Tạp_Chí
{LậpTrình ():}

Tạp chí LẬP TRÌNH

tapchilaptrinh.vn

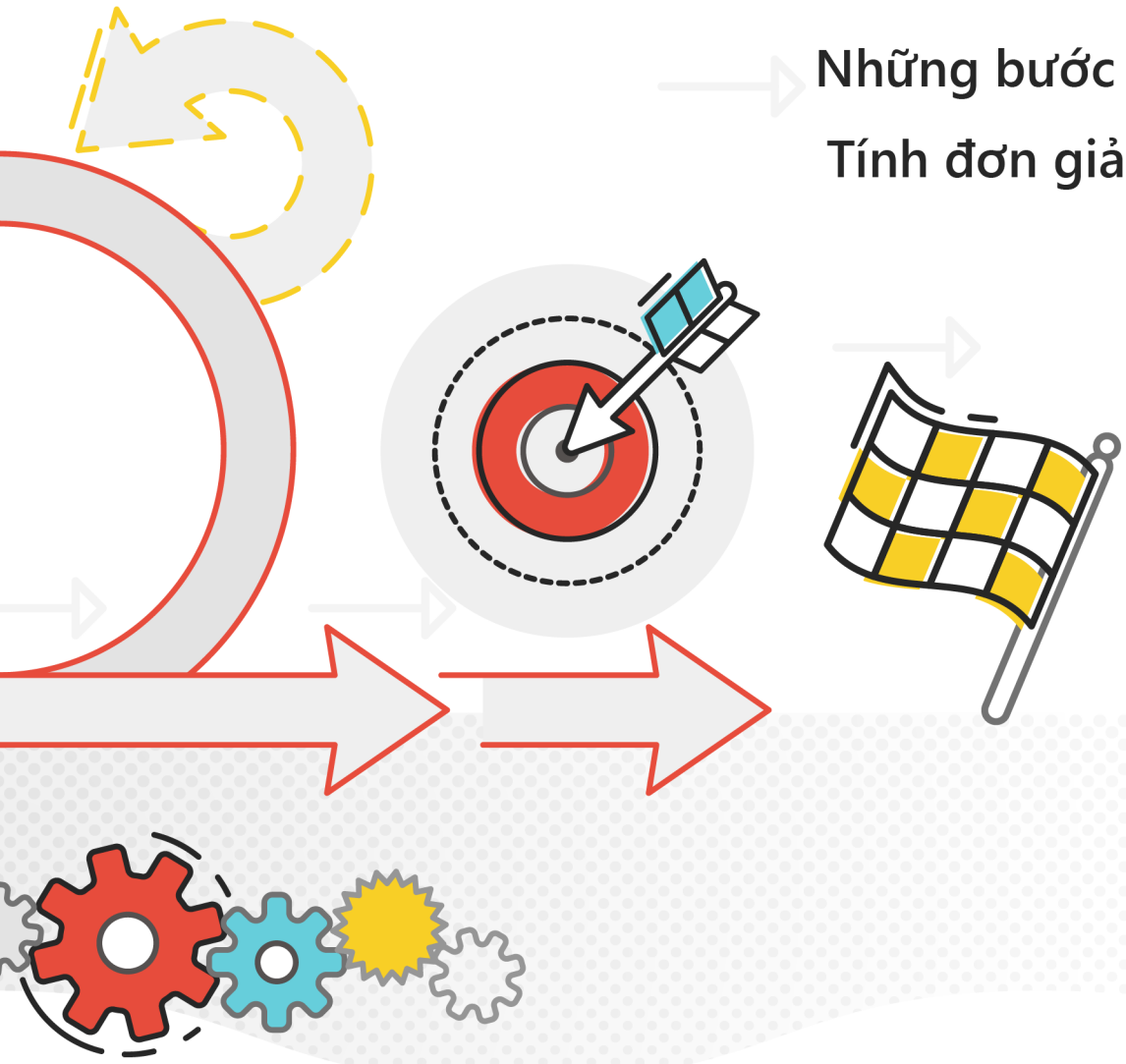
Agile Testing

Agile Developer

Dive in Linux - phần một:
Nhân và Bản phân phối

Unit Testing
Những bước chân đầu tiên

Tính đơn giản là chìa khóa



Số phát hành Vol.4

05
2020

Tạp chí lập trình
VOL.4

SỐ NÀY CÓ GÌ?

LỜI MỞ ĐẦU

Thân chào các độc giả của **Tạp chí Lập trình,**

Thế giới lập trình đúng là biến đổi liên tục, chỉ sau một vài năm thì những công nghệ, những nền tảng, những phương pháp và mô hình cũ đều đã có những thay đổi rất là sâu sắc. Từ những năm mà việc phát triển và phát hành phần mềm diễn ra rất nhỏ lẻ và chậm chạp, chúng ta đã nhanh chóng đi đến giai đoạn mà việc phát hành "liên tục" được coi như là điều hiển nhiên trong làm phần mềm. Yêu cầu của xã hội thay đổi, yêu cầu của công nghệ thay đổi, và do đó các lập trình viên chúng ta cũng cần tự thay đổi mình để thích ứng tốt hơn và tạo ra được những sản phẩm với chất lượng và tốc độ vượt bậc.

Trong số này, Ban biên tập xin gửi đến mọi người chuỗi bài viết xoay quanh chủ đề "Agile Developer". Đây là một thuật ngữ không còn quá xa lạ đối với thế giới, không còn chỉ là cuộc trò chuyện riêng giữa những người là chuyên gia đầu ngành nữa, mà ngày nay nó đã trở thành những tiêu chuẩn đối với những lập trình viên bình thường. Agile Developer - cho đến hiện tại, vẫn là một đích đến, một tiêu chuẩn, một đẳng cấp mà các lập trình viên hướng đến.

Trong khuôn khổ của một ấn phẩm thì e khó có thể đề cập đầy đủ đến tất cả các khía cạnh của một lĩnh vực rộng như Agile Developer, do đó Ban biên tập rất mong muốn qua những nội dung trong Vol4 này chúng ta lại sẽ có thêm những cuộc thảo luận, những đóng góp để cùng nhau tìm hiểu, học tập, nghiên cứu. Ban biên tập rất mong muốn mỗi người đọc của Tạp chí Lập trình sẽ gặt hái được những giá trị bổ ích thông qua ấn phẩm lần này.

Cảm ơn và chúc mọi người thật nhiều hứng thú với nghề lập trình.

Ban biên tập Tạp chí Lập trình

AGILE DEVELOPER
CÓ NGHĨA LÀ THẾ NÀO?

Nguyễn Việt Khoa

Vai trò của một Agile Developer (Nhà Phát triển Phần mềm Linh hoạt) không được hiểu rõ và rất nhiều nhà phát triển có thể gặp khó khăn khi thực hiện việc chuyển đổi sang làm việc theo Agile.

Trong môi trường hướng-kế hoạch truyền thống (còn gọi là "Waterfall" – mô hình thác nước), một nhà phát triển gần như chỉ ngồi cố định tại khoang của mình trong văn phòng, đeo tai nghe "lắc lư" theo nhạc và tách biệt với thế giới còn lại để làm môt việc đơn giản là viết ra các dòng mã. Điều đó là không thể trong một môi trường Agile thực sự.

Vai trò của một nhà phát triển trong môi trường Agile được mở rộng hơn đáng kể, họ cần phải:

- Chịu trách nhiệm ước tính, lên kế hoạch, quản lý tất cả các công việc của mình và báo cáo tiến độ. Vai trò này về cơ bản là những gì mà một nhà quản lý dự án có thể làm ở quy mô rất nhỏ.
- Cộng tác chặt chẽ với tất cả các thành viên khác của nhóm để chia sẻ trách nhiệm với những mục tiêu chung của nhóm. Vai trò này cũng tương tự như những gì một nhà quản lý dự án (PM) phải làm nhưng thay vì chỉ "Nhà Quản trị Dự án" phải làm thì trách nhiệm được phân chia cho các thành viên trong nhóm.
- Chịu trách nhiệm về chất lượng của phần mềm mà nhóm phát triển tạo ra. Thay vì chuyển mã nguồn cho một nhóm riêng biệt và độc lập kiểm thử, toàn bộ nhóm phát triển chịu trách nhiệm về chất lượng của sản phẩm do mình làm ra. Nhà phát triển có hoặc không thể tự mình kiểm thử nhưng điểm mấu chốt là chất lượng của mã nguồn không phải là trách nhiệm của một vài cá nhân nào đó.
- Tương tác với người dùng khi cần thiết để làm rõ các yêu cầu. Các nhà phát triển thường không được cung cấp các yêu cầu cụ thể, chi tiết. Thông thường, nhà phát triển sẽ nhận được một số user story khá chung chung kiểu như là "một tài liệu sơ giản để thảo luận về sản phẩm" và họ được mong đợi sẽ tương tác với Product Owner và người dùng khi cần thiết để xác định rõ hơn về những yêu cầu này. Về cơ bản, khi đó nhà phát triển có vai trò tương đương với vai trò Chuyên viên Phân tích Nghiệp vụ (BA) trên quy mô rất nhỏ.

Vai trò của một nhà phát triển trong môi trường Agile là khác biệt đáng kể và một số nhà phát triển có thể gặp khó khăn khi quyết định chuyển đổi sang mô hình này.

TÍNH ĐƠN GIẢN LÀ CHÌA KHÓA

Nguyễn Việt Khoa

Việc thiết kế đơn giản luôn mất thời gian hơn việc thiết kế phức tạp. Vì vậy, hãy làm những việc đơn giản nhất mà trước mắt hoạt động. Nếu bạn thấy có chỗ nào đó phức tạp thì hãy thay thế nó bằng thứ đơn giản. Lúc này, việc thay thế mã phức tạp của bạn sẽ luôn nhanh và rẻ hơn so với khi bạn lãng phí nhiều thời gian vào nó.

Rất nhiều người cố gắng đo lường sự đơn giản. Sự đơn giản thách thức sự đo lường bởi vì nó là một phẩm chất rất chủ quan. Sự đơn giản của người này là phức tạp của người khác. Thêm một công nghệ tiên tiến có thể đơn giản hóa một ứng dụng và tạo ra một mớ hoàn toàn hỗn độn cho mọi thứ khác.

Trong dự án cả đội quyết định điều gì là đơn giản. Cùng nhau bạn đánh giá mã nguồn của mình một cách chủ quan. Tôi khuyên bạn 4 giá trị chủ quan : kiểm thử được (Testable), dễ hiểu (Understandable), tìm kiếm được (Browsable) và dễ giải thích (Explainable).

Kiểm thử được nghĩa là bạn có thể viết các kiểm thử đơn vị và các kiểm thử chấp nhận để kiểm thử lỗi tự động. Điều này ảnh hưởng đến thiết kế tổng thể và tính phụ thuộc của các đối tượng trong ứng dụng của bạn. Hãy xé nhỏ,hệ thống của bạn thành các đơn vị nhỏ có thể kiểm thử được.

Tìm kiếm được tức là khả năng tìm những gì bạn muốn.Tên tốt giúp bạn tìm kiếm. Sử dụng tính đa hình (polymorphism), ủy thác (delegation), và thừa kế (inheritance) một cách đúng đắn giúp bạn tìm kiếm chính xác mọi thứ.

AGILE TESTING

Nguyễn Thế Nghị

Đã 9h tối rồi mà một số bạn vẫn đang ở trong phòng lab loay hoay với với các antenna của các trạm phát sóng BTS cùng với khoảng chục chiếc điện thoại di động trên bàn. Khuôn mặt của các bạn đã thấm mệt, bụng đã đói cồn cào nhưng phải cố gắng hoàn thành bộ test case trước khi ra về. Đó là một dự án đã trễ deadline vài tháng trời, nhóm viết code vừa mới hoàn thành phần viết code và chuyển giao sang cho nhóm kiểm thử.

Đó là một dự án waterfall mà tôi biết có sự tham gia của nhóm viết code, kiểm thử ở cả Ấn Độ và Việt Nam, số lượng thành viên viết code khoảng 20 người và kiểm thử khoảng 5 người. Nhóm viết code và làm kiểm thử là tách bạch nhau và chỉ khi nhóm viết code hoàn thành thì nhóm kiểm thử mới bắt đầu. Do đó, tiến độ của nhóm viết code sẽ ảnh hưởng đến tiến độ của nhóm kiểm thử.

Ở một dự án khác mà tôi tham gia theo mô hình Scrum, các thành viên viết code và kiểm thử cùng chung một nhóm gồm 7 người. Các bạn kiểm thử không phải ngồi đợi vài tháng trời mới có cái để kiểm tra mà phải tham gia vào từ rất sớm.

Khi có yêu cầu từ khách hàng, các bạn cùng với nhóm viết code phân tích, đưa ra nhận định về ảnh hưởng đến toàn hệ thống, lên kế hoạch cho mỗi phân đoạn ngắn 2 tuần, có tiêu chí chấp nhận và tiêu chí hoàn thành cùng với các kịch bản kiểm thử. Những công việc sau buổi lập kế hoạch được phân bổ theo độ ưu tiên, phân nhỏ và có thể hoàn thành nhanh trong vài tiếng hoặc dài nhất là 3 ngày, do đó, ngay trong ngày hôm sau, nhóm viết code đã có thể bàn giao những tính năng có thể kiểm thử được và khi đó, các bạn trong nhóm kiểm thử có thể bắt tay vào ngay vào việc kiểm thử mà không phải ngồi không xơi nước hết ngày ngày sang ngày khác.

Một điểm thú vị nữa trong việc tổ chức nhóm nhỏ theo Scrum đó là các bạn viết code cũng tham gia vào quá trình kiểm thử ngay khi lúc viết code và trước khi bàn giao.

