**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA KHOA CÔNG NGHỆ PHẦN MỀM**

--------------



**BÁO CÁO CHUYÊN ĐỀ CƠ BẢN SEMINAR VÀ CÁC VẤN ĐỀ HIỆN ĐẠI TRONG CÔNG NGHỆ PHẦN MỀM**

***Đề tài***

**SOFTWARE TESTING**

**Lớp: SE400.K11**

**GVHD: Dương Minh Đức**

**SINH VIÊN THỰC HIỆN:**

* **PHAN VĂN LƯỢM 15520461**
* **ĐOÀN VĂN CHÂU 15520059**

*Tp. Hồ Chí Minh, ngày 17 tháng 12 năm 2019*

Mục lục

[Bảng phân công việc 3](#_Toc27551840)

[Tổng quan về kiểm thử 4](#_Toc27551841)

[8.1 Development testing 12](#_Toc27551842)

[8.1.1 Unit testing 13](#_Toc27551843)

[8.1.2 Choosing unit test cases 15](#_Toc27551844)

[8.1.3 Component testing 20](#_Toc27551845)

[8.1.4 System testing 23](#_Toc27551846)

[8.2 Test-driven development 28](#_Toc27551847)

[8.3 Release testing 29](#_Toc27551848)

[8.3.1 Kiểm thử dựa trên yêu cầu 30](#_Toc27551849)

[8.3.2 Kiểm thử kịch bản 31](#_Toc27551850)

[8.3.3 Kiểm thử hiệu suất 32](#_Toc27551851)

[8.4 User testing 35](#_Toc27551852)

[Cài đặt 41](#_Toc27551853)

[KEY POINTS 44](#_Toc27551854)

[FURTHER READING 45](#_Toc27551855)

# Bảng phân công việc

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Stt | Họ tên | Nội dung công việc | Ngày bắt đầu | Ngày kết thúc | Ghi chú |
| 1 | Phan Văn Lượm | Viết báo cáo và làm slide cho Development testing, User testing. Cài đặt chương trình Demo | 02/08/2019 | 19/11/2019 | Hoàn thành |
| 2 | Đoàn Văn Châu | Test-driven development | 02/08/2019 | 19/11/2019 | Hoàn thành |
| 3 | Nguyễn Duy Hùng | Release testing | 02/08/2019 | 19/11/2019 | Hoàn thành |

# Tổng quan về kiểm thử

Kiểm thử nhằm mục đích cho thấy rằng một chương trình thực hiện những gì nó dự định làm và để khai thác các lỗi chương trình trước khi đưa vào sử dụng. Khi bạn kiểm thử phần mềm, bạn thực hiện một chương trình bằng dữ liệu nhân tạo. Bạn kiểm tra kết quả chạy thử xem có lỗi không, dị thường, hoặc thông tin về các thuộc tính phi chức năng của chương trình. Khi bạn kiểm thử phần mềm, bạn đang cố gắng thực hiện hai điều:

1. Chứng minh cho nhà phát triển và khách hàng rằng phần mềm đáp ứng

yêu cầu. Đối với phần mềm tùy chỉnh, điều này có nghĩa là phải có ít nhất một kiểm tra cho mọi yêu cầu trong tài liệu yêu cầu. Đối với phần mềm chung sản phẩm, điều đó có nghĩa là cần phải có các thử nghiệm cho tất cả các tính năng của hệ thống sẽ được bao gồm trong bản phát hành sản phẩm. Bạn cũng có thể kết hợp các tính năng để kiểm tra các tương tác không mong muốn giữa chúng.

1. Tìm các đầu vào hoặc các chuỗi đầu vào trong đó hành vi của phần mềm không chính xác, không mong muốn, hoặc không phù hợp với đặc điểm kỹ thuật của nó. Đây là do lỗi (lỗi) trong phần mềm. Khi bạn kiểm tra phần mềm để tìm lỗi, bạn đang cố gắng thoát ra khỏi hành vi hệ thống không mong muốn như sự cố hệ thống, tương tác không mong muốn với các hệ thống khác, tính toán không chính xác và hỏng dữ liệu.

Đầu tiên trong số này là kiểm tra xác nhận, nơi bạn mong đợi hệ thống thực hiện sử dụng chính xác một tập hợp các test case phản ánh việc sử dụng dự kiến ​​của hệ thống. Thư hai là kiểm tra lỗi, trong đó các test case được thiết kế để lộ khuyết điểm. Các test case trong kiểm tra lỗi có thể bị che khuất một cách có chủ ý và không cần phản ánh hệ thống như thế nào thường được sử dụng. Tất nhiên, không có ranh giới nhất định giữa hai cách tiếp cận này để thử nghiệm. Trong quá trình kiểm tra xác nhận, bạn sẽ tìm thấy lỗi trong hệ thống; suốt trong kiểm tra lỗi, một số bài kiểm tra sẽ cho thấy chương trình đáp ứng yêu cầu của nó. Hình 8.1 cho thấy sự khác biệt giữa kiểm tra xác nhận và kiểm tra lỗi. Suy nghĩ của hệ thống đang được kiểm thử như một hộp đen. Hệ thống chấp nhận đầu vào từ một tập I và đầu ra trong tập đầu ra **O**. Một số đầu ra sẽ bị lỗi. Đây là các đầu ra trong tập **Oe** được tạo bởi hệ thống để đáp ứng với các đầu vào trong tập **Ie**. Ưu tiên trong kiểm tra lỗi tìm các đầu vào trong tập **Ie** vì chúng cho thấy các vấn đề với hệ thống. Kiểm tra xác nhận bao gồm kiểm tra với các đầu vào chính xác nằm ngoài **Ie**. Những điều này kích thích hệ thống để tạo ra đầu ra chính xác như dự kiến. Việc kiểm tra không thể chứng minh rằng phần mềm không có lỗi hoặc nó sẽ hoạt động đúng quy định trong mọi hoàn cảnh. Luôn luôn có thể là một bài kiểm tra mà bạn đã bỏ qua khi mà có thể khái thác thêm các vấn đề với hệ thống. Edsger Dijkstra, người đóng góp sớm cho sự phát triển của công nghệ phần mềm, đã tuyên bố rằng (Dijkstra 1972):

*“Testing can only show the presence of errors, not their absence”*

Kiểm thử là một phần của quy trình xác minh và xác nhận phần mềm (V & V) rộng hơn. Xác minh và xác nhận không giống nhau, mặc dù chúng thường bị nhầm lẫn. Barry Boehm, người tiên phong về công nghệ phần mềm, đã thể hiện rõ sự khác biệt giữa họ (Boehm 1979):

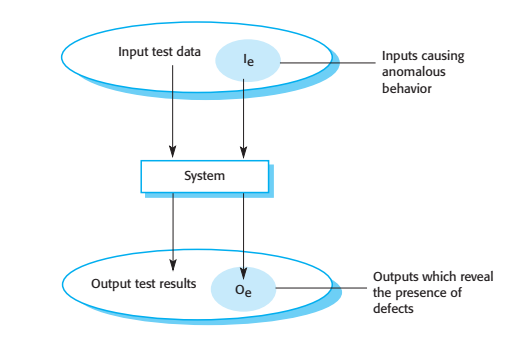


Figure 8.1 An input - output model of program testing

Validation: Are we building the right product?

Verification: Are we building the product right?

Quá trình xác minh và xác nhận có liên quan đến việc kiểm tra phần mềm đó được phát triển đáp ứng đặc điểm kỹ thuật của nó và cung cấp các chức năng mong đợi của những người trả tiền cho phần mềm. Các quy trình kiểm tra này bắt đầu ngay khi các yêu cầu có sẵn và tiếp tục qua tất cả các giai đoạn của quy trình phát triển.

Xác minh phần mềm là quá trình kiểm tra xem phần mềm có đáp ứng được không yêu cầu chức năng và phi chức năng. Xác nhận là một quá trình tổng quát hơn. Mục đích của việc xác thực phần mềm là để đảm bảo rằng phần mềm đáp ứng khách hàng kỳ vọng. Nó vượt ra ngoài việc kiểm tra sự phù hợp với đặc điểm kỹ thuật để chứng minh rằng phần mềm thực hiện những gì khách hàng mong đợi. Xác nhận là cần thiết bởi vì, như tôi đã thảo luận trong Chương 4, các tuyên bố về yêu cầu không luôn phản ánh mong muốn hoặc nhu cầu thực sự của khách hàng và người dùng hệ thống. Mục tiêu của quá trình xác minh và xác nhận là để thiết lập sự tin tưởng rằng Hệ thống phần mềm phù hợp với mục đích của người dùng. Điều này có nghĩa là hệ thống phải tốt đủ cho mục đích sử dụng của nó. Mức độ tin cậy cần thiết phụ thuộc vào mục đích của hệ thống, kỳ vọng của người dùng hệ thống và tiếp thị hiện tại môi trường cho hệ thống:

1. *Mục tiêu của phần mềm* Phần mềm càng quan trọng, thì độ tin cậy của nó càng quan trọng. Ví dụ: mức độ tin cậy cần thiết cho phần mềm được sử dụng để kiểm soát một hệ thống quan trọng về an toàn cao hơn nhiều so với yêu cầu đối với một hệ thống tạo ra các ý tưởng sản phẩm mới.
2. *Sự mong đợi của người dùng* Vì những trải nghiệm trước đây của họ khi gặp lỗi với phần mềm, người dùng đôi khi có kỳ vọng thấp về chất lượng phần mềm. Họ không ngạc nhiên khi phần mềm của họ bị lỗi.

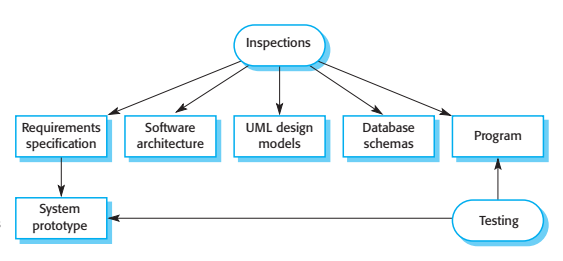


Figure 8.2 Inspections and testing

có thể chịu đựng thất bại vì lợi ích của việc sử dụng lớn hơn chi phí thất bại và hồi phục. Tuy nhiên, khi một sản phẩm phần mềm được thiết lập nhiều hơn, người dùng mong đợi nó trở nên đáng tin cậy hơn. Do đó, hãy kiểm thử kỹ lưỡng hơn phiên bản về sau của hệ thống được yêu cầu.

1. *Môi trường tiếp thị* Khi một công ty phần mềm đưa một hệ thống ra thị trường, nó phải tính đến các sản phẩm cạnh tranh, giá mà khách hàng sẵn sàng để trả tiền cho một hệ thống và lịch trình cần thiết để cung cấp hệ thống đó. Trong một môi trường cạnh tranh, công ty có thể quyết định phát hành một chương trình trước nó đã được kiểm tra và gỡ lỗi hoàn toàn. Nếu một sản phẩm hoặc ứng dụng phần mềm giá rẻ, người dùng có thể sẵn sàng chịu đựng mức độ tin cậy thấp hơn.

Cũng như kiểm thử phần mềm, quá trình xác minh và xác nhận có thể liên quan đến kiểm tra và đánh giá phần mềm. Kiểm tra và đánh giá phân tích và kiểm tra yêu cầu hệ thống, mô hình thiết kế, mã nguồn chương trình và thậm chí đề xuất kiểm tra hệ thống. Đây là những kỹ thuật V & V tĩnh mà bạn không cần thực thi để xác minh nó. Hình 8.2 cho thấy việc kiểm tra và kiểm tra phần mềm hỗ trợ V & V ở các giai đoạn khác nhau trong quy trình phần mềm. Các mũi tên chỉ các giai đoạn trong quá trình mà các kỹ thuật có thể được sử dụng.

