính năng mới.

Đừng trông đợi kiểm thử tự động tìm được nhiều lỗi. Trọng thực tế, số lượng lỗi được tìm thấy bởi kiểm thử tự động luôn ít hơn nhiều so với từ kiểm thử thủ công và thăm dò.

Đừng đặt kiểm thử tự động làm chỗ dựa duy nhất

Khi cập nhật tính năng mới, các kiểm thử hồi quy tự động vẫn xanh, đó là chỗ dựa để nhóm phát triển có được sự tự tin. Sự phụ thuộc vào kiểm thử và theo đó, nhu cầu có một bộ kiểm thử hồi quy tốt ngày càng tăng lên.

Tuy nhiên, lưu ý rằng không phải tất cả các kiểm thử đều được tự động hóa, hay có thể tự động hóa, do đó hãy luôn kết hợp cả kiểm thử tự động lẫn kiểm thử thăm dò.

Đôi khi một vài sự thay đổi tính năng đáng lẽ khiến kiểm thử thất bại; nhưng bộ kiểm thử tự động rất dễ lâm vào không phát hiện ra, nếu nhóm phụ thuộc hoàn toàn vào kiểm thử tự động thì lỗi này sẽ rất dễ bị bỏ qua.

Đặt mục tiêu nhận phản hồi nhanh

Nhận phản hồi nhanh là một trong số những mục tiêu của kiểm thử tự động bởi nhóm phát triển luôn cần biết được liệu những gì họ đang phát triển có đang hoạt động tốt và không làm hỏng những chức năng hiện tại hay không.

Để có được vòng phản hồi nhanh, các bài kiểm

thử cần được tự động hóa ở cấp độ component hay API mà không cần phụ thuộc vào giao diện người dùng.

Các kiểm thử dựa trên giao diện người dùng chạy chậm hơn và dễ lỗi hơn khi giao diện người dùng thay đổi. Nói cách khác nghĩa là chức năng không đổi nhưng kiểm thử thất bại do giao diện người dùng thay đổi. Điều đó vô tình khiến kiểm thử mất đi tính tin cậy.

Hiểu rõ ngữ cảnh

Các kiểm thử có thể được tự động hóa ở nhiều tầng, Đơn vị, API, Service, GUI. Mỗi tầng có những mục đích kiểm thử khác nhau.

Kiểm thử đơn vị đảm bảo mã hoạt động tốt ở tầm mức lớp class, rằng mã có thể biên dịch được, và logic đúng như mong muốn. Kiểm thử ở tầng này mang tính kiểm tra (verification) hơn là phê chuẩn (validation).

Kiểm thử API hay Kiểm thử Tích hợp đảm bảo một tập hợp các chức năng và các class có thể làm việc tốt cùng nhau và dữ liệu có thể được chuyển giao từ một class đến một class khác.

Kiểm thử trên GUI mặt khác kiểm tra luồng và hành trình của người dùng. Nói chung chúng ta không kiểm thử tính năng từ UI. Tính năng nên được kiểm thử ở các tầng thấp hơn.

Mục đích chính của các kiểm thử trên UI là để đảm bảo toàn bộ hệ thống hoạt động theo một số kịch bản sử dụng phổ biến. Kiểm thử ở tầng này mang nhiều tính phê chuẩn hơn là kiểm tra.

Tại tầng UI, chúng ta tự động hóa các kịch bản sử dụng, thay vì các tính năng (user stories).

Đừng tự động hóa mọi kiểm thử

100% Test Coverage là không thể vì lẽ có tới hàng triệu sự kết hợp khác nhau. Chúng ta luôn chỉ thực hiện một tập con của các kiểm thử có khả năng thực hiện. Và kiểm thử tự động cũng thế.

Tạo ra một kiểm thử tự động cần thời gian và công sức, và nhắm đến mục tiêu "Tự động hóa Mọi Kiểm thử" sẽ cần rất nhiều thời gian và công sức, thứ mà thường chúng ta không có sẵn.

Thay vào đó, sử dụng một cách tiếp cận dựa trên mức rủi ro để xác định xem kiểm thử nào cần được tự động hóa. Để nhận được nhiều giá trị nhất từ kiểm thử tự động, chỉ nên tự động hóa những nghiệp vụ và kịch bản quan trọng nhất.