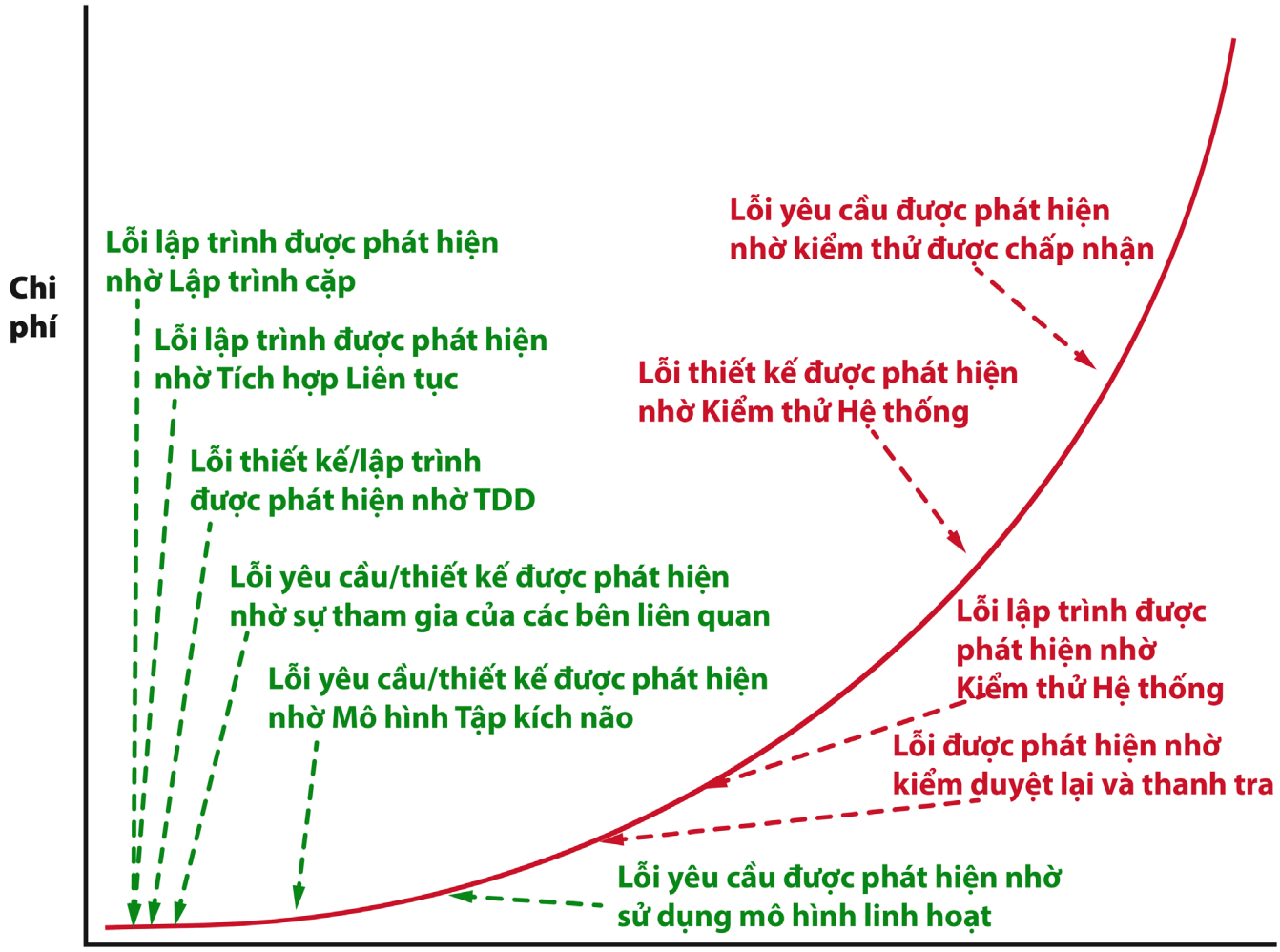
Ngay khi viết code, nhóm ứng dụng kĩ thuật Test-driven development (TDD) hay Behavior-driven development (BDD) để viết code kiểm tra trước khi thực thi. Việc làm này giúp cho việc kiểm thử diễn ra rất sớm và tất nhiên chi phí cho việc phát hiện lỗi và sửa lỗi ở giai đoạn này là rất rẻ so với việc phát hiện lỗi và sửa lỗi khi sản phẩm đã lên bàn giao cho người dùng.

Đôi khi, 2 bạn viết code cùng bắt cặp với nhau để phát triển một tính năng. Việc làm như thế này đối với một số công ty khó diễn ra vì nhiều lý do nhưng với nhóm tôi thì nó phát huy hiệu quả rất cao. Khi làm việc với nhau, người này làm người kia đóng góp ý kiến và kiểm tra ngay luôn lúc đang làm nên lỗi được phát hiện rất sớm.

Trước khi bàn giao sản phẩm, nhóm viết code cùng với nhóm kiểm thử có thể có một phần kiểm thử hồi quy (regression test) tất cả các tính năng cũ và mới trước khi bàn giao vì lúc này, không có những tính năng mới cần phát triển. Chi phí cho việc phát triển lỗi và sửa lỗi ở giai đoạn này vẫn rẻ hơn sau khi đưa ra thị trường.

Chất lượng của sản phẩm được tăng lên theo thời gian khi cả nhóm có sự đồng thuận cũng như phối hợp tốt với nhau theo thời gian.

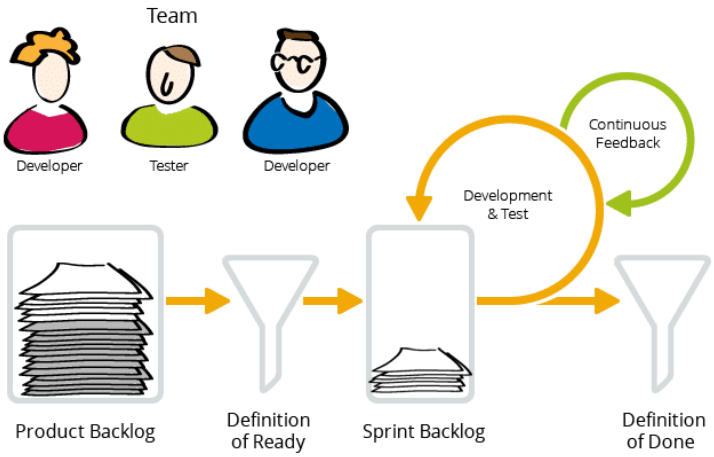
Xem chi phí của việc phát hiện lỗi và sửa lỗi qua từng công đoạn:



Copyright 2006-2009 Scott W.Ambler

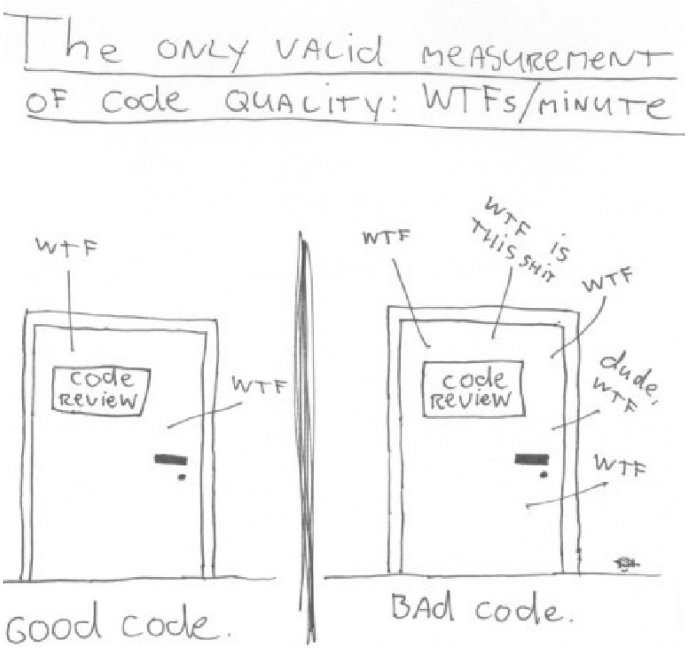
Một đặc điểm quan trọng của các nhân viên kiểm thử trong mô hình Scrum là cần có những kĩ năng để thích nghi với quá trình thay đổi liên tục đặc biệt là kĩ năng về giao tiếp với các thành viên trong nhóm phát triển, nhóm kinh doanh để cập nhật thông tin liên tục thay vì phải theo những tài liệu định trước và thường xuyên không được cập nhật như trong mô hình waterfall.

Như vậy, việc kiểm thử trong mô hình Agile/ Scrum có những điểm thay đổi tích cực so với mô hình truyền thống đó là: việc kiểm thử diễn ra từ rất sớm, các thành viên trong nhóm đều có thể đóng góp vào quá trình kiểm thử, và cần có kĩ năng giao tiếp tốt để thích nghi với quá trình thay đổi.



TÁI CẤU TRÚC (REFACTOR) TRONG PHP

Dư Thanh Hoàng
Nguồn: https://medium.com/hackernoon



Các nhà phát triển tốt được xác định bởi chất lượng mã của họ. Trong ngành công nghiệp phần mềm, viết mã tốt có nghĩa là tiết kiệm tiền có thể được đầu tư vào thử nghiệm, cập nhật, mở rộng hoặc sửa lỗi. Trong bài viết này, tôi sẽ cho bạn thấy các ví dụ thực tế về một số kỹ thuật và ý tưởng sẽ giúp bạn viết mã sạch của mình và tái cấu trúc lại nó để làm cho nó mạnh mẽ hơn và mô đun hóa. Những kỹ thuật này không chỉ giúp bạn tái cấu trúc mã cũ mà còn cung cấp cho bạn những ý tưởng tuyệt vời về cách viết mã sạch từ bây giờ.

Tái cấu trúc là gì và tại sao chúng ta cần nó? Tái cấu trúc đề cập đến các kỹ thuật và các bước giúp bạn viết mã sạch. Điều này rất quan trọng đối với các nhà phát triển khác, những người sau đó sẽ có thể đọc, mở rộng và sử dụng lại mã mà không cần phải chỉnh sửa nhiều.

Các dòng tiếp theo sẽ cho bạn thấy một số ví dụ về tái cấu trúc mã kế thừa và làm cho nó tốt hơn. Không bao giờ tái cấu trúc lại mã mà không có kiểm thử

Lời khuyên đầu tiên của tôi là đừng bao giờ bắt đầu tái cấu trúc mà không có kiểm thử.

Tôi đoán lý do rất rõ ràng: Bạn sẽ kết thúc với các chức năng bị hỏng rất khó khắc phục vì bạn sẽ không thể tìm ra những gì bị hỏng. Do đó, nếu bạn cần cấu trúc lại nó, hãy bắt đầu với thử nghiệm nó trước. Hãy chắc chắn rằng phần bạn sẽ tái cấu trúc được bao phủ bởi các bài kiểm tra. Kiểm tra phân tích mã PHPUnit.

*****Bắt đầu tái cấu trúc từ mã của bạn***** Hãy nhìn vào đoạn code tiếp theo. Đây là một dự án thực sự cho một hệ thống quản lý khách sạn mà tôi tìm thấy trên Github. Đây là một dự án nguồn mở thực sự thực sự code tệ nhất.

```
function add()
{
    $viewdata = array();
    if ($this->input->post("type") && $this->in-
put->post('price')) /* & $this -> input ->post
("quantity") */ {
        $type = $this->input->post('type');
        $price = $this->input->post('price');
        $details = $this->input->post('de-
tails');
        $quantity = $this->input->post('quanti-
ty');

        if (count($this->room_m->getRoomType($-
type)) == 0) {
            $this->room_m->addRoomType($type,
$price, $details, $quantity);
            redirect("/room-type");
        } else {
            $viewdata['error'] = "Room type al-
read exista";
        }
    }
    $data = array('title' => 'Add Room Type -
DB Hotel Management System', 'page' => 'room_
type');
    $this->load->view('header', $data);
    $this->load->view('room-type/add', $viewda-
ta);
    $this->load->view('footer');
}
```

Như bạn có thể thấy trong phương pháp này, có ba cấp độ được đánh dấu màu đỏ. cnhất phải là câu lệnh if/other lồng trong điều kiện if đầu tiên. Thông thường, điểm sâu nhất là tập trung vào một logic duy nhất giúp dễ dàng cấu trúc lại.

Như bạn có thể thấy trong phương pháp này, có ba cấp độ được đánh dấu màu đỏ. Cấp độ bên trong phải là câu lệnh if/else lồng trong điều kiện if đầu tiên. Thông thường, cấp độ cuối là tập trung vào một logic duy nhất giúp dễ dàng cấu trúc lại.

Làm cho các phương thức của bạn ngắn hơn bằng cách chia chúng thành các phương thức nhỏ hơn hoặc các tệp cấu hình file/DB table

Có thể, trong trường hợp này, ta có thể trích xuất nó thành một phương thức riêng tư như sau:

```
function addRoom (RoomType $type, int $price,
string $details, int $quantity)
{
    if (count($this->room_m->getRoomType($type)
== 0)){
        $this->room_m->addRoomType($type,$-
price,$details,$quantity);
        //this redirection will prevent the ex-
ception from being throw
        redirect(“/room-type”);
    }

    throw new Exception(“Room type already ex-
ists”);
}
```

Bây giờ, hãy xem phương thức add () sau khi tái cấu trúc các phần khác. Nó là nhiều sạch hơn, có thể đọc và kiểm tra.

```
function add()
{
    $viewData = [];
    if ($this->input->post(“type”) && $this->in-
put->post(‘price’)) {
        $data = $this->fetchPostData();
        try {
            $this->addRoom($data[‘type’],
$data[‘price’], $data[‘details’], $data[‘quanti-
ty’]);
        } catch (Exception $exception) {
            $viewData[‘error’] = $excep-
tion->getMessage();
        }
    }
    $this->initializeView($viewData);
}
```

Luôn sử dụng {} trong câu lệnh if...else

Hầu hết các ngôn ngữ lập trình đều hỗ trợ một câu lệnh if và một số nhà phát triển sử dụng nó vì nó đơn giản, tuy nhiên, nó không thể đọc được và dễ gây ra sự cố do chỉ một dòng trống có thể phá vỡ điều kiện và bắt đầu gặp sự cố. Xem sự khác biệt giữa hai ví dụ:

```
// NOT GOOD
function check_login()
{
    if (!UID)
        redirect(“login”);
}

// VERY GOOD
function check_login()
{
    if (!UID) {
        redirect(“login”);
    }
}
```

Không sử dụng số magic number & magic string:

Trong ví dụ tiếp theo, bạn nhận thấy nếu có hơn 250, nó sẽ trả về một thông báo lỗi. Trong trường hợp này, 250 được coi là một con số ma thuật. Nếu bạn không phải là nhà phát triển đã viết nó, sẽ rất khó để tìm ra những gì nó đại diện.

```
function availableRooms($room)
{
    if ($room > 250) {
        return ‘No rooms available’;
    } else {
        return true;
    }
}
```

Để tái cấu trúc lại phương pháp này, chúng ta có thể hình dung ra 250 là số phòng tối đa. Do đó, thay vì mã hóa nó, chúng ta có thể trích xuất nó thành biến $maxAvAvailableRooms. Bây giờ, nó dễ hiểu hơn đối với các nhà phát triển khác.

```
function availableRooms($room)
{
    $maxAvailableRooms = 250;
    if ($room > $maxAvailableRooms) {
        return ‘No rooms available’;
    } else {
        return true;
    }
}
```

Không sử dụng các câu lệnh khác nếu bạn không cần:

Trong cùng một hàm availablerooms () bạn nhận thấy câu lệnh if, trong đó chúng ta có thể dễ dàng loại bỏ phần khác và logic vẫn sẽ giống nhau.

```
function availableRooms($room)
{
    $maxAvailableRooms = 250;
    if ($room > $maxAvailableRooms) {
        return 'No rooms available';
    }
    return true;
}
```

Sử dụng tên có ý nghĩa cho các phương thức, biến và kiểm tra của bạn Trong ví dụ sau, bạn có thể thấy rằng có hai phương thức từ hệ thống quản lý khách sạn có tên là "index () và room_m ()". Đối với tôi, tôi không thể xác định mục đích của họ là gì. Tôi nghĩ sẽ dễ hiểu hơn nếu tên của họ được mô tả.

```
function index()
{
    $room = $this->room_m->get_rooms();

    $viewData = array('rooms' => $room);

    $data = array('title' => 'Rooms - DB Hotel
Management System', 'page' => 'room');
    $this->load->view('header', $data);
    $this->load->view('room/list', $viewData);
    $this->load->view('footer');
}
```

Sử dụng các khả năng tối đa của ngôn ngữ lập trình:

Nhiều nhà phát triển không sử dụng toàn bộ khả năng của ngôn ngữ lập trình họ sử dụng. Nhiều trong số các tính năng này có thể giúp bạn tiết kiệm rất nhiều nỗ lực và làm cho mã của bạn mạnh mẽ hơn. Hãy xem các ví dụ tiếp theo và chú ý làm thế nào có thể dễ dàng đạt được kết quả tương tự với ít mã hơn bằng cách chỉ sử dụng gợi ý loại.

```
function calcDiscount(string $name, int $age):
array
{
    return [
        'name' => $name,
        'age' => $age,
    ];
}
```

```
function calcDiscount($name, $age)
{
    if (!\is_string($name)) {
        throw new \Exception('Provided name is
not valid');
    }
    if (!\is_int($age)) {
        throw new \Exception('Provided age is
not valid');
    }
    $formattedInfo = array();
    $formattedInfo['name'] = $name;
    $formattedInfo['age'] = $age;

    return $formattedInfo;
}
```

CÂY NHỊ PHÂN TÌM KIẾM

Nguyễn Khánh Tùng

(Nguồn: Đỗ Xuân Lôi – Cấu trúc dữ liệu và giải thuật- NXB Đại Học Quốc Gia Hà Nội)

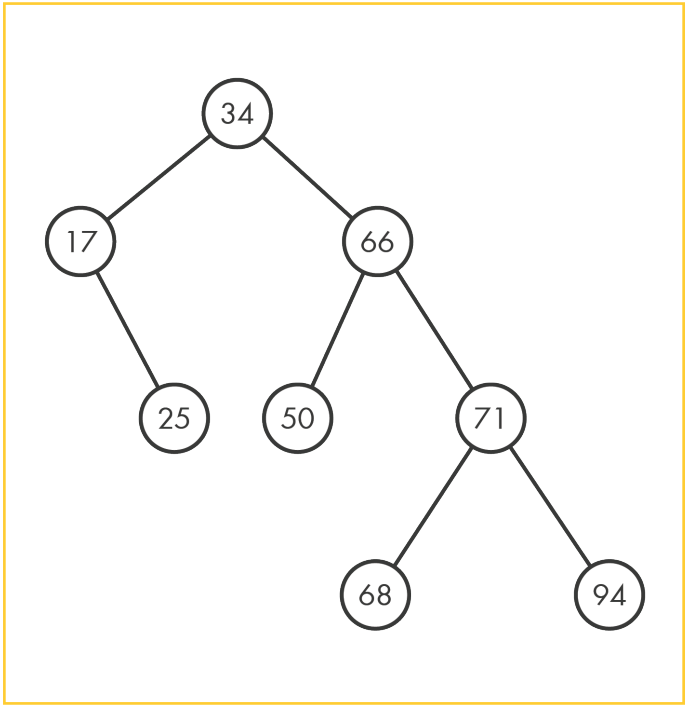
1. Định nghĩa

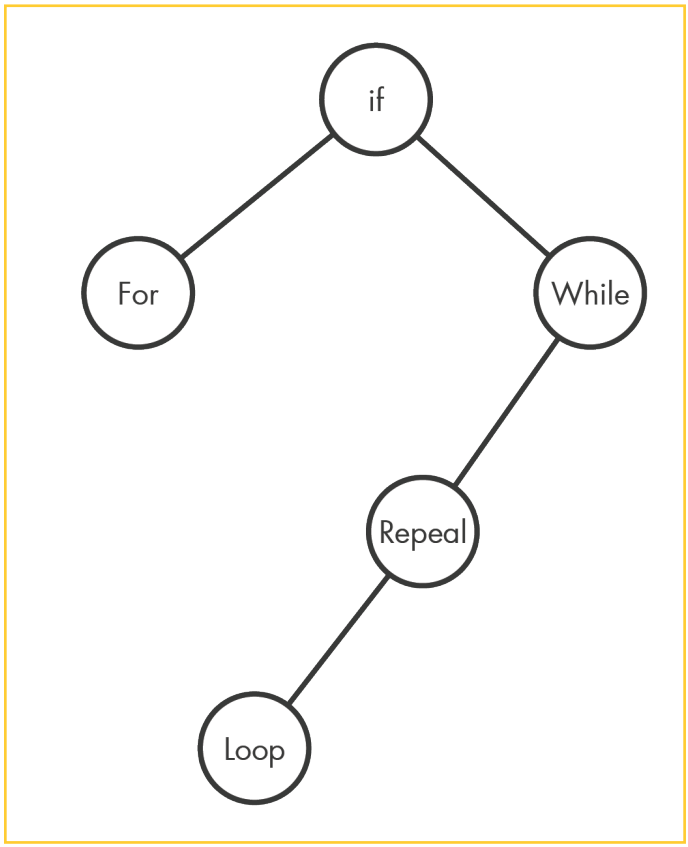
Cây nhị phân tìm kiếm ứng với n khoá k1,...,kn là một cây nhị phân mà mỗi nút của nó đều được gán một giá trị khoá nào đó trong các giá trị khoá đã cho và đối với mọi nút trên cây tính chất sau đây luôn được thoả mãn:

- Mọi khoá thuộc cây con trái nút đó đều nhỏ hơn khoá ứng với nút đó.
- Mọi khoá thuộc cây con phải nút đó đều lớn hơn khoá ứng với nút đó.

Ở đây thứ tự chon, ta quy ước là thứ tự tăng dần đối với số và thứ tự từ điển đối với chữ.

Sau đây là ví dụ về cây nhị phân tìm kiếm đối với khoá là số và chữ:





2. Giải thuật tìm kiếm

Đối với một cây nhị phân tìm kiếm để tìm xem một khoá X nào đó có trên cây đó không, ta có thể thực hiện như sau:

So sánh X với khoá ở gốc, và một trong bốn tình huống sau đây sẽ xuất hiện:

- Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm không thoả.
- X trùng với khoá gốc: Phép tìm kiếm được thoả.
- X nhỏ hơn khoá ở gốc: tìm kiếm tiếp tục thực hiện bằng cách xét cây con trái của gốc với cách làm tương tự.
- X lớn hơn khoá ở gốc: tìm kiếm tiếp tục thực

hiện bằng cách xét cây con phải của gốc với cách làm tương tự.

Như với cây ở hình trên, nếu X=68 ta thực hiện:

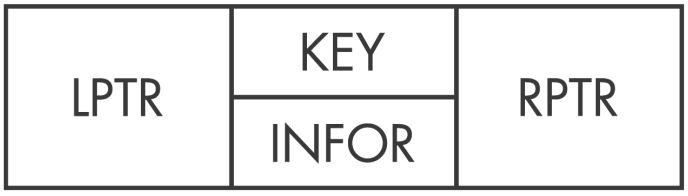
- So sánh X với 34: X>34, ta chuyển sang cây con phải.
- So sánh X với 66: X>66, ta chuyển sang cây con phải.
- So sánh X với 71: X<71, ta chuyển sang cây con trái.
- So sánh X với 68: X=68, vậy tìm kiếm đã được thoả.

Nhưng nếu X=30 thì phải:

- So sánh X với 34: X<34, ta chuyển sang cây con trái.
- So sánh X với 17: X>17, ta chuyển sang cây con phải.
- So sánh X với 25: X>25, ta chuyển sang cây con phải, nhưng cây con phải rỗng, vậy phép tìm kiếm không thoả.

Nếu sau phép tìm kiếm không thoả, ta bổ sung luôn X vào cây nhị phân tìm kiếm (như ví dụ vừa xét ta bổ sung khoá 30 vào thành con phải của nút 25) ta thấy phép bổ sung này thực hiện rất đơn giản và không làm ảnh hưởng gì tới vị trí của các khoá hiện có trên cây cả, tính chất của cây nhị phân tìm kiếm vẫn được đảm bảo.

Nếu giả sử quy cách mỗi nút của cây nhị phân tìm kiếm có dạng:



Ở đây trường LPTR và RPTR chứa các con trỏ trỏ tới gốc cây con trái và cây con phải của nút.

Trường KEY ghi nhận giá trị khoá tương ứng của nút, trường INFOR ghi nhận các thông tin khác, không có vai trò trong tìm kiếm.

Giải thuật tìm kiếm có bổ sung trên cây nhị phân tìm kiếm sẽ như sau:

```
Procedure BST(T,X,q);
{Thủ tục này thực hiện tìm kiếm trên cây nhị phân
tìm kiếm, có gốc được trỏ bởi T, nút có khoá bằng
X. Nếu tìm kiếm được thoả thì đưa ra con trỏ q trỏ
tới nút đó, nếu tìm kiếm không thoả thì bổ sung
nút mới có khoá là X vào T và đưa ra con trỏ q trỏ
tới nút mới đó kèm theo thông báo}.

1.{Khởi tạo con trỏ}
    P:=null; q:=T;

2.{Tìm kiếm}
    While q ≠ null do
    Case
       X < KEY (q): p :=q; q:= LPTR(q);
       X = KEY (q) : return;
       X > KEY (q) :p :=q; q:= RPTR(q);
    End case;

3.{Bổ sung}
    Call new (q);
       KEY(q) :=X;
       LPTR(q) :=RPTR(q) :=null;
    Case
       T = null: T:=q; {cây rỗng, đã bổ sung}
       X < KEY(p) : LPTR(p) :=q;
    Else : RPTR(p) :=q
    End case;
    Write("không thấy, đã bổ sung");
Return;
```

Với giải thuật trên có thể suy ra: ta có thể dựng được cây nhị phân tìm kiếm ứng với một dãy khoá đưa vào bằng cách liên tục bổ sung các nút ứng với từng khoá, bắt đầu từ một cây rỗng. Tất nhiên thoạt đầu phải dựng lên nút gốc cây ứng với khoá đầu tiên (trường hợp tìm kiếm trên cây rỗng). Sau đó, đối với các khoá tiếp theo, tìm trên cây không thấy thì bổ sung vào.

ĐỪNG BÀO CHỮA

Dịch - **Nguyễn Việt Khoa**
Nguồn: Uncle Bob - http://blog.cleancoder.com/uncle-bob/2017/12/18/Excuses.html

Giữa nguyên tắc ghi sổ kép và TDD có sự nhiều tương đồng rõ nét.

- Cả hai đều là những nguyên tắc được sử dụng bởi những chuyên gia khi thao tác thận trọng trên những tài liệu phức tạp với những ký hiệu khó hiểu. Những thao tác đó đòi hỏi phải chính xác hoàn toàn cả về thể loại và vị trí, và bên dưới nỗi đau của những hậu quả khủng khiếp.
- Cả hai đều liên quan đến việc mô tả một chuỗi dài các hành động nhỏ theo hai hình thức khác nhau và trên hai tài liệu khác nhau.
- Cả hai kỹ thuật đều cập nhật những tài liệu bằng một hành động nhỏ tại mỗi thời điểm và mỗi hành động cập nhật đó đều kết thúc bằng một hoạt động kiểm tra để đảm bảo hai tài liệu vẫn trong tình trạng cân đối.

Mô tả điều này cụ thể hơn như sau:

- Kế toán nhập mỗi giao dịch vào hai tài khoản khác nhau. Một tài khoản là Liability (nợ phải trả). Một tài khoản là Asset (các tài sản) hoặc Equity (vốn chủ sở hữu). Những tài khoản này được tổng hợp trên bảng cân đối kế toán (balance sheet), và chúng quan hệ với nhau bởi công thức: Assets + Equities = Liabilities
- Các kế toán được đào tạo để nhập vào một giao dịch mỗi lần, kiểm tra bảng cân đối kế toán sau khi nhập mỗi giao dịch đó.
- Các lập trình viên thực hành TDD thêm một phần nhỏ của hành vi trong hai chương trình khác nhau. Một trong số đó là chương trình kiểm thử. Chương trình còn lại là sản phẩm mong muốn. Cả hai phải thực thi trong một môi trường để minh chứng rằng mã sản phẩm hoạt động giống với mô tả trong mã kiểm thử.
- Các lập trình viên được đào tạo để thêm một phần nhỏ mã nguồn cho mỗi chương trình tại mỗi thời điểm và sau đó thực thi các kiểm thử sau mỗi hành động bổ sung đó.

Như tôi đã trình bày, hai nguyên tắc này là hoàn toàn tương đương. Chúng là những cách tiếp cận gần như giống hệt nhau.

Và không có gì đáng ngạc nhiên. Cả hai nguyên tắc này cùng phục vụ một mục đích giống nhau. Chúng đều cho phép chúng ta bảo trì những tài liệu phức tạp với đầy dẫy những ký hiệu khó hiểu, hoạt động này phải tuyệt đối chính xác để chắc chắn rằng chúng ta sẽ không chịu đau đớn nếu gây ra những hậu quả nghiêm trọng.

Các kế toán có các thời hạn không? Những quản lý có gây áp lực để họ hoàn thành công việc kế toán vào một ngày cụ thể nào không? Dĩ nhiên! Không có sự khác biệt về áp lực. Các kế toán cảm thấy áp lực cũng giống như những lập trình viên.

Những chương trình của lập trình viên có kém quan trọng hơn những tài khoản kế toán không? Tất nhiên là không! Những chương trình đó là những công cụ kiếm tiền hoặc tiết kiệm tiền cho công ty. Chúng rất quan trọng và quan trọng như những tài khoản kế toán vậy.

Vậy tại sao các kế toán có thể duy trì kỷ luật của mình một cách kiên trì và đầy đủ? Bạn có thể tưởng tượng một nhóm các kế toán nói chuyện với nhau rằng: "Này các ông, chúng ta thực sự đang nằm dưới họng súng đấy. Chúng ta phải hoàn tất các tài khoản này vào Thứ Sáu. Vì vậy, hãy hoàn thành các tài khoản Asset và Equity. Rồi quay lại thực hiện với tài khoản Liability sau."