Kiểm tra chủ yếu tập trung vào mã nguồn của một hệ thống, nhưng bất kỳ đại diện nào có thể đọc được của phần mềm, chẳng hạn như các yêu cầu của nó hoặc một mô hình thiết kế, có thể là kiểm tra. Khi bạn kiểm tra một hệ thống, bạn sử dụng kiến ​​thức về hệ thống, ứng dụng của nó tên miền và ngôn ngữ lập trình hoặc mô hình hóa để phát hiện lỗi. Kiểm tra phần mềm có ba ưu điểm so với kiểm tra:

1. Trong quá trình thử nghiệm, các lỗi có thể che dấu (ẩn) các lỗi khác. Khi một lỗi không mong muốn ở đầu ra, bạn không bao giờ chắc chắn nếu đầu ra sau này là do một lỗi mới hoặc là tác dụng phụ của lỗi ban đầu. Bởi vì kiểm tra không có liên quan đến việc thực thi hệ thống, bạn không cần phải lo lắng về các tương tác giữa các lỗi. Do đó, nên có một phiên kiểm tra duy nhất có thể khám phá ra nhiều lỗi trong một hệ thống.

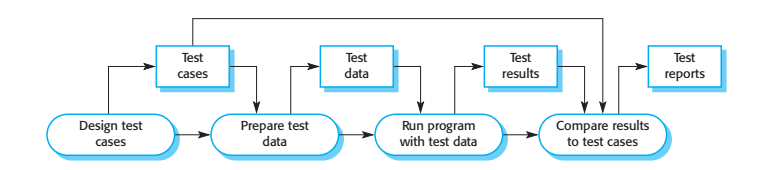


Figure 8.3 A model of the software testing process

1. Các phiên bản chưa hoàn chỉnh của một hệ thống có thể được kiểm tra mà không phải trả thêm chi phí. Nếu như một chương trình chưa hoàn chỉnh, Bạn cần phát triển các thử nghiệm để kiểm tra các bộ phận có sẵn. Điều này làm tăng thêm chi phí phát triển hệ thống.
2. Cũng như tìm kiếm các khiếm khuyết của chương trình, việc kiểm tra cũng có thể xem xét các thuộc tính chất lượng rộng hơn của một chương trình, chẳng hạn như tuân thủ các tiêu chuẩn, tính di động, và khả năng bảo trì. Có thể bạn tìm kiếm những thứ không hiệu quả, không phù hợp thuật toán và phong cách lập trình kém có thể làm cho hệ thống khó bảo trì và cập nhật.

Kiểm tra chương trình là một ý tưởng cũ, và một số nghiên cứu và thí nghiệm cho thấy việc kiểm tra có hiệu quả đối với việc phát hiện lỗi hơn là kiểm tra chương trình. Fagan (Fagan 1976) đã báo cáo rằng hơn 60% lỗi trong một chương trình có thể là do sử dụng kiểm tra chương trình không chính thức. Trong quy trình Cleanroom (Prowell et al. 1999), tuyên bố rằng hơn 90% lỗi có thể được phát hiện trong chương trình kiểm tra.

Tuy nhiên, kiểm tra không thể thay thế kiểm thử phần mềm. Kiểm tra không tốt để phát hiện ra các lỗi phát sinh do các tương tác không mong muốn giữa các phần khác nhau của chương trình, các vấn đề về thời gian hoặc các vấn đề với hiệu năng hệ thống. Trong các công ty nhỏ hoặc các nhóm phát triển, có thể khó khăn và tốn kém để đặt cùng một nhóm kiểm tra riêng vì tất cả các thành viên nhóm tiềm năng cũng có thể nhà phát triển phần mềm.

Hình 8.3 là một mô hình trừu tượng của quá trình kiểm thử truyền thống, được sử dụng trong phát triển plandriven. Các test case như thông số kỹ thuật của các đầu vào và đầu ra dự kiến ​​từ hệ thống (kết quả kiểm tra), cộng với tuyên bố đã được kiểm thử. Dữ liệu kiểm thử là đầu vào được tạo ra để kiểm tra một hệ thống. Dữ liệu kiểm thử đôi khi được tạo tự động, nhưng việc tạo ra test case tự động là không thể. Những người hiểu hệ thống được cho là phải làm gì tham gia chỉ định kết quả kiểm tra dự kiến. Tuy nhiên, thực hiện kiểm thử có thể được tự động. Các kết quả kiểm thử được tự động so sánh với kết quả dự đoán, do đó không cần phải tìm kiếm các lỗi và sự bất thường trong quá trình chạy thử. Thông thường, một hệ thống phần mềm thương mại phải trải qua ba giai đoạn thử nghiệm:

1. *Development testing*, nơi hệ thống được kiểm tra trong quá trình phát triển để phát hiện ra các lỗi. Các nhà thiết kế hệ thống và lập trình viên có khả năng tham gia vào quá trình kiểm thử.
2. *Release testing*, trong đó một nhóm kiểm thử riêng biệt kiểm tra một phiên bản hoàn chỉnh của hệ thống trước khi nó được phát hành cho người dùng. Mục đích của kiểm thử phát hành là để kiểm tra xem hệ thống có đáp ứng các yêu cầu của các bên liên quan của hệ thống không.
3. *User testing*, đối với các sản phẩm phần mềm, người dùng trên mạng có thể là một nhóm tiếp thị nội bộ quyết định xem phần mềm có thể được phát hành hay không. Acceptance testing là một loại kiểm tra người dùng trong đó khách hàng kiểm tra một hệ thống để quyết định xem có nên chấp nhận từ nhà cung cấp hệ thống hay không nếu cần phát triển thêm.

Trong thực tế, quá trình kiểm thử thường bao gồm một hỗn hợp các kiểm thử thủ công và tự động. Trong kiểm thử thủ công, một người kiểm thử chạy chương trình với một số dữ liệu thử nghiệm và so sánh kết quả mong đợi của họ. Họ sẽ báo cáo sự khác biệt cho các nhà phát triển chương trình. Trong kiểm thử tự động, các thử nghiệm được mã hóa trong một chương trình được chạy mỗi khi hệ thống đang được phát triển sẽ được thử nghiệm. Việc này nhanh hơn kiểm thử thủ công, đặc biệt khi liên quan đến kiểm thử hồi quy đã chạy lại các kiểm tra trước đó để kiểm tra xem các thay đổi đối với chương trình chưa đưa ra các lỗi mới.

Thật không may, kiểm tra không bao giờ có thể hoàn toàn tự động vì kiểm thử tự động chỉ có thể kiểm tra xem một chương trình có làm những gì nó phải làm hay không. Thực tế không thể sử dụng thử nghiệm tự động để kiểm tra các hệ thống phụ thuộc vào giao diện của mọi thứ (ví dụ: giao diện người dùng đồ họa) hoặc để kiểm tra rằng chương trình không có tác dụng phụ không lường trước được.

# 8.1 Development testing

|  |
| --- |
| **Debugging**  Gỡ lỗi là quá trình sửa lỗi và các vấn đề đã được phát hiện bằng cách thử nghiệm. Sử dụng thông tin từ các bài kiểm tra chương trình, trình gỡ lỗi sử dụng kiến thức về ngôn ngữ lập trình và kết quả dự định của bài kiểm tra để xác định vị trí và sửa chữa lỗi chương trình. Khi bạn gỡ lỗi một chương trình, bạn thường sử dụng các công cụ tương tác cung cấp thêm thông tin về thực thi chương trình.  **http://software-engineering-book.com/web/debugging/** |

Development testing bao gồm tất cả các hoạt động kiểm thử được thực hiện bởi nhóm phát triển hệ thống. Người kiểm thử phần mềm thường là lập trình viên đã phát triển phần mềm đó. Một số quy trình phát triển sử dụng các cặp lập trình viên / người thử nghiệm (Cusamano và Selby 1998) trong đó mỗi lập trình viên có một người kiểm thử liên quan, người phát triển các kiểm thử và hỗ trợ quá trình kiểm thử. Đối với các hệ thống quan trọng, một quy trình chính thức hơn có thể được sử dụng, với một nhóm thử nghiệm riêng trong nhóm phát triển. Nhóm này chịu trách nhiệm phát triển các bài kiểm tra và duy trì hồ sơ chi tiết về kết quả kiểm tra.

Development testing gồm 3 giai đoạn:

1. *Unit testing*, trong đó các đơn vị chương trình cá nhân hoặc các lớp đối tượng được kiểm tra. *Unit testing* nên tập trung vào kiểm tra chức năng của các đối tượng hoặc phương thức.
2. *Component testing*, trong đó một số đơn vị riêng lẻ được tích hợp để tạo ra các thành phần tổng hợp. *Component testing* nên tập trung vào kiểm tra các giao diện thành phần cung cấp quyền truy cập vào các chức năng thành phần.
3. *System testing*, trong đó một số hoặc tất cả các thành phần trong một hệ thống được tích hợp và toàn bộ hệ thống được kiểm tra. *System testing* nên tập trung vào kiểm tra các tương tác thành phần.

Development testing chủ yếu là quá trình kiểm tra lỗi, trong đó mục đích của kiểm thử là phát hiện ra các lỗi trong phần mềm. Do đó, nó thường được xen kẽ với việc gỡ lỗi cho quá trình định vị các vấn đề với mã và thay đổi chương trình để khắc phục các vấn đề này.

## 8.1.1 Unit testing

Unit testing là quá trình kiểm tra các thành phần chương trình, như các hàm (Function), thủ tục (Procedure), các lớp đối tượng hoặc các phương thức (Method). Các hàm hoặc phương thức riêng lẻ là những thành phần đơn giản nhất. Các chương trình này nên được kiểm thử với các tham số đầu vào khác nhau.

Ví dụ, cho đối tượng trạm thời tiết từ ví dụ trong Chương 7. Các thuộc tính và toán tử của đối tượng này được hiển thị trong Hình 8.4 như sau:

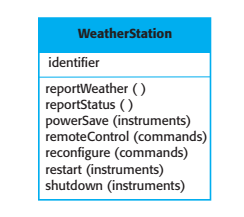


Figure 8.4 The weather station object interface

Nó có một thuộc tính định danh duy nhất. Đây là một hằng số được khai báo khi trạm thời tiết được thiết lập. Do đó, bạn chỉ cần kiểm thử để kiểm tra xem nó đã được thiết lập đúng chưa. Bạn cần định nghĩa các test case cho tất cả các phương thức liên quan với các đối tượng như **reportWeather** và **reportStatus.** Bạn nên kiểm tra phương phức độc lập, nhưng trong một vài trường hợp test sequence cũng rất cần thiết. Ví dụ, để kiểm tra phương thức shutdown các thiết bị trạm thời tiết (**Shutdown**), bạn cần phải thực hiện phương thức **restart**.

Khái quát hóa hoặc kế thừa làm cho việc kiểm thử lớp đối tượng phức tạp hơn. Các phép toán được kế thừa có thể đưa ra các giả định về các phép toán và thuộc tính khác. Các giả định này có thể không hợp lệ trong một số lớp con kế thừa các phép toán. Do đó, bạn phải kiểm tra các phép toán được kế thừa ở mọi nơi mà nó được sử dụng.