Quá nhiều kiểm thử tự động cũng sẽ khiến chi phí và độ khó của việc bảo trì tăng lên.

Một lưu ý khác nữa là không phải tất cả các kiểm thử đều có thể tự động hóa được. Có một số kiểm thử về bản chất rất phức tạp, yêu cầu nhiều ở hệ thống hạ tầng, và có thể không có tính nhất quán. Những trường hợp đó tốt nhất là để dành cho kiểm thử thủ công.

Áp dụng các kỹ thuật kiểm thử vào Kiểm thử Tự động

Những kỹ thuật kiểm thử mà bạn học được từ ISTQB không chỉ dành riêng cho kiểm thử thủ công. Chúng cũng được áp dụng để kiểm thử tự động. Những kỹ thuật như Phân tích Giá trị Biên, Phân vùng Tương đương, Kiểm thử Chuyển đổi Trạng thái, Kiểm thử cặp đôi... rất có giá trị trong kiểm thử tự động.

Đừng tự động hóa loạn

Để tối đa hóa giá trị của kiểm thử tự động cần có một quy trình QA tốt. Nếu quy trình QA đang hỗn loạn, việc thêm kiểm thử tự động vào chỉ làm cho tình hình tệ hơn.

Cố gắng trả lời rõ ràng những câu hỏi như Tự động hóa cái gì? Bao giờ thì tự động hóa? Khi nào thì thực thi các kiểm thử tự động? Ai sẽ tự động hóa các kiểm thử? Công cụ nào nên dùng?

Tổng kết

Bài viết này được dịch từ https://devqa.io/test-automation-tips-best-practices/. Thông tin được lấy từ những kinh nghiệm triển khai kiểm thử tự động trong thực tế của tác giả, và hi vọng sẽ có ích với bạn.

MỌI THỨ APPLE CÔNG BỐ TẠI

WWDC 2020

0h ngày 23/6, WWDC 2020 của Apple chính thức khai màn dưới hình thức trực tuyến. Tại đây, công ty giới thiệu iOS 14 - bản cập nhật lớn sắp tới dành cho iPhone, nâng cấp trên Apple Watch hay công bố kế hoạch chuyển từ chip Intel sang chip tự sản xuất trên Mac.

Dưới đây là mọi thứ Apple đã công bố tại WWDC 2020:

iOS 14

Apple bắt đầu sự kiện bằng màn ra mắt iOS 14 với nhiều tính năng mới. Một trong các thay đổi đáng chú ý nhất là màn hình chủ mới cho iPhone, nay có thêm không gian mới mang tên App Library. Chúng là các thư mục mà iPhone tạo ra tự động để nhóm những ứng dụng cùng danh mục, bao gồm ứng dụng vừa tải hay gợi ý.

Apple cũng thay đổi giao diện của các thanh tiện ích (widget) trong iOS 14. Widget mới có kích thước khác nhau và cung cấp nhiều dữ liệu hơn. Người dùng cũng có thể kéo chúng xuống màn hình chính để mở ra dễ hơn. Đây là đột phá lớn nhất với widget trong nhiều năm.

Ngoài ra, Apple giới thiệu App Clips, giúp người dùng sử dụng phiên bản nhẹ hơn của ứng dụng khi chỉ cần một tính năng nào đó hơn là mở toàn ứng dụng.

Siri cũng được cập nhật trong iOS 14 với giao diện mới và tính năng dịch nâng cao. Ứng dụng Messages có thêm tính năng "ghim" cuộc trò chuyện hay nhắc tới người nào đó khi chat nhóm. Với iOS 14, người dùng có thể dùng iPhone để mở cửa xe hơi.

“Chia tay” Intel

Apple xác nhận tin đồn trước đó về việc đang phát triển chip riêng dành cho máy Mac, mang tên Apple Silicon, và dần chuyển từ Intel. Chip của Apple sẽ hoạt động trong cùng môi trường với iPhone, iPad, giúp lập trình viên viết ứng dụng cho tất cả các sản phẩm của hãng dễ hơn nhiều.

Những máy tính Mac đầu tiên dùng chip mới của Apple sẽ được tung ra cuối năm 2020 nhưng công ty vẫn bán Mac dùng chip Intel. Đây được xem là bước đi táo bạo với Apple và cho phép công ty kiểm soát phần mềm, hiệu suất laptop, desktop tương lai nhiều hơn.