Không. Bạn không thể tưởng tượng ra điều đó. Hoặc, ít nhất, bạn không nên tưởng tượng vậy. Các kế toán có thể đi tù vì làm như thế.

Vậy tại sao nhiều lập trình viên lại than vãn và rên rỉ khi đối mặt với những nguyên tắc của TDD? Tại sao họ trình bày khá nhiều lý do về việc TDD là không thực tế, hoặc không phù hợp, hoặc ...

Bạn có thể tưởng tượng một bà kế toán đưa ra những bào chữa như sau: (Tất cả được lấy từ các bài viết về TDD)

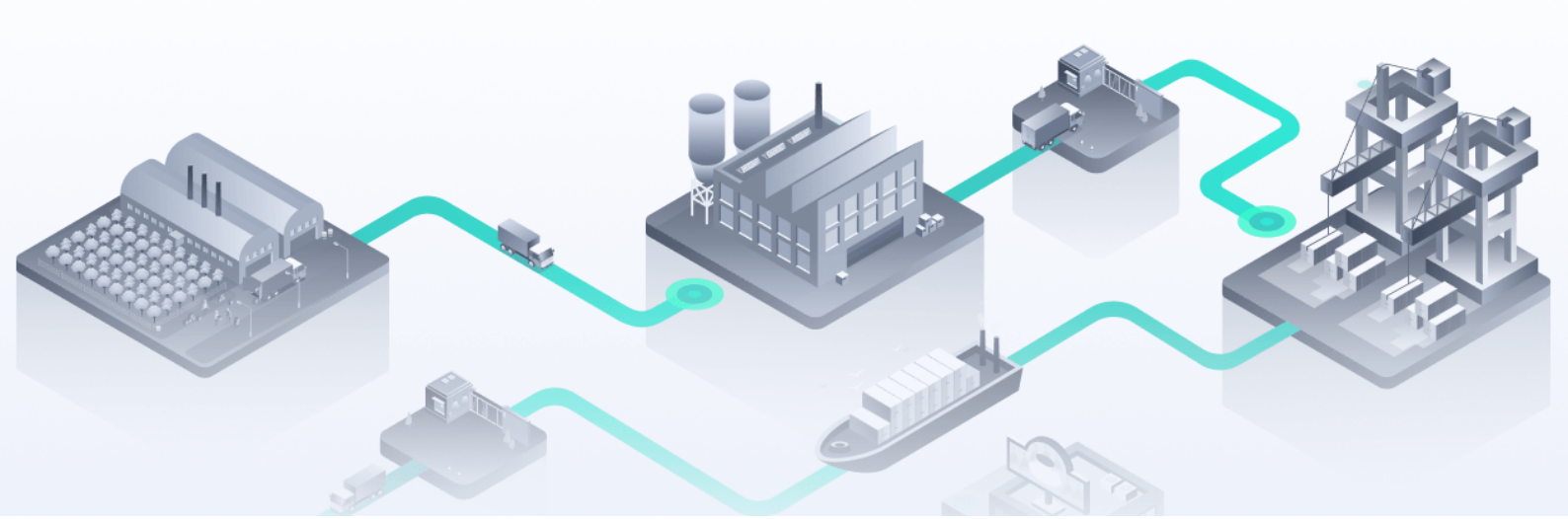
- Tôi không bao giờ ghi sổ kép bởi vì việc theo dõi tất cả những tài khoản đó mất quá nhiều thời gian.
- Các tài khoản không cần phải hoàn hảo, chúng chỉ cần đủ tốt. Ghi sổ kép là quá thừa.
- Tất cả các kế toán thực hiện ghi sổ kép quá sùng bái nó. Họ đang tuân theo một tín điều không cần thiết.
- Cân đối giữa các tài khoản Asset, Equity và Liability thực tế không chứng minh được các tài khoản đó là chính xác. Do đó ghi sổ kép là công việc tốn nhiều công sức mà hiệu quả lại thấp.
- Ghi sổ kép là trò lừa đảo được quảng bá bởi những kế toán đang tìm cách kiếm tiền qua việc bán sách, khóa học và video.
- Ghi sổ kép đã chết. Một tay tư vấn ở Andersen đã viết blog về điều này.
- Tôi chống lại việc ghi sổ kép bởi vì kinh nghiệm cho thấy rằng điều này ngăn không cho thiết kế tài khoản chất lượng cao.
- Tôi đã nghĩ rằng ghi sổ kép là một trò đùa thiết thực phức tạp khi lần đầu nghe về nó. Nó thật là ngớ ngẩn.
- Không dùng được hình thức ghi sổ kép. Tôi không quan tâm những gì bạn đã nghe về nó. Tôi không quan tâm quản lý của bạn mong muốn dùng nó như thế nào. Tôi không quan tâm ánh mắt căng thẳng của đồng nghiệp của bạn lóe lên niềm vui. Nó-Không-Hoạt động.
- Những mục ghi kép dẫn tới thiệt hại trong thiết kế tài khoản.
- Không phải mọi thứ đều có thể chịu trách nhiệm.
- Mục ghi kép kìm hãm thiết kế tài khoản.
- Những mục ghi kép chỉ hay về phương diện lý thuyết. Trong thực tế, nó không cho thấy rõ giá trị tốt đẹp của mình.
- Cuộc sống thì ngắn và một ngày chỉ có số giờ hữu hạn. Do đó chúng ta phải lựa chọn cách sử dụng thời gian của mình. Nếu chúng ta dùng nó để làm các mục ghi kép thì thời gian đó chúng ta chả làm được điều gì cả.
- Sử dụng mục kép không đảm bảo việc bạn sẽ thiết kế ra những tài khoản tốt.
- Tôi sẽ ra ngoài bằng một chân và tuyên bố một cách tàn nhẫn nhưng trung thực rằng theo nghĩa đen thì ghi sổ kép thực sự là một nghi thức lãng phí thời gian.
- Another concern is the debated degree of perfection to which one must do double entry to do it successfully. Một số trong đó nhấn mạnh rằng nếu ghi sổ kép không được thực hiện liên tục bởi những người trong nhóm từ khi bắt đầu dự án, bạn sẽ bị ảnh hưởng.
- Mục ghi kép chồng phụ thuộc vào tội lỗi. Nó khuyến khích sự hiểu biết. Nó có hàng tấn giáo điều và khẩu hiệu.
- Đối với tôi, những kẻ cuồng tín ghi sổ kép là những đồng CAD tôn giáo gõ cánh cửa của tôi, họ cố gắng chứng minh rằng cách làm việc của tôi không thể sửa lại được và con đường duy nhất để cứu dỗi là những mục kép.
- Những điều gây phiền nhiễu cho tôi về ghi sổ kép là có rất nhiều quy tắc hoặc chỉ dẫn.
- Trên dự án mục kép đầu tiên của bạn có hai thiệt hại lớn, thời gian và sự tự do cá nhân.

Tôi có thể tiếp tục thêm nữa. Bạn sẽ nhận thấy rằng không thiếu sự ngớ ngẩn, thông tin sai lệch, và những kẻ phá hoại.

Đối với tôi, điểm mấu chốt đó là sự đơn giản. TDD là nguyên tắc tốt để đảm bảo những tài liệu phức tạp, chứa đầy những ký hiệu khó hiểu, và được làm theo một cách như thế để tránh những hậu quả tiêu cực đáng kể. Tôi không biết nguyên tắc nào khác gần gũi hơn thế.

Ghi sổ kép được phát minh bởi những người Hàn Quốc hàng nghìn năm trước. Phương pháp này được tạo ra độc lập bởi những người Ý cách đây hơn 600 năm. Ghi sổ kép phát triển và lan rộng cùng với chủ nghĩa tư bản và sự thịnh vượng kinh tế. Tuy nhiên sự tiếp nhận nó không phải là không có cản trở và trì hoãn. Những quốc gia cuối cùng đã chuẩn hóa ghi sổ kép trong thế kỷ 20.

Hãy hy vọng sẽ không mất 500 năm để một nguyên tắc về kiểm thử trở thành tiêu chuẩn dành cho những nhà phát triển phần mềm.



DIVE IN LINUX

Phần một: Nhân và Bản phân phối

Nguyễn Bình Sơn

Bạn PHẢI đọc chuỗi bài viết này

Sẽ rất bất hạnh nếu bạn — người sẽ trở thành nhà phát triển phần mềm trong tương lai, mà lại không biết đến sự tồn tại của Linux, hay vẫn cho rằng nó là thứ biết thì rất tốt. Bởi chẳng chóng thì chầy bạn sẽ nhận ra rằng nó nằm trong nhóm những thứ "không biết thì chết".

Linux có ý nghĩa như thế nào?

Trả lời cho câu hỏi này, bạn hãy nghĩ, nếu bạn là kỹ sư sản xuất ô tô, vậy đường xá có ý nghĩa như thế nào với bạn?

Câu trả lời là đường xá hầu như là thứ làm cho công việc của bạn trở nên có ý nghĩa.

Cũng tương tự như việc những mẫu xe của bạn được thiết kế để chạy trên đường, bạn phải biết đặc điểm, luật chơi, các loại đường xá khác nhau và những giới hạn của chúng, gần như luôn luôn khi bạn sản xuất một chiếc xe, bạn tưởng tượng ra nó sẽ chạy trên con đường bạn nghĩ đến.

Đường xá tạo nên ý nghĩa cho ngành xe hơi, nghành vận tải, nghành dịch vụ, hay nói trắng ra là hầu như toàn bộ nghành công nghiệp. Điều đó cũng tương tự với Linux, với vai trò là bá chủ trong phân khúc hệ điều hành máy chủ, Linux là nền tảng hoạt động của hầu hết các dịch vụ. Hầu như không có điều gì bạn làm trên internet mà không đi qua một máy chủ Linux nào đấy.

Thế mà bạn — người sớm thôi sẽ (hay đang?) thiết kế và sản xuất các ứng dụng web — lại không có ý niệm gì về nó.

Chuỗi bài viết này có gì

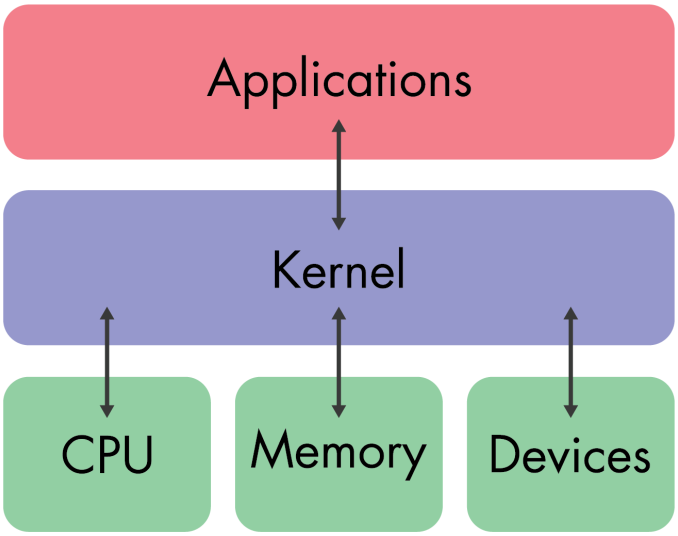
Bạn sẽ không để chuyện đó tiếp diễn, và sẽ ngay lập tức tìm cách cải thiện một cách nghiêm túc hiểu biết và kỹ năng của mình trong lĩnh vực Linux. Kế hoạch sẽ bắt đầu bằng việc tạo ra một danh sách các keywords quan trọng nhất, và bài viết này sẽ giúp bạn bắt đầu. Cụ thể, nó giúp bạn:

- Nâng cấp hiểu biết về khái niệm Linux
- Không gọi Ubuntu là hệ điều hành nữa (nâng cấp hiểu biết về khái niệm "bản phân phối")
- Biết apt là gì (nâng cấp hiểu biết về khái niệm trình quản lý gói trong Linux)
- Có khả năng chọn một cách có chủ ý bản phân phối Linux phù hợp để sử dụng
- Thấu hiệu Hệ Vỏ của Linux
- Thấu hiểu cách sử dụng các command-line của Linux

Trước tiên, trong phần một này, chúng ta sẽ tìm hiểu những khái niệm tổng quan nhất.

Linux là gì?

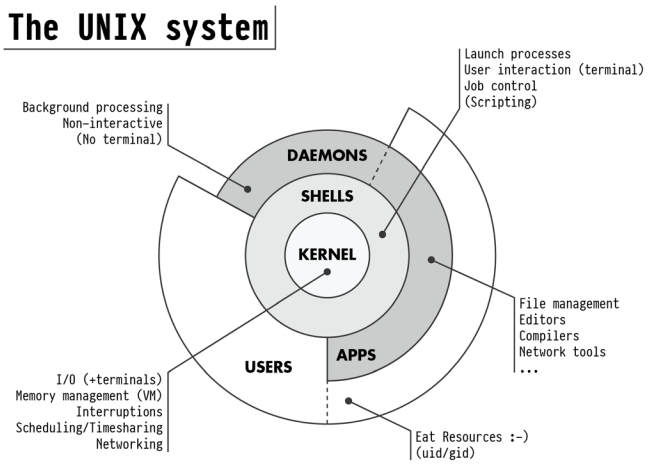
Một nhân hệ điều hành



> Một cách chính xác, ***"Linux là một phần mềm nhân hệ điều hành giống Unix được phân phối tự do."***

Phần mềm, nghĩa rằng Linux là một chương trình máy tính, nó có khả năng hướng dẫn máy tính làm việc.

Nhân hệ điều hành (kernel) ngụ ý rằng bản thân Linux không phải là một hệ điều hành hoàn chỉnh. Nó đóng vai trò làm lớp phần mềm trung tâm trong hệ điều hành, quản lý các tài nguyên của hệ thống, liên lạc các phần mềm ứng dụng với phần cứng.



Nguồn ảnh: https://www.ws.afnog.org/afnog2004/intro-freebsd/00-intro-freebsd/unix-intro/unix-intro3.html

Giống Unix, tính chất này ám chỉ rằng Linux hoạt động "gần như y hệt" với một nhân hệ điều hành Unix. Unix là một hệ điều hành quan trọng trong nền công nghiệp phần mềm, nhiều đặc tả của Unix đã trở thành tiêu chuẩn ngành. Có nhiều nhân hệ hiều hành hành xử như một nhân Unix nhưng trong số đó, Linux là phần mềm nổi tiếng và phổ biến hơn cả.