Để kiểm tra trạng thái của trạm thời tiết, bạn có thể sử dụng mô hình trạng thái được đề cập trong Chương 7 (Hình 7.8). Việc sử dụng mô hình này, bạn xác định các chuỗi chuyển tiếp trạng thái được kiểm tra và xác định chuỗi sự kiện để ràng buộc các chuyển đổi này. Về nguyên tắc, bạn nên kiểm tra mọi trình tự chuyển đổi trạng thái, mặc dù trong thực tế, điều này có thể quá đắt. Ví dụ về trình tự trạng thái cần được kiểm tra trong trạm thời tiết bao gồm:

Shutdown → Running → Shutdown  
Configuring → Running → Testing → Transmitting → Running  
Running → Collecting → Running → Summarizing → Transmitting → Running

Bất cứ khi nào có thể, bạn nên tự động hóa unit testing. Trong tự động hóa unit testing, bạn sử dụng framework kiểm thử tự động, chẳng hạn như JUnit (Tahchiev et al. 2010) để viết và chạy kiểm thử chương trình của bạn. Unit testing framework cung cấp các lớp kiểm tra chung mà bạn có thể mở rộng để tạo các test case cụ thể. Sau đó, họ có thể chạy tất cả các kiểm thử mà bạn đã triển khai và báo cáo, thường thông qua một số giao diện đơn vị đồ họa (GUI). Toàn bộ bộ kiểm tra thường có thể chạy trong vài giây, vì vậy nó có thể thực hiện tất cả các kiểm tra mỗi khi bạn thực hiện thay đổi chương trình.

Kiểm thử tự động gồm 3 phần:

1. *Setup part,* thiết lập hệ thống với các test case cụ thể là các đầu vào và đầu ra như mong đợi.
2. *Call part,* gọi các đối tượng hay phương thức được kiểm tra.
3. *Assertion part,* bạn so sánh kết quả được gọi với kết quả dự kiến. Nếu khẳng định đánh giá là đúng, kiểm thử đã thành công; nếu sai thì đã thất bại.

Đôi khi, đối tượng mà bạn đang kiểm tra có sự phụ thuộc vào các đối tượng khác có thể chưa được triển khai hoặc việc sử dụng làm chậm quá trình kiểm thử. Ví dụ: nếu một đối tượng gọi một cơ sở dữ liệu, điều này có thể làm chậm quá trình thiết lập trước khi có thể sử dụng nó. Trong những trường hợp như vậy, bạn có thể quyết định sử dụng các đối tượng giả.

Các đối tượng giả là các đối tượng có cùng giao diện với các đối tượng bên ngoài đang được sử dụng mô phỏng chức năng của nó. Ví dụ, một đối tượng giả mô phỏng cơ sở dữ liệu có thể chỉ có một vài mục dữ liệu được tổ chức trong một mảng. Chúng có thể được truy cập nhanh chóng mà không cần chi phí gọi cơ sở dữ liệu và truy cập đĩa. Tương tự, các đối tượng giả có thể được sử dụng để mô phỏng các hoạt động bất thường hoặc các sự kiện hiếm gặp. Ví dụ: nếu hệ thống của bạn dự định thực hiện hành động vào những thời điểm nhất định trong ngày, đối tượng giả của bạn chỉ có thể trả về những thời điểm đó bất kể thời gian thực.

## 8.1.2 Choosing unit test cases

Kiểm thử tốn nhiều thời gian và tiền bạc, vì thế mà bạn phải chọn một test case một cách hiệu quả. Cái hiệu quả đó thì lại được hiểu theo hai hướng:

1. Các test case sẽ cho bạn thấy rằng các thành phần mà bạn kiểm thử đúng như mục đích của nó thì nó sẽ được sử dụng đúng như bạn mong đợi.
2. Các test case sẽ cho thấy được những khiếm khuyết trong các thành phần.

Bạn nên thiết kế hai kiểu test case. Test case thứ nhất, nó sẽ cho thấy các thành phần của một chương trình hoạt động. VD: Nếu bạn đang kiểm tra việc một thành phần khởi tạo một hồ sơ bệnh nhân mới thì test case của bạn sẽ cho thấy rằng sự tồn tại của bản ghi đó trong cơ sở dữ liệu và các thuộc tính của nó đã được chỉ định sẵn. Test case còn lại, sẽ dựa vào kinh nghiệm kiểm thử về những vấn đề được phát sinh. Nó sẽ sử dụng các đầu vào bất thường để kiểm tra xem những cái này được xử lý đúng cách và không làm hỏng các thành phần.

Hai chiến lược giúp bạn chọn test case hiệu quả:

1. *Partition testing,* xác định các nhóm đầu vào có các đặc điểm chung và nên được xử lý theo cùng một cách. Bạn nên chọn các bài kiểm tra trong mỗi nhóm này.
2. *Guideline-based testing,* bạn sử dụng các testing guideline để chọn các test case. Các chỉ dẫn này phản ánh kinh nghiệm trước đây về các lỗi mà các lập trình viên thường mắc phải khi phát triển các thành phần.

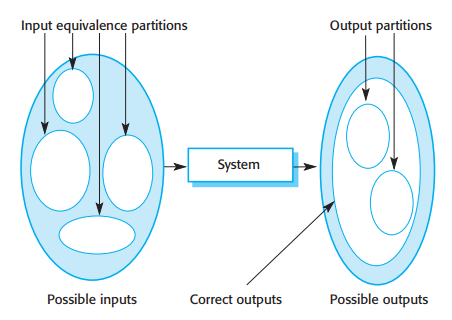


Figure 8.5 Equivalence partitioning

Dữ liệu đầu vào và kết quả đầu ra của một chương trình có thể được coi là thành viên của các tập hợp có đặc điểm chung. Ví dụ về các tập này là số dương, số âm và menu selections. Các chương trình thường hoạt động theo cách so sánh tất cả các thành viên của một tập hợp. Nghĩa là, nếu bạn kiểm tra một chương trình tính toán và yêu cầu hai số dương, bạn mong đợi chương trình sẽ hoạt động cùng một cách cho tất cả các số dương.

Do hành vi tương đương này, các lớp này đôi khi được gọi là đẳng thức equivalence partitions hoặc tên miền (Bezier 1990). Một cách tiếp cận có hệ thống đối với test case thiết kế dựa trên việc xác định tất cả các phân vùng đầu vào và đầu ra cho một hệ thống hoặc thành phần. Test case được thiết kế sao cho đầu vào hoặc đầu ra nằm trong các phân vùng này. Partition testing có thể được sử dụng để thiết kế các test case cho cả hệ thống và thành phần. Trong hình 8.5, hình ellipse ở bên trái đại diện cho tập hợp tất cả đầu vào khả thi cho chương trình đang được kiểm thử. Các hình ellipse bên phải đại diện equivalence partitions. Một chương trình đang được kiểm thử nên xử lý tất cả các thành viên của một equivalence partitions đầu vào theo cùng một cách. Equivalence partitions đầu ra là các phân vùng trong đó tất cả các đầu ra có một vài điểm chung. Đôi khi có ánh xạ 1:1 giữa đầu vào và đầu ra equivalence partitions. Tuy nhiên, đây không phải là luôn luôn như vậy; có thể bạn cần xác định một equivalence partitions đầu vào riêng biệt, trong đó đặc điểm chung duy nhất của đầu vào là chúng tạo đầu ra trong cùng một phân vùng đầu ra. Khu vực bóng mờ trong hình ellipse bên trái đại diện cho đầu vào không hợp lệ. Khu vực bóng mờ ở bên phải hình ellipse đại diện cho các ngoại lệ có thể xảy ra, nghĩa là, phản ứng với các đầu vào không hợp lệ. Khi bạn đã xác định một tập hợp các phân vùng, bạn chọn các test case từ mỗi những phân vùng này. Một nguyên tắc nhỏ để lựa chọn test case là chọn test case trên các ranh giới của các phân vùng, cộng với các trường hợp gần với điểm giữa của phân vùng. Lý do cho điều này là các nhà thiết kế và lập trình viên có xu hướng xem xét các giá trị tiêu biểu của đầu vào khi phát triển một hệ thống. Bạn kiểm tra những điều này bằng cách chọn điểm giữa của vách ngăn. Các giá trị biên thường không điển hình (ví dụ: 0 có thể biểu hiện khác với các số không âm khác) và do đó đôi khi bị các nhà phát triển bỏ qua. Lỗi chương trình thường xảy ra khi xử lý các giá trị không điển hình.

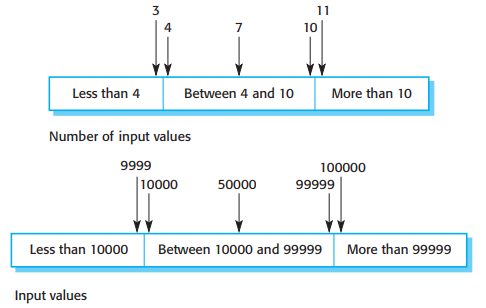


Figure 8.6 Equivalence partitions

Bạn xác định phân vùng bằng cách đặt tả chương trình hoặc tài liệu người dùng. VD: Giả sử đặc tả chương trình thì chỉ chấp nhận từ bốn đến tám đầu vào là các số nguyên có năm chữ số lớn hơn 10.000. Bạn sử dụng thông tin này để xác định các phân vùng đầu vào và các giá trị có thể kiểm thử như bạn có thể thấy ở hình 8.6 ở trên. Khi bạn sử dụng thông số kỹ thuật của một hệ thống để xác định các phần vùng tương đương (equivalence partitions), đây được gọi là kiểm thử hộp đen (black-box testing). Bạn không cần các kiến thức về cách thức hoạt động của hệ thống.

Đôi khi hữu ích khi bổ sung kiểm thử hộp bằng kiểm thử hộp trắng (white-box testing) mà bạn có thể tìm thấy các kiểm thử khả thi khác với mã nguồn của chương trình. VD: Mã nguồn của bạn có thể gồm các ngoại lệ để xử lý các đầu vào không chính xác. Bạn có thể sử dụng kiến thức này để xác định “phân vùng ngoại lệ” phạm vi khác nhau mà việc xử lý ngoại lệ tương tự cần được áp dụng.

Phân vùng tương đương là một cách tiếp cận hiệu quả để kiểm tra vì nó giúp giải quyết các lỗi mà các lập trình viên thường mắc phải khi xử lý các đầu vào ở các cạnh của phân vùng. Bạn cũng có thể sử dụng testing guideline để giúp chọn test case. Nguyên tắc gói gọn kiến thức về loại test case nào có hiệu quả để phát hiện ra lỗi. Ví dụ: khi bạn đang kiểm tra các chương trình với trình tự, mảng hoặc danh sách, các nguyên tắc có thể giúp phát hiện lỗi bao gồm:

1. Kiểm tra phần mềm với các chuỗi chỉ có một giá trị duy nhất. Các lập trình viên tự nhiên nghĩ về các chuỗi được tạo thành từ một số giá trị, và đôi khi họ đưa giả định này vào chương trình của họ. Do đó, nếu được trình bày với một chuỗi giá trị đơn, một chương trình có thể không hoạt động đúng.
2. Sử dụng các trình tự khác nhau có kích thước khác nhau trong các kiểm thử khác nhau. Điều này làm giảm khả năng một chương trình có lỗi sẽ vô tình tạo ra một đầu ra chính xác do một số đặc điểm ngẫu nhiên của đầu vào.
3. Derive test sao cho các phần tử đầu tiên, giữa và cuối của chuỗi được truy cập. Cách tiếp cận này cho thấy các vấn đề tại ranh giới phân vùng.

|  |
| --- |
| **Path testing**  Kiểm tra đường dẫn là một chiến lược thử nghiệm nhằm thực hiện mọi đường dẫn thực thi độc lập thông qua một thành phần hoặc chương trình. Nếu mọi đường dẫn độc lập được thực thi, thì tất cả các câu lệnh trong thành phần phải được thực hiện ít nhất một lần. Tất cả các tuyên bố điều kiện được kiểm tra cho cả trường hợp đúng và sai. Trong quy trình phát triển hướng đối tượng, kiểm tra đường dẫn có thể được sử dụng để kiểm tra các phương thức liên quan đến các đối tượng.  **http://software-engineering-book.com/web/path-testing/** |

Cuốn sách Whittaker của (Whittaker 2009) bao gồm nhiều ví dụ về các hướng dẫn mà có thể được sử dụng trong thiết kế test case. Một số hướng dẫn chung nhất mà ông gợi ý là:

* Chọn đầu vào buộc hệ thống tạo ra tất cả các thông báo lỗi:
* Thiết kế các đầu vào khiến bộ đệm đầu vào bị tràn.
* Lặp lại cùng một đầu vào hoặc một loạt các đầu vào nhiều lần.
* Buộc đầu ra không hợp lệ được tạo ra.
* Kết quả tính toán lực quá lớn hoặc quá nhỏ.