WatchOS 7

Bản cập nhật watchOS 7, cuối cùng cũng mang tới tính năng theo dõi giấc ngủ. Apple Watch sẽ dùng các cảm biến chuyển động, cảm biến đo nhịp tim và microphone để thu thập dữ liệu giấc ngủ, nhắc người dùng sạc chúng vào buổi sáng. Màn hình đồng hồ sẽ tắt vào buổi tối để bảo tồn năng lượng khi người dùng đang ngủ.

Đây là cập nhật đáng giá giúp Apple Watch tăng tốc đuổi theo các đối thủ như Fitbit, Garmin. Ngoài ra, đồng hồ của Apple có thêm các tính năng như chia sẻ mặt đồng hồ với người khác, phát hiện người dùng đã rửa tay đủ lâu hay chưa.

iPadOS 14

Apple cũng trình làng một số tính năng mới trên iPad và AirPods. Tính năng quan trọng nhất có mặt trên iPad là Scribble, cho phép viết trong bất kỳ ô nào bằng Apple Pencil và tự động chuyển sang văn bản. Các ứng dụng iPad cũng có sidebar mới để điều hướng dễ hơn. Thanh tiện ích mới trong iOS 14 cũng xuất hiện trong iPadOS 14.

Với AirPods, Apple thêm tính năng tự động chuyển giữa các thiết bị khác nhau, tạo ra trải nghiệm xuyên suốt khi dùng iPhone, iPad.

macOS Big Sur

Bản cập nhật hệ điều hành mới nhất cho máy tính Mac được Apple đặt tên là macOS Big Sur, mang đến thiết kế mới, Safari nâng cấp và các tính năng biến Mac trở nên gần gũi hơn với iPhone và iPad.

Một số tính năng mới bao gồm thiết kế mới với các biểu tượng ứng dụng được thiết kế lại, trung tâm điều khiển tương tự iPhone, cập nhật cho các ứng dụng như Messages, Apple Maps để có tính năng như iPhone, trình duyệt Safari mới.

Nhìn lại WWDC 2020, có thể thấy Apple đang muốn thống nhất trải nghiệm sử dụng iPhone, iPad và Mac trong khi mang đến khả năng tùy biến và linh hoạt cao hơn. Điều đó có thể thấy rõ nhất trong hệ điều hành macOS Big Sur.

Nguồn: https://ictnews.vietnamnet.vn/

HÀNH TRÌNH TỪ 1% THÀNH CÔNG ĐẾN CỘT MỐC LỊCH SỬ CỦA SPACEX

Cột mốc mới cho ngành hàng không vũ trụ vừa được xác lập: lần đầu tiên một công ty tư nhân phát triển thành công phương tiện tối tân đưa con người vào vũ trụ.

Đúng 15h22 ngày 30-5 (giờ Mỹ), tức rạng sáng 31-5 theo giờ Việt Nam, tên lửa đẩy Falcon 9 của Công ty SpaceX đã rời bệ phóng từ Trung tâm vũ trụ Kennedy ở bang Florida.

Vận tốc của tên lửa đẩy tăng nhanh chóng chỉ sau 2 phút. Tầng một của tên lửa tách ra ở độ cao hơn 87km và quay trở về mặt đất khoảng 7 phút sau đó, trong lúc động cơ ở tầng 2 khởi động và tiếp tục đưa tàu Crew Dragon tiến vào quỹ đạo.

Sau hành trình bay tự do dài gần 19 tiếng, tối 31-5 (giờ Việt Nam), tàu Crew Dragon chở theo phi hành gia NASA Doug Hurley và Bob Behnken đã kết nối thành công với Trạm không gian quốc tế (ISS).

Khi thành lập SpaceX năm 2002, tỉ phú Elon Musk từng cho rằng những gì công ty ông thực hiện là một sứ mạng không tưởng. Musk ước tính, khả năng thành công của công ty chỉ từ 0,1-1%.

Tuy nhiên đến nay, SpaceX không chỉ tạo nên một bước ngoặt mới cho Elon Musk mà còn cả ngành hàng không vũ trụ Mỹ và trên thế giới.