Việc "giống Unix" là rất có ý nghĩa. Unix mang trong nó một triết lý phát triển hệ điều hành, mang tên là "thiết kế module", mà trong đó hệ điều hành không gì khác hơn là một tập hợp các công cụ cô lập mà mỗi cái lại đảm nhiệm một tính năng giới hạn, ít, nhưng tốt hết mức có thể. Chúng ngầm định với nhau một hệ thống cấu trúc file thống nhất để hoạt động trên đó, và các công cụ Hệ Vỏ (Shell System) có thể kết hợp các công cụ này lại để thực hiện những nhiệm vụ phức tạp. Bản chất module này khiến cho Unix và hầu hết các Unix-like khỏe một cách tự nhiên.

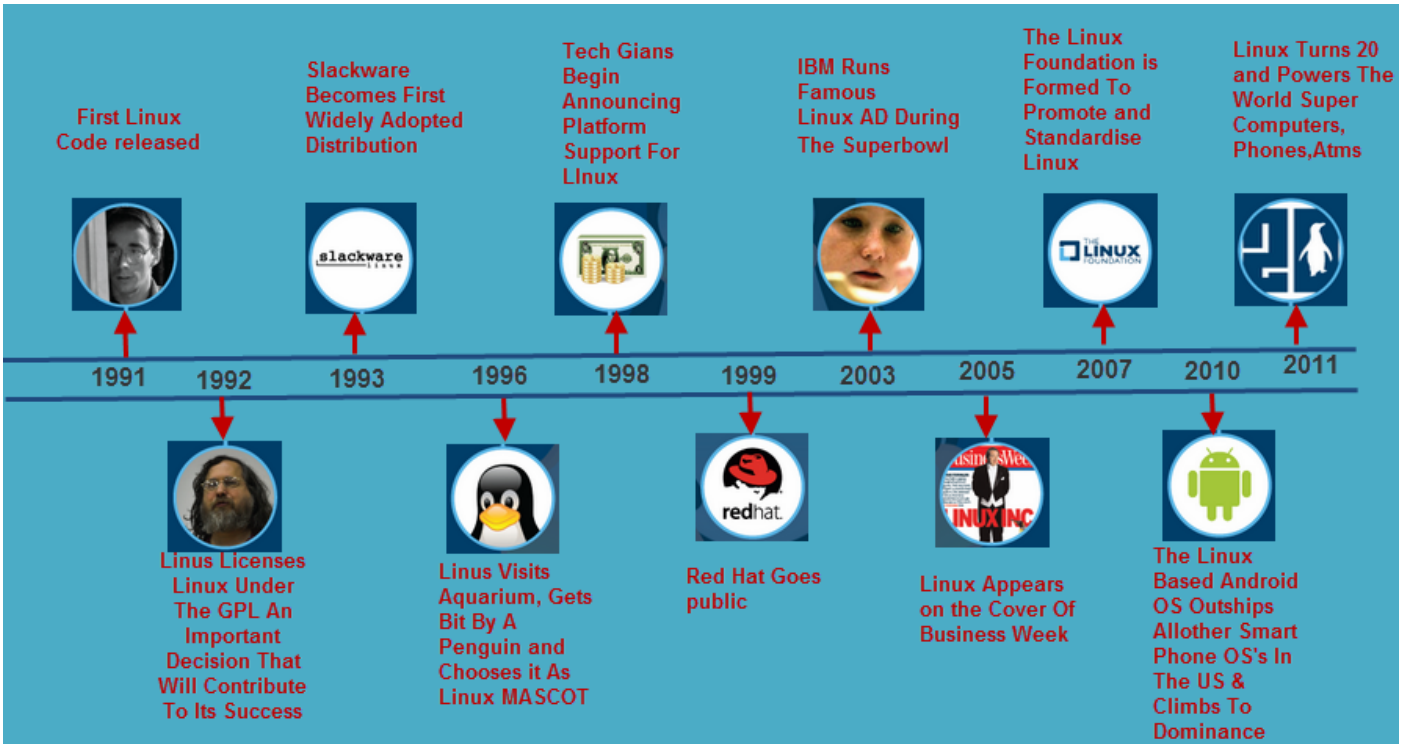
Phân phối tự do, thể hiện bởi giấy phép GNU General Public License(GNU GPL or GPL), nghĩa là mọi người dùng cuối đều có quyền tự do sử dụng, nghiên cứu, chia sẻ, và chỉnh sửa phần mềm Linux. Đây gần như là quyền tự do triệt để nhất trong số các giấp phép phần mềm tự do mã nguồn mở.

Một bản phân phối

Trên thực tế, trong tình huống giao tiếp thường nhật, Linux được sử dụng với ý nghĩa ***"Là một hệ điều hành giống Unix được phân phối tự do, bao gồm một nhân, các công cụ hệ thống, các chương trình và một giao diện người dùng hoàn chỉnh."***

Định nghĩa này nói lên một thực tế là để có thể sử dụng được, nhân Linux cần được kết hợp với những phần mềm hệ thống, các chương trình ứng dụng, và một giao diện để tương tác người-máy (cho dù nó là command-line interface hay graphic interface).

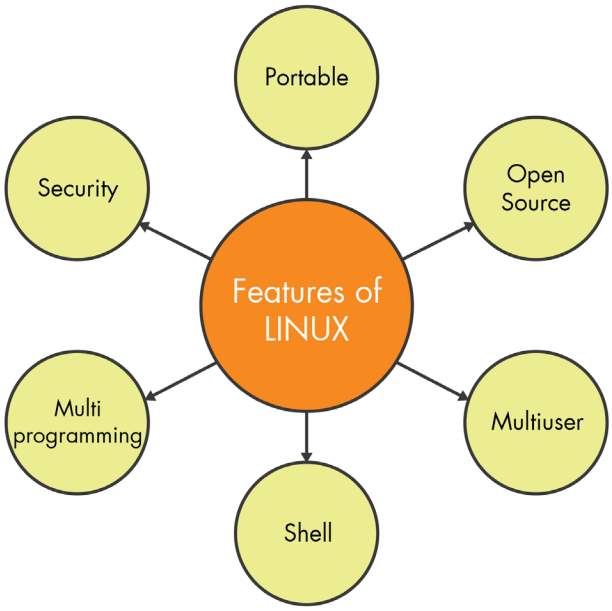
Do việc sử dụng song song cả hai khái niệm có thể gây rối cuộc giao tiếp, nên trong những trường hợp cần rạch ròi (đặcbiệt là khi giao tiếp giữa những người hiểu chuyện với nhau), người ta dùng Linux với định nghĩa đầu tiên, và để ám chỉ một hệ điều hành hoàn chỉnh, họ dùng thuật ngữ bản phân phối (distribution, hay distro). Một ẩn dụ



cho việc nhân Linux bám vào chúng và được phát tán đi khắp nơi.

Lịch sử

Linux được lấy tên từ tác giả của nó, ngài Linus Torvalds. Ông là người Phần Lan và tự đọc tên mình là Li-nú-x, giống như cách phần lớn người Việt Nam đọc. Ông có niềm đam mê mãnh liệt với phần mềm tự do mã nguồn mở, trong số những sản phẩm của ông, Linux và Git là hai phần mềm rất phổ biến và có ảnh hưởng mạnh mẽ trong việc định hình nghành công nghiệp phần mềm hiện đại.

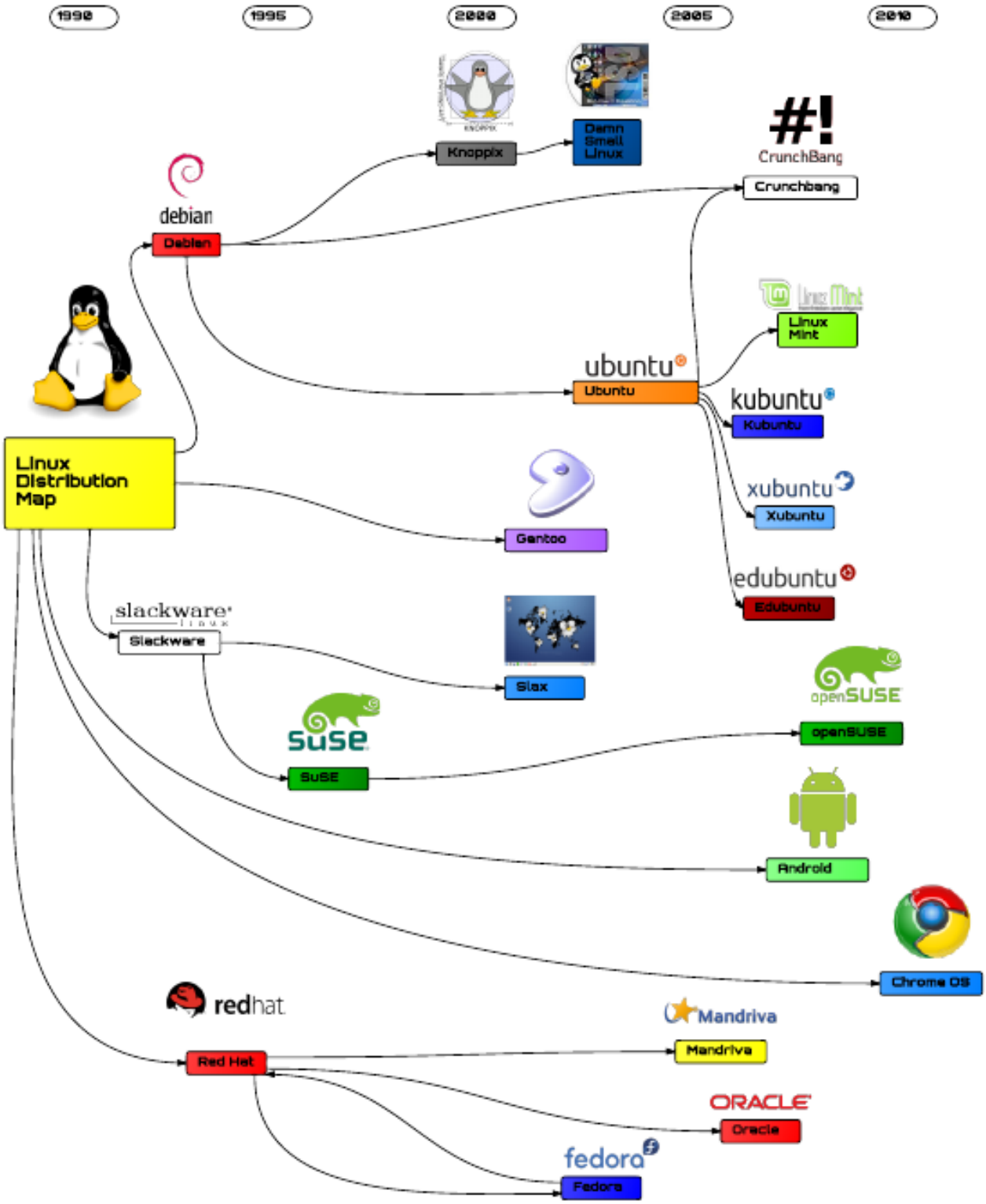


Linus viết nhân Linux khi còn là sinh viên trường đại học Helsinki, như một sản phẩm để vọc (điều được viết trong cuốn sách của ông Just For Fun), với mục tiêu tạo ra một nhân hệ điều hành giống Unix. Và sau đó mời người khác tham gia cùng. Qua thời gian, cộng đồng các nhà phát triển đã phát tán Linux từ một phòng trọ sinh viên trở thành nhân hệ điều hành được sử dụng rộng rãi nhất thế giới.

Cây kế thừa của các bản phân phối

Với kiến trúc module, các bản phân phối Linux rất dễ tùy biến. Bổ sung tính năng bằng cách bổ sung các module, thay thế module này bằng một module khác có tính năng tương tự, hay sử dụng một phiên bản module cũ hơn/mới hơn để cân bằng giữa tối tân và ổn định. Điều đó tạo nên sự đa dạng của các bản phân phối Linux, có vô số các bản phân phối khác nhau, mỗi bản phân phối lại mang trong mình một triết lý thiết kế riêng, với những điểm mạnh riêng. Trong số đó có những cái tên rất nổi tiếng như Ubuntu, Debian, Fedora, Mint, Android, ChromeOS...

Tùy biến một thiết kế có sẵn thành thiết kế mới có sẵn có ưu điểm là tiết kiệm thời gian, và đặc biệt là để kế thừa những điểm mạnh của bản thiết kế trước. Những điểm mạnh này có thể là:



Nguồn ảnh: http://bruessard.org/2017_winner.html

- Trình quản lý gói mạnh
- Tập hợp hợp lý của các công cụ hệ thống
- Một giao diện người dùng thân thiện
- Một tập hợp các phần mềm được thiết kế rất tốt cho một mục đích đặc thù nào đó, ví dụ: để bản phân phối được nhỏ, hay phục vụ giáo dục, hay thiết kế đồ họa, hay bảo mật/ tấn công mạng...

Nếu biểu diễn sự kế thừa của các bản phân phối, chúng ta sẽ được một sơ đồ hình cây như hình trên. Chúng ta sẽ đi vào chi tiết những nhánh chính và những bản phân phối nổi tiếng trong cây kế thừa vào bài viết tiếp theo.

Hẹn gặp lại các bạn!

UNIT TESTING

NHỮNG BƯỚC CHÂN ĐẦU TIÊN

Đặng Huy Hòa

> ***"Hành trình vạn dặm bắt đầu từ một bước chân."***
>
> **Lão Tử**

Giới thiệu

Chất lượng công việc là một trong những yếu tố quan trọng xác định thành công của bạn tại nơi làm việc. Có nhiều cách để làm điều này trong lĩnh vực công nghệ phần mềm. Nhưng có một cách dễ dàng và hiệu quả là áp dụng kiểm thử. Lý do đơn giản là khả năng viết mã đạt chất lượng thường quan trọng hơn nhiều so với việc viết một khối lượng lớn mã khó bảo trì và tồn tại nhiều lỗi.

Unit Test (Kiểm thử đơn vị) là kỹ thuật kiểm thử những khối thành phần nhỏ nhất trong phần mềm (thường là các hàm hoặc phương thức). Đây là một trong những cấp độ kiểm thử đơn giản và có thể bắt đầu sớm trong vòng đời phát triển phần mềm. Thậm chí, bạn có thể viết unit test trước khi viết mã. Tuy nhiên, đây không phải là một thuật ngữ mới trong lĩnh vực phần mềm. Khái niệm unit test xuất hiện lần đầu trong ngôn ngữ lập trình Small-talk vào những năm 1970. Đến nay, unit test gần như đã trở thành một chuẩn mực trong ngành bởi mục đích của nó là phục vụ yêu cầu nâng cao chất lượng sản phẩm phần mềm.

Với nhiều lập trình viên, dù mới vào nghề hay đã gạo cội, thì unit test là một trong những kỹ năng không thể thiếu khi làm việc. Nếu bạn chưa từng nghe qua hoặc chưa có điều kiện thực hành thì cùng bước những bước chân đầu tiên qua bài viết này nhé!