Khi bạn có kinh nghiệm về kiểm thử, bạn có thể phát triển các hướng dẫn của riêng mình về cách chọn test case hiệu quả. Tôi đưa ra nhiều ví dụ về các hướng dẫn kiểm tra trong phần tiếp theo.

## 8.1.3 Component testing

Các thành phần trong phần mềm thường được tạo thành từ một số đối tượng tương tác. Ví dụ, trong hệ thống trạm thời tiết, thành phần cấu hình lại bao gồm các đối tượng xử lý từng khía cạnh của cấu hình lại. Bạn truy cập các chức năng của những các đối tượng thông qua các giao diện thành phần (xem Chương 7). Do đó, việc kiểm tra các thành phần hỗn hợp nên tập trung vào việc hiển thị giao diện thành phần hoặc giao diện hoạt động theo đặc điểm kỹ thuật của nó. Bạn có thể giả định rằng các unit test trên các đối tượng riêng lẻ trong thành phần đã được hoàn thành.

Hình 8.7 minh họa ý tưởng kiểm thử giao diện thành phần. Giả sử rằng các thành phần A, B và C đã được tích hợp để tạo ra một thành phần hoặc hệ thống con lớn hơn. Các trường hợp kiểm thử không được áp dụng cho các thành phần riêng lẻ mà thay vào đó là giao diện của thành phần hỗn hợp được tạo bằng cách kết hợp các thành phần này. Lỗi giao diện trong thành phần hỗn hợp có thể không phát hiện được bằng cách kiểm tra các đối tượng riêng lẻ vì các lỗi này là do tương tác giữa các đối tượng trong thành phần.

Có các loại giao diện khác nhau giữa các thành phần chương trình và do đó, các loại lỗi giao diện khác nhau có thể xảy ra:

1. *Parameter interfaces,* Đây là những giao diện trong đó dữ liệu hoặc là đôi khi tham chiếu chức năng được truyền từ thành phần này sang thành phần khác. Các phương thức trong một đối tượng có parameter interface.

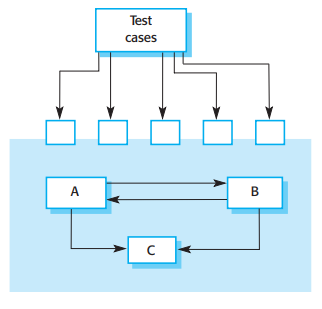


Figure 8.7 interface testing

1. *Shared memory interfaces*, Đây là các giao diện trong đó một khối bộ nhớ được chia sẻ giữa các thành phần. Dữ liệu được đặt trong bộ nhớ bởi một hệ thống con và được các hệ thống con khác lấy từ đó. Loại giao diện này được sử dụng trong các hệ thống nhúng, trong đó các cảm biến tạo dữ liệu được lấy và xử lý bởi các thành phần hệ thống khác.
2. *Procedural interfaces,* Đây là các giao diện trong đó một thành phần đóng gói một tập các thủ tục có thể được gọi bởi các thành phần khác. Đối tượng và các thành phần tái sử dụng có hình thức giao diện này.
3. *Message passing interfaces,* Đây là các giao diện trong đó một thành phần yêu cầu dịch vụ từ thành phần khác bằng cách gửi tin nhắn đến nó. Một thông báo trả lại bao gồm kết quả thực hiện dịch vụ. Một số hệ thống hướng đối tượng có dạng giao diện này, cũng như các hệ thống client-server.

Lỗi giao diện là một trong những dạng lỗi phổ biến nhất trong các hệ thống phức tạp (Lutz 1993). Những lỗi này thuộc ba lớp:

* *Interface misuse*, thành phần gọi đến một số thành phần khác và gây ra lỗi trong việc sử dụng giao diện của nó. Loại lỗi này là phổ biến trong các giao diện tham số, trong đó các tham số có thể sai loại hoặc được truyền sai thứ tự hoặc số tham số sai có thể được thông qua.
* *Interface misunderstanding*, thành phần hiểu sai đặc điểm kỹ thuật của giao diện của thành phần được gọi và đưa ra các giả định về hành vi của nó. Thành phần được gọi không hoạt động như mong đợi, sau đó gây ra hành vi không mong muốn trong thành phần gọi. Ví dụ, một phương thức tìm kiếm nhị phân có thể được gọi với một tham số là một mảng không có thứ tự. Việc tìm kiếm sau đó sẽ thất bại.
* *Timing errors*, điều này xảy ra trong các hệ thống thời gian thực sử dụng bộ nhớ dùng chung hoặc giao diện truyền tin nhắn (message-passing interface). Nhà sản xuất dữ liệu và người tiêu dùng dữ liệu có thể hoạt động ở các tốc độ khác nhau. Nếu không được quan tâm đến thiết kế giao diện, người tiêu dùng có thể truy cập thông tin lỗi thời vì nhà sản xuất của thông tin chưa cập nhật thông tin giao diện được chia sẻ.

Việc kiểm tra các lỗi giao diện gặp khó khăn vì một số lỗi giao diện chỉ có thể xuất hiện trong các điều kiện bất thường. Ví dụ, giả sử một đối tượng thực hiện hàng đợi dưới dạng cấu trúc dữ liệu có độ dài cố định. Đối tượng được gọi có thể giả định rằng hàng đợi được triển khai như một cấu trúc dữ liệu vô hạn và do đó nó không kiểm tra tràn hàng đợi (queue overflow) khi một mục được nhập.

Điều kiện này chỉ có thể được phát hiện trong quá trình kiểm tra bằng cách thiết kế một chuỗi các test case buộc phải tràn hàng đợi (queue overflow). Các test case nên kiểm tra việc gọi các đối tượng xử lý tràn. Tuy nhiên, vì đây là một điều kiện hiếm gặp, những người kiểm thử có thể nghĩ rằng đây không phải là giá trị kiểm tra khi viết bộ kiểm tra cho đối tượng hàng đợi.

Một vấn đề nữa có thể phát sinh do sự tương tác giữa các lỗi trong các mô-đun hoặc đối tượng khác nhau. Lỗi ở một đối tượng chỉ có thể được phát hiện khi một số đối tượng khác hành xử theo cách không mong muốn. Nói một đối tượng gọi một đối tượng khác để nhận một số dịch vụ và đối tượng gọi giả định rằng phản hồi là chính xác. Nếu dịch vụ được gọi bị lỗi theo một cách nào đó, giá trị trả về có thể hợp lệ nhưng không chính xác. Các do đó, vấn đề không được phát hiện ngay lập tức mà chỉ trở nên rõ ràng khi một số tính toán sau này, sử dụng giá trị được trả về, bị sai.

Một số hướng dẫn chung để kiểm tra giao diện là:

1. Mã nguồn sẽ được kiểm tra và xác định mỗi cuộc gọi đến một thành phần bên ngoài. Thiết kế một tập các kiểm thử trong đó các giá trị của các tham số cho các thành phần bên ngoài nằm ở cuối cực phạm vi của chúng. Những giá trị vô cực này rất có thể tiết lộ sự không nhất quán giao diện.
2. Trong đó con trỏ được truyền qua một giao diện, luôn kiểm tra giao diện với các tham số con trỏ null.
3. Khi một thành phần được gọi thông qua giao diện thủ tục, bản thiết kế kiểm thử cố tình làm cho thành phần đó bị lỗi. Giả định thất bại khác nhau là một trong những hiểu lầm đặc điểm kỹ thuật phổ biến nhất.
4. Sử dụng stress testing trong các hệ thống truyền tin nhắn. Điều này có nghĩa là bạn nên bản thiết kế các kiểm thử tạo ra nhiều thông điệp hơn khả năng xảy ra trong thực tế. Đây là một cách hiệu quả để tiết lộ các vấn đề thời gian.
5. Khi một số thành phần tương tác thông qua bộ nhớ dùng chung, các bản thiết kế kiểm thử thay đổi thứ tự các thành phần này được kích hoạt. Các kiểm thử này có thể tiết lộ các giả định ngầm định được lập trình viên đưa ra về thứ tự mà dữ liệu chia sẻ được sản xuất và tiêu thụ.

Đôi khi, tốt hơn là sử dụng kiểm tra và đánh giá thay vì kiểm thử để tìm lỗi giao diện. Kiểm tra có thể tập trung vào các giao diện thành phần và câu hỏi về hành vi giao diện giả định được hỏi trong quá trình kiểm tra.

## 8.1.4 System testing

Kiểm tra hệ thống trong quá trình phát triển bao gồm tích hợp các thành phần để tạo phiên bản của hệ thống và sau đó kiểm tra hệ thống tích hợp. Kiểm tra hệ thống kiểm tra các thành phần có tương thích, tương tác chính xác và truyền đúng dữ liệu vào đúng thời điểm trên các giao diện của chúng không. Nó rõ ràng trùng lặp với thử nghiệm thành phần, nhưng có hai điểm khác biệt quan trọng:

1. Trong quá trình kiểm thử hệ thống, các thành phần có thể tái sử dụng đã được phát triển riêng và các hệ thống sẵn có có thể được tích hợp với các thành phần mới được phát triển. Hệ thống hoàn chỉnh sau đó được kiểm thử.
2. Các thành phần được phát triển bởi các thành viên trong nhóm hoặc nhóm nhỏ khác nhau có thể được tích hợp ở giai đoạn này. Kiểm thử hệ thống là một tập thể chứ không phải là một quá trình cá nhân. Ở một số công ty, kiểm thử hệ thống có thể liên quan đến một nhóm kiểm thử riêng biệt mà không có sự tham gia của các nhà thiết kế và lập trình viên.

Tất cả các hệ thống có hành vi nổi lên. Điều này có nghĩa là một số chức năng và đặc điểm hệ thống chỉ trở nên rõ ràng khi bạn đặt các thành phần lại với nhau. Điều này có thể được lên kế hoạch hành vi nổi lên, mà phải được kiểm thử. Ví dụ: bạn có thể tích hợp một thành phần xác thực với một thành phần cập nhật cơ sở dữ liệu hệ thống. Sau đó, bạn có một tính năng hệ thống hạn chế cập nhật thông tin cho người dùng được ủy quyền. Tuy nhiên, đôi khi, hành vi nổi lên là không có kế hoạch và không mong muốn. Bạn phải phát triển các bài kiểm thử để kiểm tra xem hệ thống chỉ đang làm những gì nó phải làm.

Kiểm thử hệ thống nên tập trung vào kiểm tra các tương tác giữa các thành phần và đối tượng tạo nên một hệ thống. Bạn cũng có thể kiểm tra các thành phần hoặc hệ thống có thể tái sử dụng để kiểm tra xem chúng có hoạt động như mong đợi khi chúng được tích hợp với các thành phần mới hay không. Kiểm tra tương tác này sẽ khám phá những lỗi thành phần chỉ được tiết lộ khi một thành phần được sử dụng bởi các thành phần khác trong hệ thống. Kiểm tra tương tác cũng giúp tìm ra những hiểu lầm, được thực hiện bởi các nhà phát triển thành phần, về các thành phần khác trong hệ thống.