Chuyến bay đánh dấu SpaceX trở thành doanh nghiệp tư nhân đầu tiên chở phi hành gia vào vũ trụ.
Trong ảnh: 2 thành viên mới từ Crew Dragon vừa đặt chân lên ISS - Ảnh: NASA

Doug Hurley và Bob Behnken là 2 người nhận nhiệm vụ lần này. Cả hai cũng từng có kinh nghiệm đi trên các tàu con thoi. Tại ISS, họ sẽ kết hợp với đồng nghiệp Chris Cassidy và 2 phi hành gia người Nga ở đây thực hiện các nhiệm vụ khoa học - Ảnh: NASA

Tàu Crew Dragon 4 chỗ, hình dạng giống với các tàu Apollo trước đây nhưng được trang bị các công nghệ hiện đại. "Chúng tôi đã ngủ một ít vào đêm qua. Chúng tôi ngạc nhiên khi có thể ngủ rất ngon trên chiếc tàu mới này", Behnken cho biết. Trước đó, tên lửa Falcon 9 đưa tàu Crew Dragon lên quỹ đạo từ bệ phóng 39A ở Trung tâm Vũ trụ Kennedy của NASA, Florida (Mỹ) vào ngày 30-5 (giờ Mỹ) - Ảnh: NASA

SpaceX từng vượt qua nhiều trở ngại trước khi tạo được bước ngoặt như hiện nay. Trước đó hồi tháng 4-2019, khoang tàu Crew Dragon từng bị hủy trong thử nghiệm trên mặt đất ở Cape Canaveral. Năm 2016, một tên lửa Falcon 9 cũng phát nổ sau khi bay 139 giây. Trong ảnh là đợt phát nổ vào năm 2016 - Ảnh: NASA

Ban đầu, ngày phóng tàu dự kiến tổ chức vào 28-5. Đến phút chót, SpaceX hủy phóng do thời tiết xấu. Ngay cả ngày 30-5, Falcon 9 vẫn chưa chắc có thể cất cánh do trời nhiều mây và rủi ro sét đánh - Ảnh: NASA

Thử thách với Crew Dragon vẫn còn phía trước. Sau khi các phi hành gia xong nhiệm vụ trong 4 tháng tới, Crew Dragon tiếp tục chở họ về Trái đất. Chuyến đi này không dễ dàng, dù Crew Dragon được trang bị một lớp lá chắn nhiệt và 4 chiếc dù nhằm giảm tốc độ của tàu vũ trụ khi rơi xuống Đại Tây Dương. Đến lúc đó mới có thể khẳng định nhiệm vụ này của Crew Dragon hoàn toàn thành công hay không. Trong ảnh: Doug Hurley và Bob Behnken đang điều khiển Crew Dragon - Ảnh: NASA

Tổng thống Donald Trump, đệ nhất phu nhân Melania và Phó tổng thống Mike Pence đã đến theo dõi buổi phóng tàu Ảnh: NASA

Kể từ chương trình tàu con thoi 30 năm kết thúc năm 2011, Mỹ đều nhờ vào các tàu của Nga nếu muốn lên ISS với chi phí khoảng 86 triệu USD mỗi người. Năm 2014, NASA ký hợp đồng gần 6,8 tỉ USD với hai tập đoàn là Boeing và SpaceX chế tạo tàu vũ trụ thay thế - Ảnh: NASA

Hiện tại, SpaceX có khoảng 7.000 nhân viên và ngân sách hoạt động là 1,7 tỉ USD/năm. Trong khi đó, NASA có khoảng 17.500 nhân viên và ngân sách hoạt động 20 tỉ USD/năm. Dù hơi "lép vế" về nhân lực và tài chính, SpaceX hiện đi tiên phong so với NASA trong việc phát triển rất nhiều công nghệ phức tạp và táo bạo - Ảnh: NASA

nguồn: tuoitre.vn

BAN BIÊN TẬP

Nguyễn Khắc Nhật

Nguyễn Việt Khoa

Nguyễn Khánh Tùng

Nguyễn Bình Sơn

Đặng Huy Hoà

Dư Thanh Hoàng

Đỗ Minh Hải

Nguyễn Thị Hiền

THIẾT KẾ

Đỗ Đình Tâm