Bài viết này có gì

Với bài viết này, bạn học được những nội dung sau:

- Một số khái niệm cơ bản trong hoạt động testing và unit testing
- Sử dụng JUnit trong dự án Java nói chung
- Kết hợp Mockito và JUnit để thực hiện việc kiểm thử trong một số tình huống thực tế
- Case Study và những bài học khi thực hiện kiểm thử

Lợi ích của Unit Testing

- Tách rời việc kiểm thử với mã nguồn; không cần viết mã vào phương thức `main()` để có thể kiểm tra phương thức có hoạt động đúng đắn hay không.
- Duy trì một bộ kiểm thử liên tục được cập nhật.
- Đảm bảo mã mới không ảnh hưởng và gây lỗi tới những chức năng hiện có (qua việc thực hiện chạy lại toàn bộ bộ test đã viết từ trước).

Thuật ngữ

Để đọc hiểu nội dung hướng dẫn này, bạn cần biết đến một số thuật ngữ thường được sử dụng trong các hoạt động kiểm thử.

Test case

Test case là các trường hợp cần kiểm thử với đầu vào và đầu ra được xác định cụ thể. Một test case thường có hai thành phần dưới đây:

- Expected value: Giá trị mà chúng ta mong đợi khối lệnh trả về
- Actual value: Giá trị thực tế mà khối lệnh trả về

Sau khi thực hiện khối lệnh cần kiểm thử, chúng ta sẽ nhận được `actual value`. Lấy giá trị đó so sánh với `expected value`. Nếu hai giá trị này trùng khớp nhau thì kết quả của test case là `PASS`. Ngược lại, kết quả là `FAIL`.

Application (hoặc Code) Under Test

Application Under Test (AUT)) là thuật ngữ thường được dùng để chỉ đến hệ thống/ứng dụng đang được kiểm thử. Với hoạt động unit test, các đơn vị kiểm thử của chúng ta là những thành phần nhỏ nhất trong hệ thống nên có thể dùng các thuật ngữ khác phù hợp hơn như Code Under Test (CUT).

Mock và Stub

Đây là các thành phần bên ngoài được mô phỏng hoặc giả lập trong ngữ cảnh của hoạt động kiểm thử. Thông thường, để AUT hoạt động đúng chức năng thì sẽ cần đến những thành phần bên ngoài như Web Service, Database,... Ở cấp độ unit test, chúng ta cần phải tách rời các thành phần phụ thuộc này để có thể dễ dàng thực thi test case. Phần này sẽ được giải thích rõ hơn trong mục Sử dụng Mockito (Mocking framework).

> ***Lưu ý***
> Ngoài thuật ngữ mock và stub, thỉnh thoảng bạn sẽ gặp các từ khác như Spy và Fake.

Thiết kế test case

Trong phần này, chúng ta sẽ tìm hiểu các loại test case, cấu trúc thường gặp ở một test case và xem xét một số yếu tố tạo nên một test case tốt. Dựa vào các đặc tính đó, chúng ta sẽ tìm hiểu những nguyên tắc để có thể thiết kế và thực hiện được các test case tốt.

Phân loại test case

1. Positive test case: Là những trường hợp kiểm thử đảm bảo người dùng có thể thực hiện được thao tác với dữ liệu hợp lệ.

2. Negative test case: Là những trường hợp kiểm thử tìm cách gây lỗi cho ứng dụng bằng cách sử dụng các dữ liệu không hợp lệ.

Hãy làm rõ các loại test case trên qua một ví dụ đơn giản như sau. Giả sử, chúng ta đang thiết kế ứng dụng đặt phòng khách sạn và có một yêu cầu là:

Hệ thống cho phép khách hàng có thể đặt phòng mới với thời gian xác định.

Với yêu cầu trên, chúng ta có một số trường hợp cần kiểm thử như sau:

1. Trường hợp positive là đảm bảo có thể thêm phòng với các dữ liệu hợp lệ như mã phòng cần đặt, thời gian hợp lệ, mã khách hàng hợp lệ, giá tiền được tính với số ngày đặt,...
2. Còn các trường hợp negative sẽ cố gắng thực hiện thao tác đặt phòng với những dữ liệu không hợp lệ như:
   - Đặt phòng mới mà không có mã phòng
   - Đặt phòng mới với thời gian không hợp lệ (thời gian ở quá khứ)
   - Đặt phòng mới với mã khách hàng không tồn tại trong cơ sở dữ liệu
   - Đặt phòng mới với giá tiền âm (nhỏ hơn 0).
   - ... và nhiều trường hợp khác

Hy vọng qua ví dụ trên, bạn có thể phân loại được các test case và tự xác định được các test case cho yêu cầu phần mềm mà bạn đang thực hiện.

Cấu trúc một test case

Các trúc mã mà chúng ta nên tuân thủ trong một test case là `cấu trúc AAA`. Cấu trúc này gồm 3 thành phần:

1. Arrange - Chuẩn bị dữ liệu đầu vào và các điều kiện khác để thực thi test case.
2. Act - Thực hiện việc gọi phương thức/hàm với đầu vào đã được chuẩn bị ở `Arrange` và nhận về kết quả thực tế.
3. Assert - So sánh giá trị mong đợi và giá trị thực tế nhận được ở bước `Act`. Kết quả của test case sẽ là một trong hai trạng thái sau:
   - PASS: nếu kết quả mong đợi và kết quả thực tế khớp nhau
   - FAIL: nếu kết quả mong đợi khác với kết quả thực tế

Đôi khi bạn sẽ bắt gặp một số bài viết dùng `từ cấu trúc Given-When-Then`. Về bản chất, cũng chính là `cấu trúc AAA` như trên.

Thành phần cố định (Fixtures)

Là những thành phần được lặp đi lặp lại qua mỗi test case và có thể chia sẻ các thao tác chung giữa các test case. Ví dụ: thiết lập cấu hình hoặc chuẩn bị dữ liệu trước khi bộ test được thực thi, và dọn dẹp bộ nhớ sau khi hoàn thành. Thành phần cố định phải được đặt lên trên cùng của bộ kiểm thử.

Có bốn loại thành phần cố định chính:

Setup

Là thành phần được thực thi trước khi test case thực thi. Trong một số thư viện xUnit (công cụ hỗ trợ viết và thực thi unit test), chúng ta thường gặp những phương thức/hàm, hoặc annotion có tên là `BeforeEach`. Thành phần này chính là Setup.

One-Time Setup

Là thành phần được thực thi đầu tiên (trước cả khi cả setup và test case được thực thi). Trong một số thư viện xUnit (công cụ hỗ trợ viết và thực thi unit test), chúng ta thưsờng gặp những phương thức/hàm, hoặc annotion có tên là BeforeAll. Thành phần này chính là One-Time Setup.

Teardown

Là thành phần được thực thi sau khi `test case` được thực thi. Trong một số thư viện xUnit (công cụ hỗ trợ viết và thực thi unit test), chúng ta thường gặp những phương thức/hàm, hoặc annotion có tên là `AfterEach`. Thành phần này chính là Teardown.

One-Time Teardown

Là thành phần được thực thi sau cùng (sau khi tất cả test case và teardown được thực thi). Trong một số thư viện xUnit (công cụ hỗ trợ viết và thực thi unit test), chúng ta thường gặp những phương thức/hàm, hoặc annotion có tên là AfterAll. Thành phần này chính là One-Time Teardown.

Đặc tính của một unit test tốt

Một ca kiểm thử tốt sẽ có những đặc tính sau:

- Dễ viết - Có thể bao quát được nhiều trường hợp kiểm thử mà không mất quá nhiều công sức.
- Dễ đọc - Có thể mô tả được chính xác hành vi hoặc chức năng được kiểm thử.
- Tự động hoá - Có thể thực thi lặp lại nhiều lần.
- Dễ thực thi và thực thi nhanh.
- Đồng nhất - Luôn trả về cùng kết quả sau mỗi lần chạy (nếu không thay đổi mã nguồn bên trong).
- Cô lập - Có thể thực thi độc lập mà không phụ thuộc vào các thành phần khác trong hệ thống. Bạn có thể tham khảo mục "Sử dụng Mockito" để làm rõ hơn ý này.
- Khi kết quả kiểm thử thất bại (FAILED), có thể dễ dàng tìm ra giá trị mong đợi và nhanh chóng xác định được vấn đề.

Quy ước đặt tên

Tên lớp chứa mã kiểm thử

Tên lớp chứa mã kiểm thử thường sử dụng hậu tố "Tests" sau tên lớp được kiểm thử. Ví dụ: tên lớp là StockService thì tên lớp chứa mã kiểm thử sẽ là StockServiceTests.

Tên phương thức kiểm thử (test case)

Theo nguyên tắc, tên phương thức kiểm thử phải giải thích nhiệm vụ rõ ràng. Có thể tham khảo một số quy ướt đặt tên cho phương thức như sau:

1. Sử dụng từ should.

> ***Ví dụ: FavouriteStocksShouldbeSaved, todayPriceShouldBeShowed.***

```
@Test
public void favouriteStocksShouldbeSaved() {}
```

2. Viết theo mẫu Given [Đầu-Vào] When [Hành-Vi] Then [Kết-Quả-Mong-Đợi].

> ***Ví dụ:***

```
@Test
public void GivenNullUsernameWhenCreateStudent-
ThenShouldThrowException() {}
```

3. Viết theo mẫu when[hành-vi]_then[Kết-quả]

```
@Test
public void whenEnterValidUsernameAndPassword_
thenLoginSuccessfully() {}
```

Gợi ý viết kiểm thử tốt

1. Mỗi test case nên là một phương thức độc lập, có thể thực thi mà không phụ thuộc vào bất kỳ test case nào khác.
2. Thứ tự thực hiện của mỗi test case không nên ảnh hưởng đến kết quả thực thi (mặc dù có thể).
3. Khi phát hiện bug trong chương trình, hãy viết ngay kiểm thử cho trường hợp xảy ra bug đó để có thể kiểm tra lại sau này.
4. Tên phương thức kiểm thử phải rõ ràng. Vì vậy không phải do dự nếu tên phương thức quá dài. Ví dụ TestDivisionWhenNumPositive-DenomNegative tốt hơn DivisionTest3.
5. Hãy kiểm thử những trường hợp ném ra ngoại lệ (nếu có). Ví dụ WhenDivisionByZero-ShouldThrowException.
6. Hãy kiểm thử các trường hợp negative để làm rõ hình thức phản hồi khi đầu vào là dữ liệu không hợp lệ.

Sử dụng JUnit

Hiện nay, JUnit được tích hợp và hỗ trợ ở phần lớn các IDE hiện tại cho Java (như Eclipse, IntelliJ, NetBeans,...). Việc sử dụng JUnit trong các dự án Java không khó. Các bạn có thể tìm hiểu cách cài đặt thư viện cho dự án của mình qua những hướng dẫn trên mạng.

Trong mục này, chúng ta sẽ cùng lượt qua những tính năng được hỗ trợ trong JUnit 5 - phiên bản mới nhất hiện nay.

Ví dụ đầu tiên

Dưới đây là ví dụ giúp bạn có cái nhìn tổng quan về một kiểm thử được viết với JUnit5:

```
import com.codegym.Calculator;
import org.junit.jupiter.api.Test;
import  static  org.junit.jupiter.api.Assertions.
assertEquals;

class CalculatorTests {
    private final Calculator calculator = new Cal-
culator();

    @Test
    void shouldReturn2When1Plus1() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

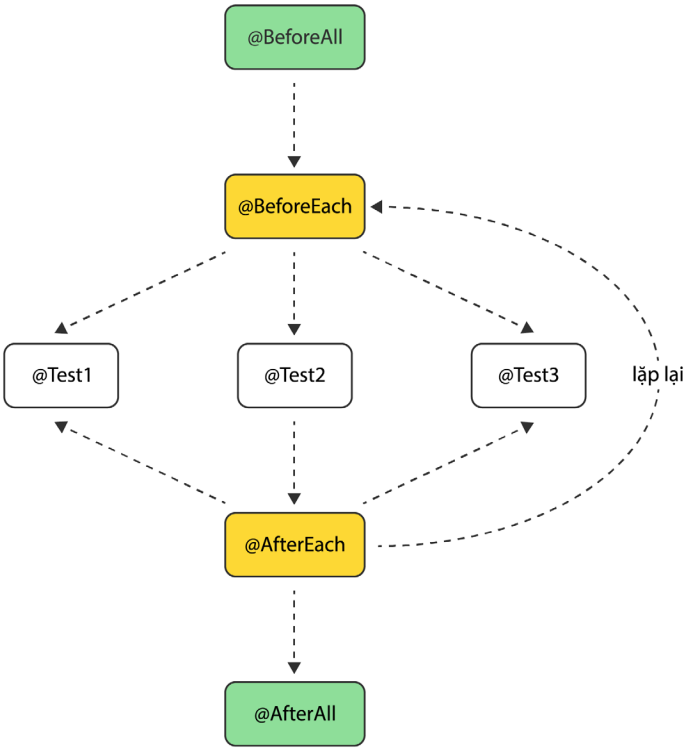
Giải thích ví dụ:

- `@Test` là annotation đánh dấu phương thức `shouldReturn2When1Plus1()` là một test case. Hãy chú ý rằng tên của phương thức test được viết rất rõ là `nên trả về kết quả 2 khi 1 cộng 1.`
- Ở phần thân của phương thức chứa một dòng mã kiểm tra kết quả của phương thức `add()` với đầu vào là hai số có giá trị lần lượt là `1 và 1`.
- So sánh giá trị thực tế trả về của phương `add()` với giá trị mong đợi là `2`.
- Sau khi chạy test case này, kết quả sẽ là `PASS` nếu phương thức `add(1, 1))` trả về kết quả đúng bằng `2`.

Các mục tiếp theo sẽ cung cấp thêm chi tiết về một số tính năng cơ bản được hỗ trợ trong JUnit5.

Các annotation trong JUnit

Sơ đồ dưới đây thể hiện thứ tự thực hiện các phương thức khi được đánh dấu với annotion tương ứng:



Các annotaion @BeforeAll, @BeforeEach,@AfterEach, @AfterAll là những thành phần cố định, thực hiện các chức năng lặp đi lặp lại. Annotation @Test được dùng để xác định một test case.

Assertions

Assertions là lớp chứa các phương thức hỗ trợ đánh giá các điều kiện trong kiểm thử.

So với phiên bản trước, JUnit 5 vẫn giữ các phương thức cũ và thêm một số phương thức mới tận dụng những tích năng của Java 8.

Dưới đây là danh sách những phương thức assertion có trong JUnit5.

AssertTrue và assertFalse

Phương thức `assertTrue` được dùng để kiểm tra kết quả của điều kiện có bằng `true` hay không.

Ví dụ:

```
@Test
public void whenAssertingConditions_thenVeri-
fied() {
    assertTrue(10 > 5, “10 lớn hơn 5”);
}
```

Ngược lại, phương thức `assertFalse` được dùng để kiểm tra kết quả của điều kiện có bằng `false` hay không.