Do tập trung vào các tương tác, sử dụng kiểm thử dựa trên trường hợp là một cách tiếp cận hiệu quả để kiểm tra hệ thống. Một số thành phần hoặc đối tượng thường thực hiện từng trường hợp sử dụng trong hệ thống. Test case sử dụng buộc các tương tác này xảy ra. Nếu bạn đã phát triển một sơ đồ trình tự để mô hình hóa việc thực hiện ca sử dụng, bạn có thể thấy các đối tượng hoặc các thành phần có liên quan đến tương tác.

Trong ví dụ về trạm thời tiết nơi hoang dã, phần mềm hệ thống báo cáo dữ liệu thời tiết tóm tắt cho một máy tính từ xa được mô tả trong Hình 7.3. Hình 8.8 cho thấy chuỗi các hoạt động trong trạm thời tiết khi nó đáp ứng yêu cầu thu thập dữ liệu cho hệ thống bản đồ. Bạn có thể sử dụng sơ đồ này để xác định các hoạt động sẽ được kiểm tra và giúp thiết kế các trường hợp kiểm thử để thực hiện các bài kiểm tra. Do đó, việc đưa ra một yêu cầu cho một báo cáo sẽ dẫn đến việc thực hiện các luồng phương thức sau:

SatComms:request→WeatherStation:reportWeather→Commslink:Get(summary) → WeatherData:summarize

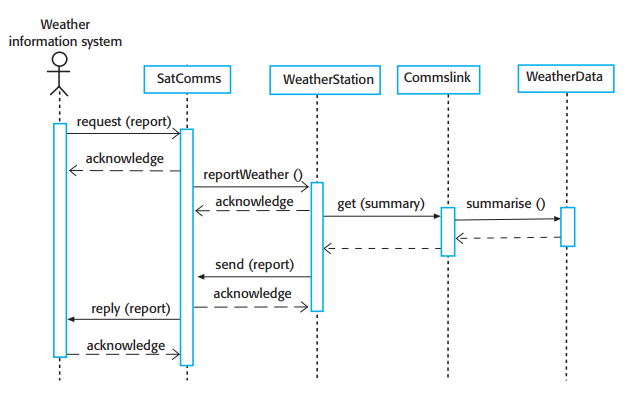


Figure 8.8 Collect weather data sequence chart

Sơ đồ trình tự (Sequence diagram) giúp bạn thiết kế các trường hợp kiểm thử cụ thể mà bạn cần, vì nó cho thấy đầu vào nào được yêu cầu và đầu ra nào được tạo:

1. Đầu vào của một yêu cầu cho báo cáo nên có một xác nhận liên quan. Một báo cáo cuối cùng nên được trả lại từ yêu cầu. Trong quá trình kiểm thử, bạn nên tạo dữ liệu tóm tắt có thể được sử dụng để kiểm tra xem báo cáo có được tổ chức chính xác không.
2. Một yêu cầu đầu vào với một báo cáo cho **WeatherStation** dẫn đến một báo cáo tóm tắt được tạo ra. Bạn có thể kiểm tra điều này một cách cô lập bằng cách tạo dữ liệu thô tương ứng với bản tóm tắt mà bạn đã chuẩn bị để kiểm tra **SatComms** và kiểm tra xem đối tượng **WeatherStation** có tạo ra bản tóm tắt này một cách chính xác không. Dữ liệu thô này cũng được sử dụng để kiểm tra đối tượng **WeatherData**.

Tất nhiên, tác giả đã đơn giản hóa sơ đồ trình tự trong Hình 8.8 để nó không hiển thị ngoại lệ. Một bài kiểm tra tình huống / kịch bản sử dụng hoàn chỉnh phải tính đến các ngoại lệ này và đảm bảo rằng chúng được xử lý chính xác.

Đối với hầu hết các hệ thống, thật khó để biết có bao nhiêu hệ thống được kiểm thử là cần thiết và khi nào bạn nên ngừng kiểm thử. Kiểm thử toàn diện, trong đó mọi trình tự thực hiện chương trình có thể được kiểm tra là điều không thể. Do đó, kiểm thử phải dựa trên một tập hợp các test case có thể. Lý tưởng nhất, các công ty phần mềm nên có chính sách cho việc lựa chọn tập hợp con này. Các chính sách này có thể dựa trên các chính sách kiểm thử chung, chẳng hạn như chính sách mà tất cả các câu lệnh chương trình nên được thực hiện ít nhất một lần. Ngoài ra, họ có thể dựa trên kinh nghiệm sử dụng hệ thống và tập trung vào kiểm tra các tính năng của hệ điều hành. Ví dụ:

|  |
| --- |
| **Incremental integration and testing**  Kiểm tra hệ thống liên quan đến việc tích hợp các thành phần khác nhau, sau đó kiểm tra hệ thống tích hợp mà bạn đã tạo. Bạn phải luôn luôn sử dụng cách tiếp cận gia tăng để tích hợp và kiểm tra khi bạn tích hợp một thành phần, kiểm tra hệ thống, tích hợp một thành phần khác, kiểm tra lại, v.v. Nếu vấn đề xảy ra, họ có lẽ do tương tác với thành phần tích hợp gần đây nhất. Tích hợp và thử nghiệm tăng dần là nền tảng cho các phương pháp nhanh, trong đó các thử nghiệm hồi quy được chạy mỗi lần một sự gia tăng mới được tích hợp.  **http://software-engineering-book.com/web/integration/** |

1. Tất cả các chức năng hệ thống được truy cập thông qua các menu nên được kiểm tra.

2. Kết hợp các chức năng (ví dụ: định dạng văn bản) được truy cập thông qua cùng một menu phải được kiểm tra.

3. Trường hợp đầu vào của người dùng được cung cấp, tất cả các chức năng phải được kiểm tra với cả đầu vào đúng và không chính xác.

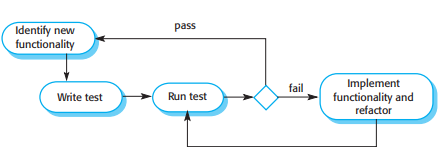
Rõ ràng từ kinh nghiệm với các sản phẩm phần mềm lớn như bộ xử lý văn bản hoặc bảng tính mà các hướng dẫn tương tự thường được sử dụng trong quá trình thử nghiệm sản phẩm. Khi các tính năng của phần mềm được sử dụng một cách cô lập, chúng thường hoạt động. Các vấn đề phát sinh, như Whittaker giải thích (Whittaker 2009), khi sự kết hợp của các tính năng ít được sử dụng chưa được kiểm thử cùng nhau. Ông đưa ra ví dụ về cách mà trong một trình xử lý văn bản thường được sử dụng chú thích có bố cục nhiều màu gây ra bố cục văn bản không chính xác.

Hệ thống kiểm thử tự động thường khó hơn kiểm thử đơn vị hoặc thành phần tự động. Kiểm thử đơn vị tự động dựa vào dự đoán đầu ra và sau đó mã hóa các dự đoán này trong một chương trình. Dự đoán sau đó được so sánh với kết quả. Tuy nhiên, điểm thực hiện một hệ thống có thể là tạo ra các đầu ra lớn hoặc không thể dự đoán dễ dàng. Bạn có thể kiểm tra một đầu ra và kiểm tra độ tin cậy của nó mà không nhất thiết có thể tạo ra nó trước.

# 8.2 Test-driven development

Test-driven development (TDD):

* Là một cách tiếp cận để phát triển chương trình, đó là việc xen kẽ giữa việc testing and code development. Trong quá trình code để phát triển phần mềm, chúng ta sẽ thực hiện việc test song song với code. Chúng ta sẽ không code phần tiếp cho đến khi phần chúng ta đang code đã được test và pass hoàn toàn.
* Test-driven development đã được giới thiệu như một phần của phương pháp phát triển phần mềm Agile.



* Quá trình TDD cơ bản được thể hiện qua các bước như trong hình:
* Xác nhận một chức năng của một yêu cầu mới. Thông thường chức năng này nhỏ và có thể thực hiện trong vài dòng lệnh.
* Viết test cho chức năng sau khi code xong. Test và report xem ticket đó pass hay fail.
* Nếu fail sẽ implement functionality and refactor và tiếp tục run test.
* Nếu pass chúng ta sẽ thực hiện qua function tiếp theo.
* Một môi trường auto test ví dụ như Junit hỗ trợ trong test chương trình Java là điều cần thiết cho TDD.
* Những đoạn code được phát triển với số lượng rất nhỏ. Do đó phải có chức năng tự động chạy test mỗi khi chúng ta thêm chức năng hoặc cấu trúc lại chương trình.

# 8.3 Release testing

Kiểm thử phát hành là quá trình kiểm tra một bản phát hành cụ thể của một hệ thống được dự định sử dụng bên ngoài nhóm phát triển. Thông thường, việc phát hành hệ thống là dành cho khách hàng và người dùng. Tuy nhiên, trong một dự án phức tạp, việc phát hành có thể dành cho các đội khác đang phát triển các hệ thống liên quan. Đối với các sản phẩm phần mềm, việc phát hành có thể dành cho quản lý sản phẩm, người sau đó chuẩn bị bán.

Có hai điểm khác biệt quan trọng giữa Kiểm thử phát hành và Kiểm thử hệ thống trong quá trình phát triển:

* Nhóm phát triển hệ thống, không nên chịu trách nhiệm kiểm thử phát hành..
* Kiểm tra phát hành là một quá trình kiểm tra xác nhận để đảm bảo rằng một hệ thống đáp ứng các yêu cầu của nó và đủ tốt sử dụng bởi hệ thống khách hàng. Kiểm tra hệ thống bởi nhóm phát triển nên tập trung vào việc phát hiện ra các lỗi trong hệ thống (kiểm tra lỗi).

Mục tiêu chính của quy trình kiểm thử phát hành là thuyết phục nhà cung cấp hệ thống rằng nó đủ tốt để sử dụng. Nếu vậy, nó có thể được phát hành dưới dạng sản phẩm hoặc phân định cho khách hàng. Do đó, kiểm thử phát hành phải chỉ ra rằng hệ thống cung cấp chức năng, hiệu suất và độ tin cậy được chỉ định và không bị lỗi trong quá trình sử dụng thông thường.

Kiểm tra phát hành thường là một quy trình kiểm thử hộp đen, theo đó các kiểm tra được lấy từ đặc tả hệ thống. Hệ thống được coi là một hộp đen mà hành vi của chúng chỉ có thể được xác định bằng cách nghiên cứu các đầu vào của nó và các đầu ra liên quan. Một tên khác cho việc này là kiểm thử chức năng, nó được gọi như vậy là vì người kiểm thử chỉ quan tâm đến chức năng chứ không liên quan đến việc triển khai phần mềm.

## 8.3.1 Kiểm thử dựa trên yêu cầu

Một nguyên tắc chung của thực hành kỹ thuật yêu cầu tốt là các yêu cầu phải được kiểm tra. Đó là, yêu cầu phải được viết để có thể thiết kế một bài kiểm tra cho yêu cầu đó. Sau đó, người kiểm tra có thể kiểm tra xem yêu cầu đã được thỏa mãn chưa. Do đó, kiểm thử dựa trên yêu cầu là một cách tiếp cận có hệ thống đối với thiết kế trường hợp kiểm thử trong đó bạn xem xét từng yêu cầu và rút ra một bộ kiểm thử cho nó. Kiểm thử dựa trên yêu cầu là xác nhận thay vì kiểm tra lỗi, bạn đang cố chứng minh rằng hệ thống đã thực hiện đúng yêu cầu của mình.

Ví dụ, hãy xem xét các yêu cầu hệ thống Mentcare sau đây có liên quan đến việc kiểm tra dị ứng thuốc:

*Nếu một bệnh nhân được biết là dị ứng với bất kỳ loại thuốc cụ thể nào, thì việc kê đơn thuốc đó sẽ dẫn đến một thông điệp cảnh báo được đưa ra cho người sử dụng hệ thống.*

*Nếu một người kê đơn chọn bỏ qua một cảnh báo dị ứng, anh ta hoặc cô ta sẽ cung cấp một lý do tại sao điều này đã bị bỏ qua.*

Để kiểm tra xem các yêu cầu này đã được thỏa mãn chưa, bạn cần phát triển một số kiểm thử liên quan:

* Thiết lập hồ sơ bệnh nhân không có dị ứng được biết đến. Kê đơn thuốc cho dị ứng được biết là tồn tại. Kiểm tra xem thông điệp cảnh báo không được hệ thống đưa ra.
* Thiết lập hồ sơ bệnh nhân bị dị ứng đã biết. Kê toa thuốc mà bệnh nhân bị dị ứng và kiểm tra xem cảnh báo có được đưa ra bởi hệ thống không?
* Thiết lập hồ sơ bệnh nhân trong đó ghi nhận dị ứng với hai loại thuốc trở lên. Kê đơn cả hai loại thuốc này một cách riêng biệt và kiểm tra xem cảnh báo chính xác cho từng loại thuốc được đưa ra.
* Kê toa hai loại thuốc mà bệnh nhân bị dị ứng. Kiểm tra xem hai cảnh báo được đưa ra chính xác.
* Kê đơn thuốc đưa ra cảnh báo và ghi đè cảnh báo đó. Kiểm tra xem hệ thống yêu cầu người dùng cung cấp thông tin giải thích lý do cảnh báo bị ghi đè.

Bạn có thể thấy trong danh sách này rằng kiểm thử một yêu cầu không có nghĩa là chỉ viết một kiểm thử. Bạn thường phải viết một số kiểm thử để đảm bảo rằng bạn có phạm vi yêu cầu. Bạn cũng nên lưu giữ hồ sơ truy xuất nguồn gốc của kiểm dựa trên yêu cầu của mình, liên kết các kiểm thử với các yêu cầu cụ thể mà bạn đã kiểm thử.

## Kiểm thử kịch bản

Kiểm thử kich bản là một cách tiếp cận để kiểm thử phát hành, theo đó bạn nghĩ ra các kịch bản sử dụng điển hình và sử dụng các kịch bản này để phát triển các trường hợp kiểm thử cho hệ thống. Kịch bản là một câu chuyện mô tả một cách mà hệ thống có thể được sử dụng. Các kịch bản phải thực tế và người dùng hệ thống thực sẽ có thể liên quan đến chúng. Nếu bạn đã sử dụng các kịch bản hoặc câu chuyện người dùng như một phần của quy trình kỹ thuật yêu cầu (được mô tả trong Chương 4), thì bạn có thể sử dụng lại chúng làm kịch bản kiểm thử.

Trong một bài viết ngắn về kiểm tra kịch bản, Kaner (Kaner 2003) đề xuất rằng kiểm tra kịch bản nên là một câu chuyện kể đáng tin cậy và khá phức tạp. Nó nên tạo ra các bên liên quan; nghĩa là, họ nên liên quan đến kịch bản và tin rằng điều quan trọng là hệ thống vượt qua bài kiểm tra. Ông cũng gợi ý rằng nó nên dễ dàng để đánh giá. Nếu có vấn đề với hệ thống, thì nhóm thử nghiệm phát hành sẽ nhận ra chúng.

Như một ví dụ về kịch bản có thể xảy ra từ hệ thống Mentcare, Kịch bản này kiểm tra một số tính năng của hệ thống Mentcare:

* Xác thực bằng cách đăng nhập vào hệ thống.
* Tải xuống và tải lên hồ sơ bệnh nhân được chỉ định vào máy tính xách tay.
* Lên lịch khám tại nhà.
* Mã hóa và giải mã hồ sơ bệnh nhân trên thiết bị di động.
* Ghi lại và sửa đổi.
* Liên kết với cơ sở dữ liệu thuốc duy trì thông tin tác dụng phụ.
* Hệ thống nhắc nhở cuộc gọi.

Nếu bạn là người kiểm thử phát hành, bạn chạy qua kịch bản này, đóng vai trò của George và quan sát cách hệ thống hành xử để đáp ứng với các đầu vào khác nhau. Là George, bạn có thể mắc lỗi cố ý, chẳng hạn như nhập sai cụm từ khóa để giải mã các bản ghi. Điều này kiểm tra phản ứng của hệ thống với lỗi. Bạn nên lưu ý với bất cứ vấn đề phát sinh nào, bao gồm các vấn đề về hiệu suất. Nếu một hệ thống quá chậm, điều này sẽ thay đổi cách sử dụng nó. Ví dụ: nếu mất quá nhiều thời gian để mã hóa bản ghi, thì người dùng không đủ thời gian có thể bỏ qua giai đoạn này. Nếu sau đó họ bị mất máy tính xách tay, một người không được ủy quyền có thể xem hồ sơ bệnh nhân.

Khi bạn sử dụng cách tiếp cận dựa trên kịch bản, thông thường bạn sẽ thử nghiệm một số yêu cầu trong cùng một kịch bản. Do đó, bạn vừa kiểm tra các yêu cầu riêng lẻ, vừa đang kiểm tra xem các kết hợp yêu cầu không gây ra vấn đề.

## Kiểm thử hiệu suất

Khi một hệ thống đã được tích hợp hoàn toàn, có thể kiểm tra các thuộc tính rõ nét, chẳng hạn như hiệu suất và độ tin cậy. Các kiểm thử hiệu suất phải được thiết kế để đảm bảo hệ thống có thể xử lý dự định nạp vào của nó. Điều này thường liên quan đến việc chạy một loạt các kiểm thử trong đó bạn tăng tải cho đến khi hệ thống hoạt động trở nên không thể chấp nhận được.

Cũng như các loại kiểm thử khác, kiểm thử hiệu năng có liên quan cả với việc giải thích rằng hệ thống đáp ứng các yêu cầu của nó và phát hiện ra các vấn đề và khiếm khuyết trong hệ thống. Để kiểm tra xem các yêu cầu về hiệu suất có đạt được hay không, bạn có thể phải xây dựng một hồ sơ hoạt động. Một hồ sơ hoạt động (xem Chương 11) là một tập hợp các bài kiểm tra phản ánh hỗn hợp công việc thực tế sẽ được hệ thống xử lý. Do đó, nếu 90% giao dịch trong một hệ thống thuộc loại A, 5% loại B và phần còn lại của loại C, D và E, thì bạn phải thiết kế hồ sơ hoạt động sao cho phần lớn các kiểm thử là thuộc loại A. Nếu không, bạn sẽ không có được một bài kiểm thử chính xác về hiệu suất hoạt động của hệ thống.

Tất nhiên, cách tiếp cận này, không nhất thiết là cách tiếp cận tốt nhất để kiểm thử nhược điểm. Kinh nghiệm đã chỉ ra rằng một cách hiệu quả để khám phá các khuyết điểm là thiết kế các kiểm thử xung quanh các giới hạn của hệ thống. Trong kiểm thử hiệu năng, điều này có nghĩa là nhấn mạnh hệ thống bằng cách đưa ra các yêu cầu nằm ngoài giới hạn thiết kế của phần mềm. Điều này được gọi là thử nghiệm áp lực.

Giả sử bạn đang kiểm tra một hệ thống xử lý giao dịch được thiết kế để xử lý tối đa 300 giao dịch mỗi giây. Bạn bắt đầu bằng cách thử nghiệm hệ thống này với ít hơn 300 giao dịch mỗi giây. Sau đó, bạn tăng dần tải trên hệ thống vượt quá 300 giao dịch mỗi giây cho đến khi vượt quá tải thiết kế tối đa của hệ thống và hệ thống bị lỗi.

Thử nghiệm áp lực giúp bạn làm hai việc:

* Kiểm tra hành vi lỗi của hệ thống. Hoàn cảnh có thể phát sinh thông qua sự kết hợp bất ngờ của các sự kiện trong đó tải được đặt trên hệ thống vượt quá tải dự kiến tối đa. Trong những trường hợp này, lỗi hệ thống không được gây ra hỏng dữ liệu hoặc mất dịch vụ người dùng bất ngờ. kiểm thử áp lực kiểm tra rằng quá tải hệ thống khiến hệ thống không thành công, thay vì sụp đổ dưới tải.
* Hiển thị các lỗi chỉ hiển thị khi hệ thống được tải đầy đủ. Mặc dù có thể lập luận rằng những khiếm khuyết này không có khả năng gây ra lỗi hệ thống khi sử dụng, nhưng có thể có những kết hợp bất thường của tình huống mà thử nghiệm áp lực lặp lại.

Thử nghiệm áp lực đặc biệt liên quan đến các hệ thống phân tán dựa trên một mạng lưới các bộ xử lý. Các hệ thống này thường thể hiện sự xuống cấp nghiêm trọng khi chúng quá tải. Mạng tràng trong dữ liệu phối hợp mà các quy trình khác nhau phải trao đổi. Các quy trình trở nên chậm hơn và chậm hơn khi chúng chờ dữ liệu cần thiết từ các quy trình khác. Thử nghiệm áp lực giúp bạn khám phá khi sự xuống cấp bắt đầu để bạn có thể thêm séc vào hệ thống để từ chối các giao dịch vượt quá điểm này.

# 8.4 User testing

Kiểm thử người dùng hoặc khách hàng là một giai đoạn trong quy trình kiểm thử, trong đó người dùng hoặc khách hàng cung cấp đầu vào và tư vấn về kiểm thử hệ thống. Điều này có thể liên quan đến việc kiểm thử một hệ thống chính thức đã được ủy quyền từ một nhà cung cấp bên ngoài. Ngoài ra, nó có thể là một quy trình không chính thức nơi người dùng thử nghiệm một sản phẩm phần mềm mới để xem họ có thích nó không và để kiểm tra xem nó có làm những gì họ cần không. Kiểm tra người dùng là điều cần thiết, ngay cả khi hệ thống toàn diện và kiểm tra phát hành đã được thực hiện. Ảnh hưởng từ môi trường làm việc của người dùng có thể có ảnh hưởng lớn đến độ tin cậy, hiệu suất, khả năng sử dụng và sự mạnh mẽ của một hệ thống.

Trên thực tế, nhà phát triển hệ thống không thể sao chép môi trường làm việc của hệ thống, vì các thử nghiệm trong môi trường của nhà phát triển không thể tránh khỏi việc nhân tạo ra nó. Ví dụ, một hệ thống được thiết kế để sử dụng trong bệnh viện được sử dụng trong môi trường lâm sàng nơi những điều khác đang diễn ra, chẳng hạn như trường hợp khẩn cấp của bệnh nhân và các cuộc trò chuyện với người thân. Tất cả đều ảnh hưởng đến việc sử dụng một hệ thống, nhưng các nhà phát triển không thể đưa chúng vào môi trường kiểm thử của họ.

Có 3 loại user testing khác nhau:

1. *Alpha testing*, trong đó một nhóm người dùng phần mềm được chọn làm việc chặt chẽ với nhóm phát triển để kiểm tra các bản phát hành sớm của phần mềm.
2. *Beta testing*, trong đó bản phát hành phần mềm được cung cấp cho một nhóm người dùng lớn hơn để cho họ trải nghiệm và đưa ra các vấn đề mà họ phát hiện ra với các nhà phát triển hệ thống.
3. *Acceptance testing*, trong đó khách hàng kiểm tra một hệ thống để quyết định xem nó có sẵn sàng được chấp nhận từ các nhà phát triển hệ thống và triển khai trong môi trường khách hàng hay không.