```
@Test
public void whenAssertingConditions_thenVeri-
fied() {
    assertTrue(5 > 10, “5 không lớn hơn 10”);
}
```

AssertEquals và assertNotEquals

Phương thức `assertEquals` được dùng để kiểm tra giá trị mong đợi và thực tế có bằng nhau hay không.

Ví dụ:

```
@Test
public void whenAssertingConditions_thenVeri-
fied() {
     String actual = new String(“CodeGym”).toUp-
perCase();
     String expected = “CODEGYM”;
     assertEquals(expected, actual);
}
```

Ngược lại, Phương thức `assertNotEquals` được dùng để kiểm tra giá trị mong đợi và thực tế có bằng nhau hay không. Chúng ta cập nhật lại ví dụ ở trên:

```
@Test
public void whenAssertingConditions_thenVeri-
fied() {
     // thay thế phương thức toUpperCase() thành
toLowerCase()
     String actual = new String(“CodeGym”).toLow-
erCase();
     String expected = “CODEGYM”;
     assertNotEquals(expected, actual);
}
```

Với trường hợp các giá trị chúng ta đang so sánh thuộc kiểu Object, `assertEquals` và `assertNotEquals` sẽ gọi phương thức equals để so sánh giá trị.

Có một điểm cần lưu ý nếu giá trị là kiểu số thực (`float` hoặc `double`). Trên thực tế, có nhiều trường hợp mà giá trị số thực mong đợi và thực tế có thể chênh lệch với nhau trong khoảng chấp nhận được. Với tình huống này, phương thức `assertEquals` và `assertNotEquals` hỗ trợ tham số thứ ba là delta (bên cạnh `expected` và `actual`). Chúng ta cùng xem qua ví dụ dưới đây:

```
@Test
public void whenAssertingConditions_thenVeri-
fied() {
    float actual = 12 / 3.0001f;
    float expected = 4;
    assertEquals(expected, actual, 0.001f);
}
```

Kết quả của phép chia 12 cho 3.001 thì sẽ là một số thực 3.9998667. Kết quả của ca kiểm thử ví dụ trên là PASSED vì chúng ta đã cho phép mức chênh lệch tối đa là 0.001.

AssertArrayEquals

Phương thức `assertArrayEquals` có thể xác nhận mảng mong đợi và thực tế có bằng nhau hay không. Chúng ta cùng xem xét ví dụ dưới đây:

```
public void whenAssertingArraysEquality_thenE-
qual() {
    char[] expected = { ‘C’, ‘o’, ‘d’, ‘e’, ‘G’,
‘y’, ‘m’ };
    char[] actual = “CodeGym”.toCharArray();

    assertArrayEquals(expected, actual, “Mảng
phải giống nhau”);
}
```

AssertSame và assertNotSame

Khu chúng ta muốn xác nhận giá trị mong đợi và thực tế tham chiếu đến cùng một đối tượng hay không, chúng ta phải dùng `assertSame` hoặc `assertNotSame`:

```
@Test
public void whenAssertingSameObject_thenVeri-
fied() {
    String actual = new String("CodeGym");
    String expected = "CodeGym";

    assertSame(expected, actual);
}
```

Kết quả của phương thức test trên là FAILED vì hai biến `actual` và `expected` đang tham chiếu đến hai đối tượng khác nhau trong bộ nhớ.

Chúng ta nên lưu ý về sự khác nhau giữa `assertSame` và `assertEquals` (đã tìm hiểu ví dụ trước):

- `assertEquals` chỉ quan tâm đến giá trị có bằng nhau không (thông qua phương thức equals) mà không cần biết hai giá trị được so sánh có phải cùng là một đối tượng hay không.
- `assertSame` sẽ trả về PASSED chỉ khi cả hai biến cùng tham chiếu đến một đối tượng.

AssertIterableEquals

`assertIterableEquals` so sánh các giá trị được chứa bên trong hai đối tượng kiểu Iterable. Để trả về kết quả PASSED, hai iterable phải trả về bằng số phần tử, giá trị của các phần tử đó và cả vị trí của các phần tử. Hãy cùng xem ví dụ dưới đây:

```
@Test
public void givenTwoLists_whenAssertingItera-
bles_thenEquals() {
    Iterable<String> al = new Array-
List<>(asList("CodeGym", "Coding", "Bootcamp",
"Java"));
    Iterable<String> ll = new
LinkedList<>(asList("CodeGym", "Coding", "Boot-
camp", "Java"));

    assertIterableEquals(al, ll);
}
```

Kết quả của test case trên là PASSED. Phương thức assertIterableEquals chỉ so sánh giá trị các phần tử bên trong mà không quan tâm đến việc các phần tử này đang được lưu trữ tại hai biến thuộc kiểu khác nhau (ArrayList và LinkedList).

AssertThrows

Để có thể xác nhận được phương thức đang kiểm thử có ném ra một ngoại lệ hay không, chúng ta có thể sử dụng assertThrows. Giả sử chúng ta có một phương thức như sau:

```
static void throwAnException() {
      throw new IllegalArgumentException("Tham
số không hợp lệ");
}
```

Dùng cách thức dưới đây để kiểm tra xem phương thức throwAnExcepiont() có ném ra một ngoại lệ hay không, và ngoại lệ đó có phải là IllegalArgumentException hay không:

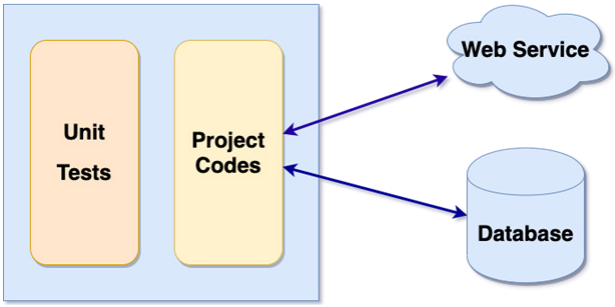
```
@Test
void whenAssertingException_thenThrown() {
    Exception e = assertThrows(IllegalArgument-
Exception.class, () -> throwAnException());
    assertEquals("Tham số không hợp lệ", e.get-
Message());
}
```

Các assertion khác

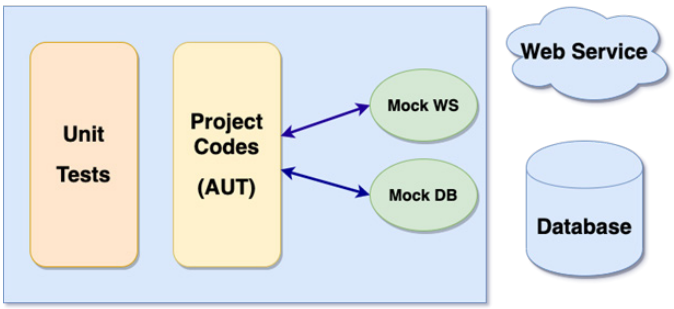
- assertLinesMatch
- fail
- assertNotNull và assertNull
- assertAll
- assertTimeout và assertTimeoutPreemptively

Sử dụng Mockito (Mocking framework)

Khi xây dựng phần mềm, AUT sẽ phụ thuộc vào các thành phần bên ngoài như cơ sở dữ liệu, API, hệ thống file,... Các thành phần phụ thuộc này có thể chưa sẵn sàng hoặc thậm chí chưa tồn tại ở thời điểm chúng ta viết Unit Test. Ngay cả khi những thành phần này đã được chuẩn bị sẵn sàng thì việc thực thi một test case có phụ thuộc sẽ chậm hơn vì phải cần thời gian đợi và tương tác với thành phần bên ngoài.



Cô lập AUT là một trong những kỹ thuật giúp giải quyết vấn đề trên. Và lúc này, chúng ta sẽ phải cần đến các mocking framework (tạm dịch là khung mô phỏng) để giả lập các thành phần bên ngoài, nhờ đó có thể cô lập và kiểm thử AUT dễ dàng hơn. Đối tượng mô phỏng này sẽ không gây phá vỡ cấu trúc mã nguồn khi đối tượng thật được thiết kế và triển khai. Hình dưới đây thể hiện việc tạo hai đối tượng mô phỏng là Mock WS và Mock DB để thay thế sự phụ thuộc vào WebService và Database.



Việc tìm hiểu cách thiết lập và sử dụng các mocking framework này là bước quan trọng giúp mở rộng Unit Test cho các hệ thống lớn và phức tạp. Với lập trì viên Java, Mockito là một công cụ không thể thiếu.

Tạo đối tượng mô phỏng

Phương thức `mock()` cho phép chúng ta tạo đối tượng mô phỏng từ một class hoặc interface. Phương thức này không yêu cầu thêm gì khi sử dụng. Và nó có thể tạo các thuộc tính class mô phỏng hoặc các đối tượng mô phỏng cần dùng trong phương thức. Ví dụ dưới đây thể hiện cách tạo một đối tượng mô phỏng kiểu `UserRepository` (đã được định nghĩa trước):

```
StockRepository stockRepository = mock(Stock-
Repository.class);
```

Mô phỏng hành vi

Sử dụng phương thức `when()` để mô phỏng hành vi của đối tượng. Để xác định kết quả thực hiện, chúng ta có thể sử dụng `thenReturn()` hoặc `thenThrow()`.

- `thenReturn()` trả về kết quả
- `thenThrow()` sẽ ném ra một ngoại lệ

```
when(stockRepository.count()).thenReturn(100);
```

Nếu muốn trả về nhiều kết quả cho nhiều lần gọi, chúng ta sử dụng `thenReturn()` nhiều lần:

```
when(stockRepository.count())
  .thenReturn(50)
  .thenReturn(100)
  .thenReturn(200);

// Kết quả in ra màn hình sẽ là:
System.out.println(stockRepository.count()); //
50
System.out.println(stockRepository.count()); //
100
System.out.println(stockRepository.count()); //
200
```

Kiểm chứng

Chúng ta có thể kiểm tra xem phương thức/ hàm có được gọi hay không qua phương thức `verify()`.

```
verify(stockRepository).count();
```

Mockito hỗ trợ những tham số giúp chúng ta có thể mở rộng khả năng kiểm chứng việc gọi phương thức như:

- Số lần gọi với `times()`
- Thời gian thực hiện với `timeout()`, giúp kiểm chứng thời gian thực hiện thuật toán có đảm bảo yêu cầu

Ví dụ:

```
verify(stockRepository, times(2)).count();
// gọi 2 lần
verify(stockRepository, timeout(10)).count(); //
10 mili giây
```

Kết hợp JUnit

Đây là đoạn mã ví dụ cách kết hợp Mockito và JUnit để viết mã kiểm thử đơn vị:

```
@Test
public void tryMockitoMock() {
  StockRepository stockRepository = mock(Stock-
Repository.class);

            when(stockRepository.count())
     .thenReturn(10);

    long stockCount = stockRepository.count();