Trong kiểm thử alpha, người dùng và nhà phát triển làm việc cùng nhau để kiểm tra hệ thống khi nó đang được phát triển. Điều này có nghĩa là người dùng có thể xác định các vấn đề và vấn đề không dễ thấy đối với nhóm thử nghiệm phát triển. Các nhà phát triển chỉ có thể thực sự làm việc từ các yêu cầu, nhưng những điều này thường không phản ánh các yếu tố khác ảnh hưởng đến việc sử dụng phần mềm thực tế. Do đó, người dùng có thể cung cấp thông tin về thực hành giúp thiết kế các bài kiểm tra thực tế hơn.

Kiểm thử alpha thường được sử dụng khi phát triển các sản phẩm hoặc ứng dụng phần mềm. Người dùng có kinh nghiệm về các sản phẩm này có thể sẵn sàng tham gia vào quá trình kiểm thử alpha vì điều này cung cấp cho họ thông tin sớm về các tính năng hệ thống mới mà họ có thể khai thác. Nó cũng làm giảm nguy cơ những thay đổi không lường trước được đối với phần mềm ảnh hưởng đột phá đến kinh doanh của họ. Tuy nhiên, kiểm thử alpha cũng có thể được sử dụng khi phần mềm tùy chỉnh đang được phát triển. Các phương pháp phát triển Agile ủng hộ sự tham gia của người dùng vào quá trình phát triển và người dùng nên đóng vai trò chính trong việc thiết kế các thử nghiệm cho hệ thống.

Kiểm thử beta diễn ra khi việc phát hành sớm, đôi khi chưa hoàn thành, một hệ thống phần mềm được cung cấp cho một nhóm lớn hơn của khách hàng và người dùng để đánh giá. Những người thử nghiệm bản beta có thể là một nhóm khách hàng được lựa chọn là những người sớm chấp nhận hệ thống.

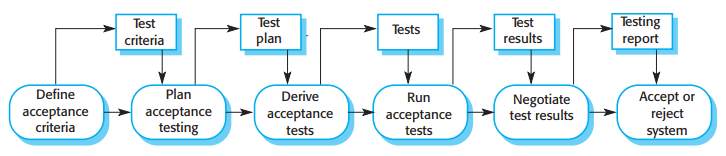


Figure 8.11 The acceptance testing process

Ngoài ra, phần mềm có thể được cung cấp công khai để sử dụng bởi bất kỳ ai quan tâm đến việc thử nghiệm nó.

Kiểm thử beta chủ yếu được sử dụng cho các sản phẩm phần mềm được sử dụng trong nhiều cài đặt khác nhau. Điều này rất quan trọng vì, không giống như các nhà phát triển sản phẩm tùy chỉnh, không có cách nào để nhà phát triển sản phẩm giới hạn môi trường hoạt động của phần mềm. Các nhà phát triển sản phẩm không thể biết và sao chép tất cả các cài đặt mà sản phẩm phần mềm sẽ được sử dụng. Do đó, kiểm thử beta được sử dụng để khám phá các vấn đề tương tác giữa phần mềm và các tính năng của môi trường hoạt động. Kiểm thử beta cũng là một hình thức tiếp thị. Khách hàng tìm hiểu về hệ thống của họ và những gì nó có thể làm cho họ.

Acceptance testing là một phần vốn có của phát triển hệ thống tùy chỉnh. Khách hàng kiểm tra một hệ thống, sử dụng dữ liệu của riêng họ và quyết định xem nó có được chấp nhận từ nhà phát triển hệ thống hay không. Chấp nhận ngụ ý rằng thanh toán cuối cùng nên được thực hiện cho phần mềm. Hình 8.11 cho thấy đây là sáu giai đoạn trong quy trình acceptance testing:

1. Define acceptance criteria, giai đoạn này lý tưởng nên diễn ra sớm trong quá trình trước khi hợp đồng cho hệ thống được ký kết. Các tiêu chí chấp nhận phải là một phần của hợp đồng hệ thống và được khách hàng và nhà phát triển chấp thuận. Tuy nhiên, trong thực tế, có thể khó xác định các tiêu chí quá sớm trong quá trình này. Yêu cầu chi tiết có thể không có sẵn, và các yêu cầu gần như chắc chắn sẽ thay đổi trong quá trình phát triển.
2. Plan acceptance testing, giai đoạn này liên quan đến việc quyết định các nguồn lực, thời gian và ngân sách để thử nghiệm chấp nhận và thiết lập lịch thử nghiệm. Kế hoạch kiểm tra chấp nhận cũng nên thảo luận về phạm vi yêu cầu của các yêu cầu và thứ tự các tính năng hệ thống được kiểm tra. Nó nên xác định rủi ro cho quá trình thử nghiệm như sự cố hệ thống và hiệu suất không đầy đủ và thảo luận về cách giảm thiểu những rủi ro này.
3. Derive acceptance tests, khi các tiêu chí chấp nhận đã được thiết lập, các kiểm thử phải được thiết kế để kiểm tra xem một hệ thống có được chấp nhận hay không. Các thử nghiệm chấp nhận nên nhằm mục đích kiểm tra cả các đặc tính chức năng và phi chức năng (ví dụ: hiệu suất) của hệ thống. Họ nên lý tưởng cung cấp bảo hiểm đầy đủ của các yêu cầu hệ thống. Trong thực tế, rất khó để thiết lập các tiêu chí chấp nhận hoàn toàn khách quan. Thường có phạm vi tranh luận về việc có hay không một bài kiểm tra cho thấy một tiêu chí chắc chắn đã được đáp ứng.
4. Run acceptance tests, các kiểm thử được đồng ý được thực hiện trên hệ thống. Lý tưởng nhất là bước này sẽ diễn ra trong môi trường thực tế nơi hệ thống sẽ được sử dụng, nhưng điều này có thể gây rối và không thực tế. Do đó, một môi trường thử nghiệm người dùng có thể phải được thiết lập để chạy các thử nghiệm này. Rất khó để tự động hóa quy trình này vì một phần của các thử nghiệm chấp nhận có thể liên quan đến việc kiểm tra các tương tác giữa người dùng cuối và hệ thống. Một số đào tạo của người dùng cuối có thể được yêu cầu.
5. Negotiate test results, rất khó có khả năng tất cả các bài kiểm thử chấp nhận được xác định sẽ vượt qua và sẽ không có vấn đề gì với hệ thống. Nếu đây là trường hợp, sau đó thử nghiệm chấp nhận hoàn tất và hệ thống có thể được bàn giao. Thông thường hơn, một số vấn đề sẽ được phát hiện. Trong những trường hợp như vậy, nhà phát triển và khách hàng phải thương lượng để quyết định xem hệ thống có đủ tốt để sử dụng hay không. Họ cũng phải đồng ý về cách nhà phát triển sẽ khắc phục các sự cố đã xác định.
6. Reject/accept system, giai đoạn này bao gồm một cuộc gặp mặt giữa các nhà phát triển và khách hàng quyết định xem có nên chấp nhận hệ thống hay không. Nếu hệ thống không đủ tốt để sử dụng, thì cần phải phát triển thêm để khắc phục các sự cố đã xác định. Sau khi hoàn thành, giai đoạn thử nghiệm chấp nhận được lặp lại.

Bạn có thể nghĩ rằng acceptance testing là một vấn đề hợp đồng rõ ràng. Nếu một hệ thống không vượt qua các bài kiểm tra chấp nhận của nó, thì nó không nên được chấp nhận và không nên thực hiện thanh toán. Tuy nhiên, thực tế phức tạp hơn. Khách hàng muốn sử dụng phần mềm càng sớm càng tốt vì lợi ích của việc triển khai ngay lập tức. Họ có thể đã mua phần cứng mới, nhân viên được đào tạo và thay đổi quy trình của họ. Họ có thể sẵn sàng chấp nhận phần mềm, bất kể các vấn đề, bởi vì chi phí không sử dụng phần mềm lớn hơn chi phí làm việc xung quanh các vấn đề.

Do đó, kết quả của các cuộc đàm phán có thể là sự chấp nhận có điều kiện của hệ thống. Khách hàng có thể chấp nhận hệ thống để việc triển khai có thể bắt đầu. Nhà cung cấp hệ thống đồng ý sửa chữa các sự cố khẩn cấp và cung cấp phiên bản mới cho khách hàng nhanh nhất có thể.

Trong các phương thức nhanh như Extreme Programming, có thể không có hoạt động kiểm tra chấp nhận riêng biệt. Người dùng cuối là một phần của nhóm phát triển (tức là anh ta hoặc cô ta là người thử nghiệm alpha) và cung cấp các yêu cầu hệ thống về các câu chuyện của người dùng. Người đó cũng chịu trách nhiệm xác định các bài kiểm tra, quyết định xem phần mềm được phát triển có hỗ trợ các câu chuyện của người dùng hay không. Do đó, các xét nghiệm này tương đương với các thử nghiệm chấp nhận. Các thử nghiệm được tự động hóa và quá trình phát triển không được tiến hành cho đến khi các thử nghiệm chấp nhận câu chuyện được thực hiện thành công.

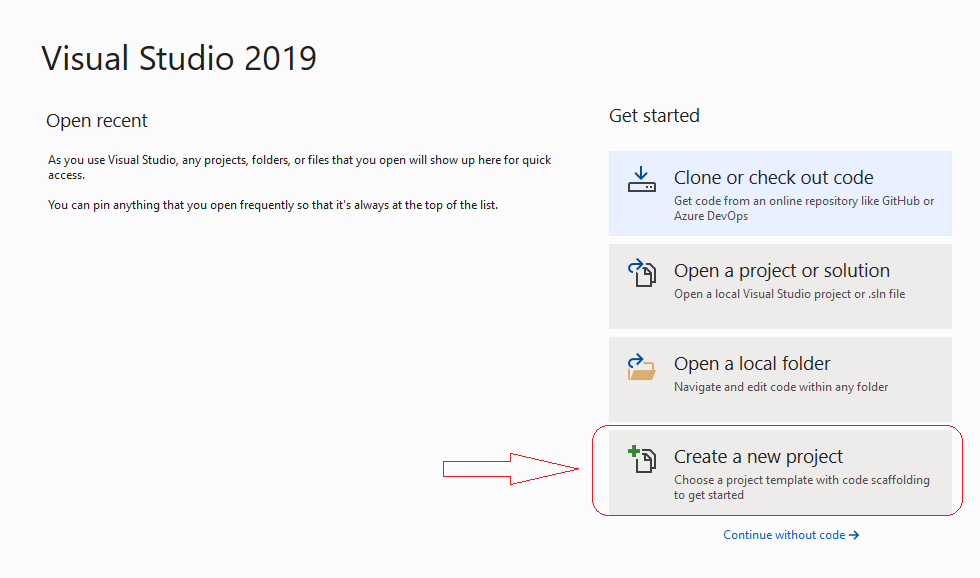
Khi người dùng được gắn vào một nhóm phát triển phần mềm, lý tưởng nhất là người dùng của họ điển hình với các kiến ​​thức chung về cách sử dụng hệ thống. Tuy nhiên, nó có thể khó tìm thấy những người dùng như vậy, và vì vậy các bài kiểm tra chấp nhận thực sự có thể không phải là một sự phản ánh đúng về cách sử dụng một hệ thống trong thực tế. Hơn nữa, yêu cầu kiểm tra tự động giới hạn tính linh hoạt của các hệ thống tương tác thử nghiệm. Đối với các hệ thống như vậy, thử nghiệm chấp nhận có thể yêu cầu các nhóm người dùng cuối sử dụng hệ thống như thể đó là một phần công việc hàng ngày của họ. Do đó, về nguyên tắc, một người dùng nhúng nhúng là một khái niệm hấp dẫn về nguyên tắc, nó không nhất thiết dẫn đến các thử nghiệm chất lượng cao của hệ thống. Vấn đề về sự tham gia của người dùng vào các nhóm nhanh là một lý do tại sao nhiều công ty sử dụng kết hợp thử nghiệm nhanh và truyền thống hơn. Hệ thống có thể được phát triển bằng các kỹ thuật nhanh, nhưng, sau khi hoàn thành một bản phát hành chính, thử nghiệm chấp nhận riêng biệt được sử dụng để quyết định xem hệ thống có được chấp nhận hay không.