    Assertions.assertEquals(10, stockCount);
    verify(stockRepository).count();
}
```

Ở phương thức trên, chúng ta mô phỏng hành vi lấy số lượng user thông qua phương thức count() được định nghĩa trong UserRepository. Khi count() được gọi, đối tượng mô phỏng sẽ trả về kết quả là 111 thay vì phải truy vấn vào cơ sở dữ liệu để lấy thông tin.

Case Study

Dự án mà chúng ta sẽ thực hiện là một trang cửa hàng trực tuyến đơn giản hỗ trợ duyệt danh sách sản phẩm và thông tin chi tiết của từng mặt hàng. Khách hàng có thể chọn sản phẩm và lưu vào giỏ hàng để thanh toán.

Trước khi thực sự viết mã, hãy thiết kế chi tiết bao gồm những interface, class và các phương thức có thể cần để thực hiện được ứng dụng này. Dựa vào bản thiết kế, hãy viết các unit test case cho ứng dụng.

Chặng đường tiếp theo

Cảm ơn bạn đã đồng hành cùng bài viết đến đây. Hy vọng những bước chân đầu tiên này sẽ mang lại nhiều ý nghĩa cho chặng đường học hỏi tiếp theo của bạn. Hãy tìm tòi và thực hành nhiều hơn để có thể làm chủ được kỹ năng Unit Test nói riêng và automation testing nói chung. Tới đây, chúng ta nên làm gì để học và thực hành hiệu quả hơn ở kỹ năng này?

Câu châm ngôn của mình là "Thế giới này thật là rộng lớn.. và có quá nhiều sách để đọc". Nên gợi ý đầu tiên luôn là đọc những đầu sách hay về Unit Test, Test-Driven Development và những chủ đề liên quan. Các bạn xem qua các gợi ý sách và website bên dưới nhé!

Sách nên tham khảo

1. Test Driven Development: By Example - Tác giả: Kent Beck
2. The Art of Unit Testing - Roy Osherove (Các ví dụ trong sách được viết vớt .NET nhưng vẫn có thể tham khảo để phát triển ứng dụng trên Java)

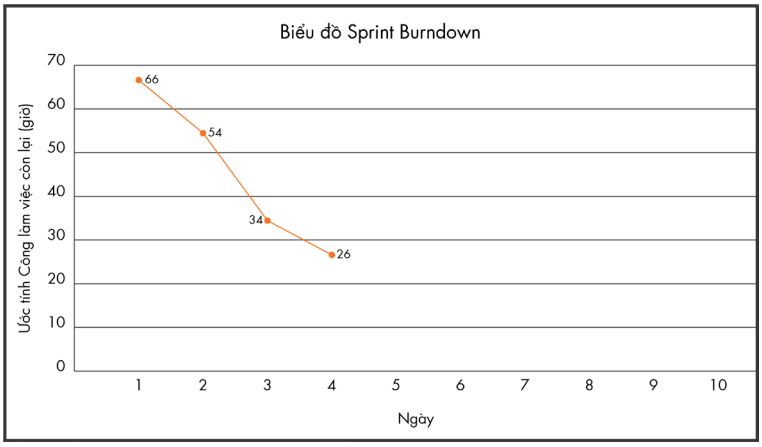
Website nên tham khảo

xUnit Patterns (http://xunitpatterns.com/)

BIỂU ĐỒ SPRINT BURNDOWN CÓ VAI TRÒ GÌ TRONG SCRUM?

Nguyễn Việt Khoa

Biểu đồ Sprint Burndown thể hiện các ước tính cập nhật hàng ngày về khối lượng công việc còn lại tới khi hoàn thành. Biểu đồ Sprint Burndown lấy dữ liệu ước tính từ Sprint Backlog.



Biểu đồ Sprint Burndown giúp nhóm biết được trạng thái công việc hiện tại. Nếu xu hướng công việc đang tiến triển đang chậm hơn so với kỳ vọng, một dự báo rằng Sprint này có thể không thể hoàn thành hết công việc, như vậy Nhóm nên tập trung hoàn thành những nội dung quan trọng nhất để đạt mục tiêu Sprint và thậm chí làm việc với PO. Cũng trong trường hợp chậm này, Nhóm cũng nhìn thấy rằng mình đang gặp một số trở ngại để tiến tới mục tiêu Sprint. Lúc này Nhóm có thể cùng thanh tra để tìm ra nguyên nhân của những trở ngại và hành động thích nghi phù hợp.

Tuy nhiên, biểu đồ này chỉ thể hiện một ước lượng, do đó Nhóm không nên quá tin tưởng vào biểu đồ mà nên liên tục thanh tra và thích nghi dựa vào những công việc thực tế khác.

LẬP TRÌNH VIÊN VÀ CÂU CHUYỆN PHÁT HÀNH SẢN PHẨM

Nguyễn Khắc Nhật

Mười mấy năm trước

Mười mấy năm trước, các phần mềm được phát hành chậm hơn rất nhiều, và khối lượng của các đợt phát hành cũng lớn hơn rất nhiều so với bây giờ.

Chẳng hạn, thời của những hệ điều hành như Windows XP, Windows Vista thì phải mất đến một vài năm chúng ta mới nhận được một phiên bản cập nhật. Và mỗi bản cập nhật đó cũng chứa đựng rất nhiều thay đổi, kể cả về tính năng lẫn giao diện và bảo mật. Hoặc, các phần mềm phổ thông khác như trình duyệt IE, Firefox, các phần mềm nghe nhạc cũng rất lâu mới được cập nhật, thông thường là tính bằng đơn vị nửa năm hoặc một năm.

Năm	*Số lần phát hành*
2001	2 phiên bản
2002	1 phiên bản
2003	1 phiên bản
2004	1 phiên bản
2005	1 phiên bản

Bảng 1: Số lần phát hành rất thưa thớt của trình duyệt IE trước đây.

Có nhiều nguyên nhân để việc phát hành phần mềm ở giai đoạn này diễn ra chậm như vậy, một phần là do tư duy về phát triển sản phẩm truyền thống, một phần khác là do các kỹ thuật và công cụ phát triển phần mềm chưa cho phép các nhóm có thể phát hành sản phẩm một cách thường xuyên, và tất nhiên cũng không thể không kể đến nguyên nhân do trình độ của lập trình viên chưa đáp ứng được.

Ở trong quá khứ, để phát triển mới hoặc nâng cấp một phần mềm, các nhóm phát triển thường dành nhiều thời gian để lập kế hoạch rất chi tiết, sau đó lập trình và rồi kiểm thử lần lượt cẩn thận trước khi tung ra cho người dùng cuối. Với tư duy phát triển sản phẩm như vậy cho nên vòng đời của việc phát triển sản phẩm thường kéo dài.

Các kênh để phát hành sản phẩm trước đây cũng rất khác so với bây giờ, khi mà hầu hết các phần mềm đều được tải về và cài đặt thủ công lên máy của người dùng, khó khăn và bất tiện hơn rất nhiều so với việc phân phối phần mềm dưới dạng dịch vụ như ngày nay.

Mười mấy năm trước, hãy thử tưởng tượng tình huống một phần mềm mới được cập nhật, hằng trăm nghìn, thậm chí là hằng triệu người dùng cần lên website của nhà sản xuất, tải về và tự cài đặt thủ công. Mất rất nhiều thời gian và nỗ lực, sai sót và rủi ro. Chẳng thế mà trong giai đoạn đó, vai trò của những người hỗ trợ kỹ thuật là rất quan trọng, họ là các anh hùng trong mắt những người dùng bình thường, bởi vì họ có khả năng cài đặt và cấu hình các sản phẩm rất bình dân.

Mười mấy năm trước, các kỹ thuật lập trình như kiểm thử tự động, tích hợp liên tục vẫn chưa được sử dụng phổ biến và chưa hiệu quả như bây giờ, do đó rất nhiều thao tác cần phải thực hiện thủ công, mất rất nhiều thời gian và công sức. Do đó thời gian phát triển bị kéo dài cũng là chuyện dễ hiểu.

Dăm năm trước

Dăm năm trước, các phần mềm bắt đầu được phát triển và cập nhật thường xuyên hơn, các bản phát hành cũng thường là nhỏ hơn, đôi khi chỉ là để sửa một vài lỗi nhỏ, hoặc là cải thiện một chút về trải nghiệm người dùng. Việc phát hành sản phẩm theo từng tháng, từng tuần hoặc thậm chí là từng ngày là chuyện không còn xa lạ gì.

Dẫn đầu cho xu hướng phát hành thường xuyên thì phải kể đến trình duyệt Chrome, mặc dù

ra muộn hơn so với các trình duyệt như IE, Firefox hay Opera, nhưng cách phát hành liên tục của Chrome đã khiến cho các trình duyệt khác không theo kịp, và chẳng mấy chốc bị biến thành các gã đối thủ chậm chạp, lỗi thời. Chẳng hạn, trong năm 2010, Chrome có 6 phiên bản phát hành, trong năm 2011 có 8 phiên bản, năm 2012 có 12 phiên bản, năm 2013 có 23 phiên bản... và cứ như vậy.

Năm	*Số lần phát hành*
2010	6 phiên bản
2011	8 phiên bản
2012	12 phiên bản
2013	23 phiên bản
2014	24 phiên bản
2015	24 phiên bản

Bảng 1: Số lần phát hành rất thưa thớt của trình duyệt IE trước đây.

Đặc biệt, với sự thịnh hành của các thiết bị điện thoại thông minh thì tần suất phát hành của các phần mềm trên đó còn trở nên thường xuyên hơn bao giờ hết. Các phần mềm trên điện thoại di động, bao gồm cả ứng dụng lẫn trò chơi, đều được cập nhật rất thường xuyên, có thể theo cả đơn vị ngày hoặc giờ.

Đạt được trình độ phát hành thường xuyên như vậy thì phải kể đến sự dịch chuyển trong tư duy phát triển sản phẩm, sự hỗ trợ về mặt kỹ thuật và trình độ của các nhóm phát triển phần mềm ngày càng tốt lên.

Tư duy phát triển phần mềm ngày càng linh hoạt hơn, hướng đến việc phục vụ người dùng hơn, lắng nghe người dùng hơn, do đó các nhóm đều cố gắng nhanh chóng tung ra sản phẩm và thử nghiệm các tính năng để thu thập được phản hồi của người dùng. Do đó, việc rút ngắn thời gian phát triển và phát hành là yêu cầu bắt buộc đối với các sản phẩm.

Sự phát triển của các công nghệ, công cụ và nền tảng cũng là một yếu tốt rất quan trọng giúp cho các nhóm có thể tăng tốc trong việc phát triển và phát hành sản phẩm. Chẳng hạn, với sự hỗ trợ của các công cụ kiểm thử tự động, giờ đây thay vì mất hằng tuần, hằng tháng để kiểm thử các sản phẩm thì chúng ta chỉ mất vài phút để có thể chạy một loạt các kiểm thử.

Các công cụ tích hợp liên tục và chuyển giao liên tục cũng giúp cho quá trình ổn định sản phẩm nhanh hơn, quá trình cài đặt và cập nhật sản phẩm hoàn toàn được tự động hoá, giúp giảm thiểu sai sót và tiết kiệm được rất nhiều thời gian. Chẳng hạn, với việc sử dụng các công cụ như Docker kết hợp với quy trình chuyển giao liên tục mà một phần mềm có thể được phát hành và cập nhật trên hằng trăm, thậm chí là hằng nghìn máy một cách rất dễ dàng trong khoảng thời gian tính bằng phút.

Bây giờ

Xu hướng phát hành phần mềm dưới dạng dịch vụ (SaaS - Software As A Service) vẫn tiếp tục thể hiện các lợi ích của mình và đang rất phổ biến. Các công cụ và kỹ thuật tự động hoá cho gần như tất cả các công đoạn trong phát triển sản phẩm ngày càng trở nên thông minh hơn, hoàn thiện hơn, giúp cho lập trình viên tập trung sức lực vào xây dựng nghiệp vụ cho phần mềm, thay vì phải mất công sức vào những thao tác kỹ thuật lặp đi lặp lại.

Các hệ thống phần mềm ngày càng trở nên khổng lồ hơn, phức tạp hơn nhưng đồng thời lại yêu cầu sự linh hoạt, dễ thay đổi, dễ mở rộng hơn, do đó quy trình phát triển phần mềm và trình độ của các lập trình viên cũng phải phát triển hơn để đáp ứng được nhu cầu này.

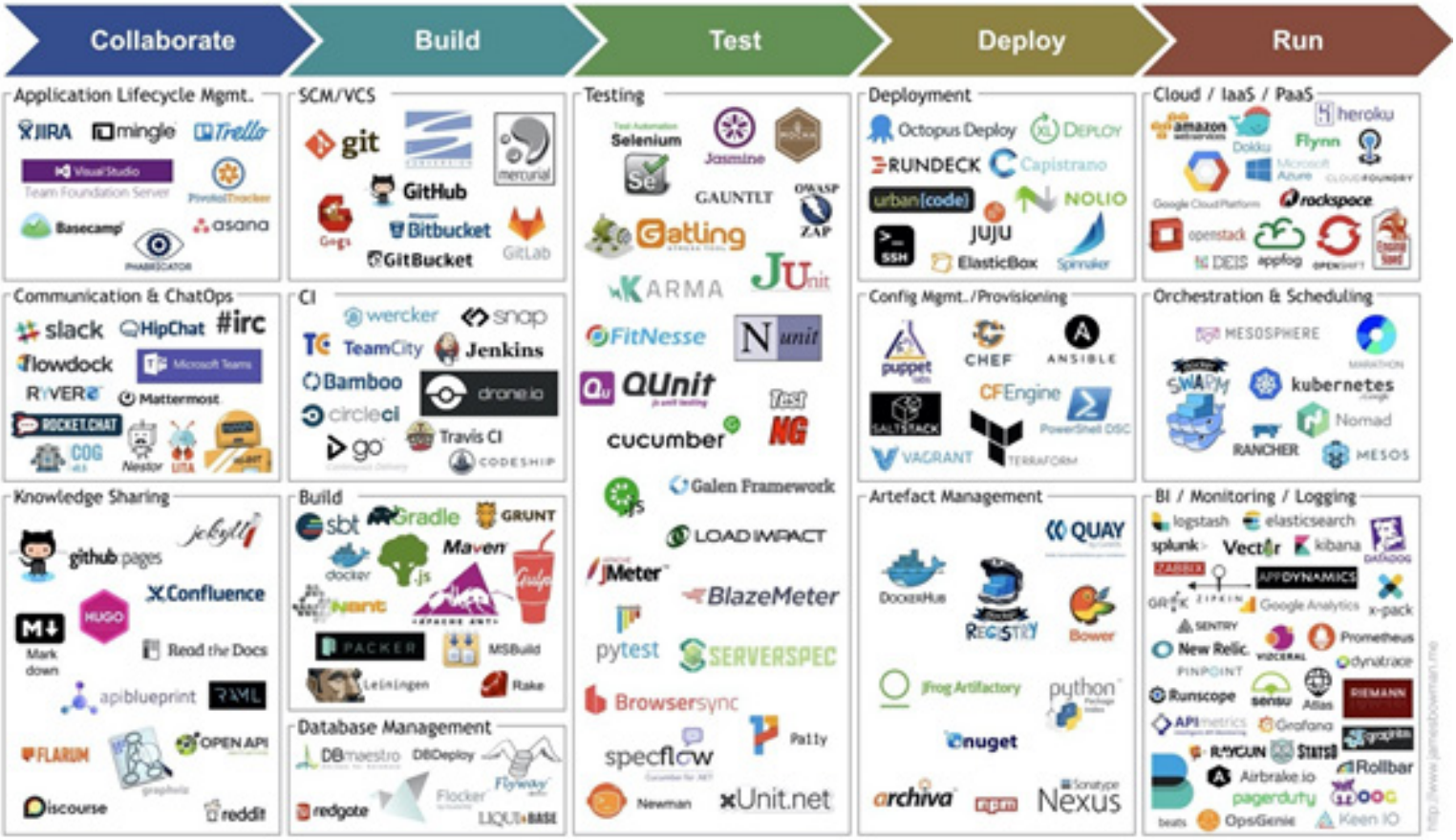
Trước đây, lập trình viên cố gắng để phần mềm "chạy được" là đã coi như hoàn thành nhiệm vụ rồi, nhưng bây giờ thì yêu cầu phải "chạy tốt", rồi thì "dễ mở rộng", "dễ thay đổi", "dễ kết nối". Trước đây, các nhóm phần mềm thường có quy mô nhỏ, việc cộng tác diễn ra đơn giản giữa các thành viên gần gũi nhau, giờ đây, các nhóm phần mềm có thể lên đến vài trăm thậm chí vài nghìn thành viên, phân bố ở rất nhiều nơi trên thế giới, việc cộng tác diễn ra rất chặt chẽ và khắt khe với các tiêu chuẩn cao về mặt chất lượng, do đó trình độ của lập trình viên cũng phải đáp ứng tương xứng.

Lập trình viên hiện đại có rất nhiều công cụ để dùng

Có thể nói, ở hầu hết tất cả các công đoạn của việc phát triển và vận hành phần mềm bây giờ thì các lập trình viên đều được hỗ trợ "đến tận răng", có thể kể đến:

- Các IDE với tính năng thông minh giúp gõ code nhanh hơn
- Các công cụ giúp phân tích độ "sạch" của mã nguồn
- Các công cụ giúp phát hiện các lỗi thường gặp
- Các công cụ giúp kiểm thử tự động ở hầu hết các mức độ như đơn vị, tích hợp, hệ thống, người dùng...
- Các công cụ giúp quản lí mã nguồn một cách tiện lợi
- Các công cụ giúp tích hợp tự động
- Các công cụ giúp cài đặt tự động
- Các công cụ giúp giám sát quá trình hoạt động tự động
- Các công cụ giúp giao tiếp tiện lợi
- ...

Bởi vậy, một lập trình viên hiện đại không chỉ cần biết code thuần tuý, mà còn phải am hiểu tư duy phát triển sản phẩm mới, phương pháp tiếp cận mới trong lập trình và thành thạo sử dụng các công cụ cần thiết trong chuỗi các công cụ ở trên để nhằm mục đích nhanh chóng phát hành sản phẩm và mang đến giá trị cho người dùng sớm.



Tạp chí lập trình - Bạn đường của lập trình viên