# Cài đặt

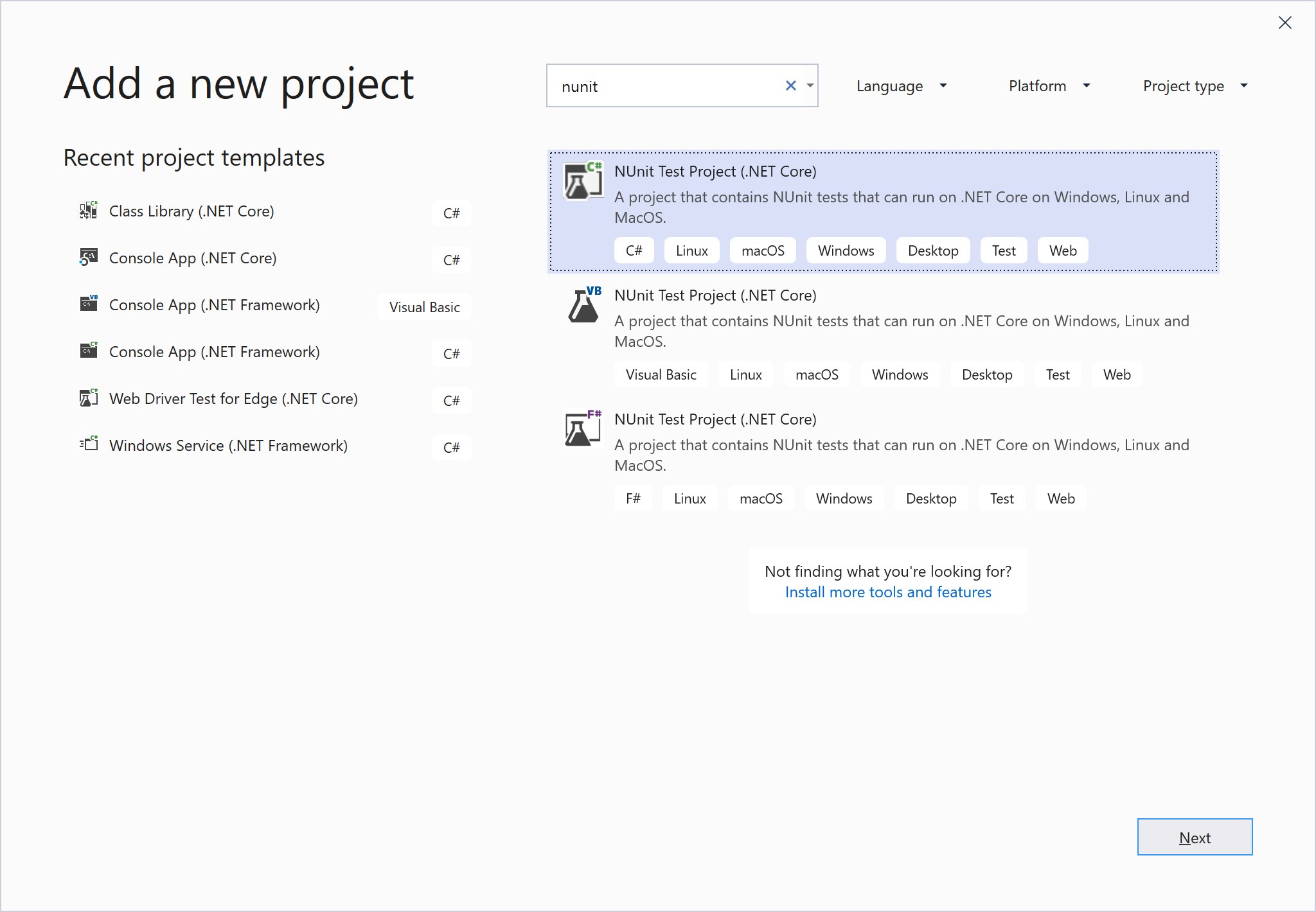
Chúng ta sẽ cài đặt NUnit test, ngoài ra chúng ta còn có những Unit test khác như JUnit test, PHPUnit, JMockit, v.v. trong phần này chúng ta sẽ làm việc với NUnit test framework.

1. Đầu tiên chúng ta sẽ mở visual studio phiên bản 2019

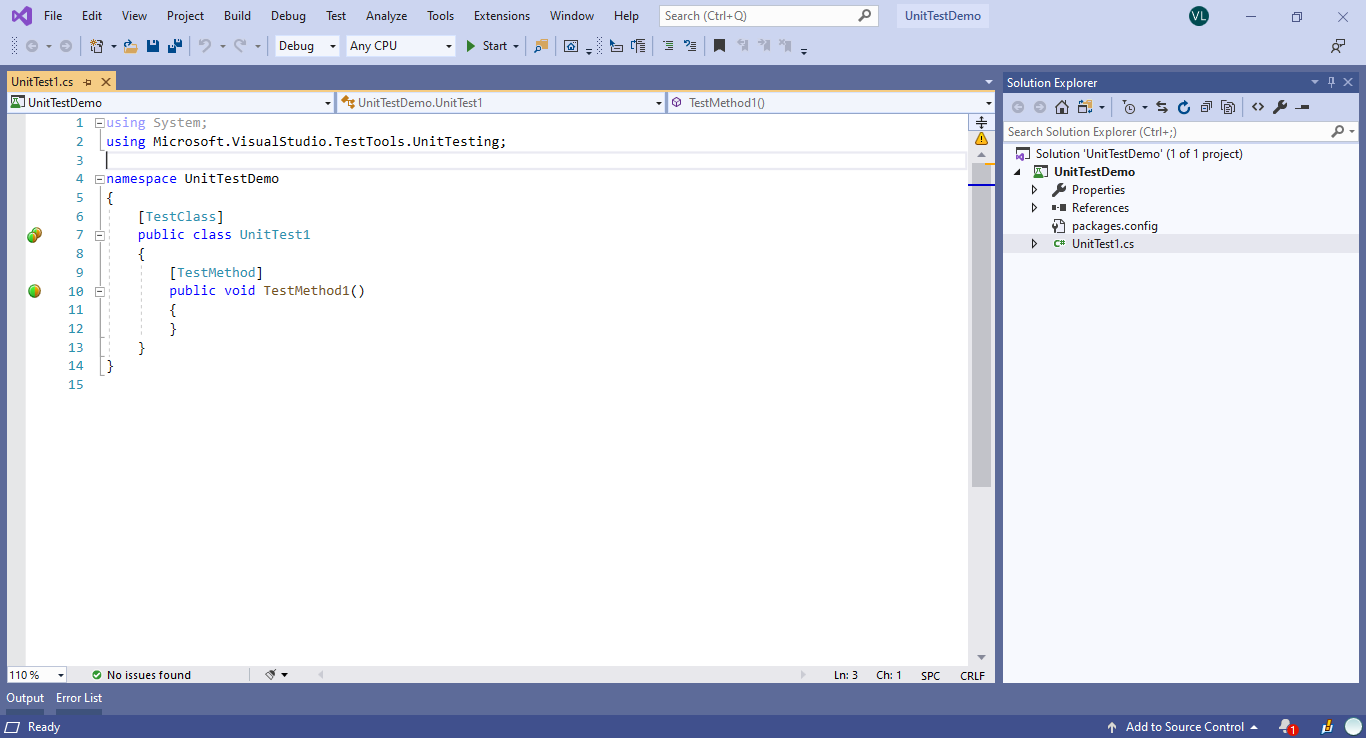
Windows -> Start -> Visual Studio 2019.



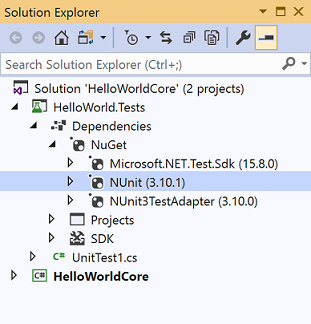
1. Ta sẽ chọn create a new project.



Trong combobox All language ta chọn C#, và combobox All project type ta chọn Test sau khi chọn xong nó sẽ hiện ra danh sách các dự án, ta chọn NUnit Test project (.NET Core) và ta chọn Next để khởi tạo.



1. Dự án ta đã thêm vào trong solution



Chúng ta sẽ thêm NuGet tham chiếu đến NUnit và NUnit3TestAdapter.

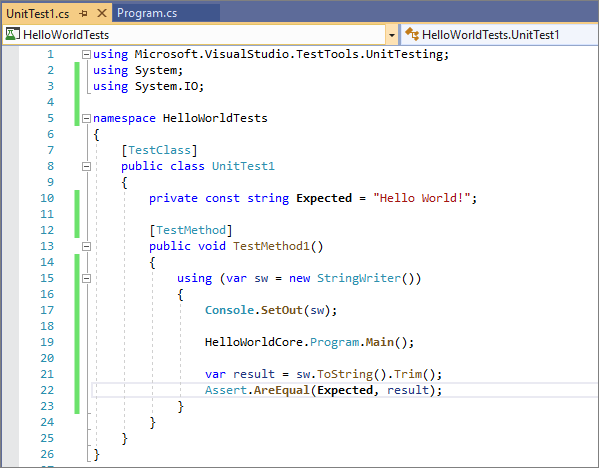
1. Trong unit test project, add một reference vào trong project mà chúng ta muốn test bằng cách nhấn chuột phải vào HelloWorldTests chọn

**References -> Add Reference.**

Chọn project chứa code mà chúng ta muốn test.

****

1. Thêm code cho phương phức unit test



1. Chạy test từ Test Explorer hoặc là click chuột phải vào đoạn code chọn **Run test(s)**

# 

# KEY POINTS

* Testing can only show the presence of errors in a program. It cannot show that there are no remaining faults.
* Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers. In the user testing process, customers or system users provide test data and check that tests are successful.
* Development testing includes unit testing in which you test individual objects and methods; component testing in which you test related groups of objects; and system testing in which you test partial or complete systems.
* When testing software, you should try to “break” the software by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems. Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
* Test-first development is an approach to development whereby tests are written before the code to be tested. Small code changes are made, and the code is refactored until all tests execute successfully.
* Scenario testing is useful because it replicates the practical use of the system. It involves inventing a typical usage scenario and using this to derive test cases.
* Acceptance testing is a user testing process in which the aim is to decide if the software is good enough to be deployed and used in its planned operational environment.

# 

# FURTHER READING

“How to design practical test cases.” A how-to article on test-case design by an author from a Japanese company that has a good reputation for delivering software with very few faults. (T. Yamaura, *IEEE Software*, 15(6), November 1998) <http://dx.doi.org/10.1109/52.730835>.

“Test-driven development.” This special issue on test-driven development includes a good general overview of TDD as well as experience papers on how TDD has been used for different types of software. (*IEEE Software*, 24 (3) May/June 2007).

*Exploratory Software Testing*. This is a practical, rather than theoretical, book on software testing which develops the ideas in Whittaker’s earlier book, *How to Break Software.* The author presents a set of experience-based guidelines on software testing. (J. A. Whittaker, 2009, Addison-Wesley).

*How Google Tests Software*. This is a book about testing large-scale cloud-based systems and poses a whole set of new challenges compared to custom software applications. While I don’t think that the Google approach can be used directly, there are interesting lessons in this book for largescale system testing. (J. Whittaker, J. Arbon, and J. Carollo, 2012, Addison-Wesley).