

MỤC LỤC

MỤC LỤC	1
ĐÁNH GIÁ CỦA GIÁO VIÊN HƯỚNG DẪN	4
PHẦN I: PHẦN MỞ ĐẦU	5
PHẦN II: PHẦN NỘI DUNG	7
CHƯƠNG I: KHÁI QUÁT KIỂM THỬ PHẦN MỀM	7
1.1. Các khái niệm cơ bản trong kiểm thử phần mềm.....	7
1.1.1. Khái niệm về kiểm thử phần mềm.....	7
1.1.2. Mục đích của kiểm thử phần mềm.	7
1.1.3. Các phương pháp kiểm thử.....	8
1.1.4. Các chiến lược kiểm thử.....	8
1.1.4.1. Kiểm thử hộp đen – Black box.....	8
1.1.4.2. Kiểm thử hộp trắng – White box.....	10
1.1.4.3. Kiểm thử hộp xám – Gray box testing	10
1.1.5. Các cấp độ kiểm thử trong kiểm thử phần mềm.....	11
1.1.5.1. Kiểm thử đơn vị - Unit test.	11
1.1.5.2. Kiểm thử tích hợp – Intergration test.	12
1.1.5.3. Kiểm thử hệ thống – System test.	14
1.1.5.4. Kiểm thử chấp nhận – Acceptance test.	15
1.1.5.5. Mô hình chữ V trong kiểm thử phần mềm.....	16
1.1.6. Một số cấp độ kiểm thử khác.....	18
1.2. Nguyên tắc trong kiểm thử phần mềm.	19
CHƯƠNG II: UNIT TESTING	19
2.1. Tổng quan về Unit test	19
2.1.1. Định nghĩa về Unit testing.....	20
2.1.2. Mục đích	20
2.1.3. Yêu cầu.	20
2.1.4. Người thực hiện Unit test.	20
2.1.5. Vòng đời của một Unit test.....	21
2.1.6. Lợi ích của Unit test.	21
2.1.7. Tác dụng của Unit test.	22

2.1.8.	Chiến lược viết mã hiệu quả với Unit test.	22
2.2.	Sử dụng Unit test với mô hình đối tượng ảo (Mock Object)	23
2.2.1.	Định nghĩa	23
2.2.2.	Đặc điểm.....	23
2.2.3.	Lợi ích.....	24
2.2.4.	Phạm vi sử dụng	24
2.2.5.	Các đối tượng được mô phỏng.	25
2.2.6.	Thiết kế MO.....	25
CHƯƠNG III: THIẾT KẾ TEST CASE		26
3.1.	Định nghĩa.	26
3.2.	Vai trò của việc thiết kế test case.	27
3.3.	Quy trình thiết kế test case.	27
3.3.1.	Phương pháp kiểm thử hộp đen – Black box testing.....	27
3.3.1.1.	Phân vùng tương đương	27
3.3.1.2.	Phân tích giá trị biên – Boundary Values Analysis.....	31
3.3.1.3.	Đồ thị nguyên nhân – hệ quả.....	36
3.3.1.4.	Đoán lỗi – Error Guessing.....	42
3.3.2.	Phương pháp kiểm thử hộp trắng – White box testing.	42
3.3.2.1.	Bao phủ câu lệnh.	42
3.3.2.2.	Bao phủ quyết định.	44
3.3.2.3.	Bao phủ điều kiện.....	45
3.3.2.4.	Bao phủ quyết định – điều kiện.....	46
3.3.2.5.	Bao phủ đa điều kiện.....	48
CHƯƠNG IV: TÌM HIỂU VỀ NUNIT		50
4.1.	Các công cụ kiểm thử của từng ngôn ngữ kiểm thử.....	50
4.1.1.	Junit và J2ME Unit trong Java.	50
4.1.2.	C++ Unit trong C/C++.	51
4.1.3.	Vb Unit trong Visual Basic.	52
4.1.4.	PyUnit trong Python.	52
4.1.5.	Perl Unit trong Perl.....	53
4.2.	Nunit trong C#.....	53

4.2.1.	Định nghĩa.	53
4.2.2.	Đặc điểm của NUnit.	53
4.2.3.	Thuộc tính hay dùng trong thư viện Nunit.Framework.	54
4.2.4.	Phương thức tĩnh hay dùng trong Nunit.Framework.Assert	57
4.2.5.	Cài đặt Nunit.	58
4.2.6.	Cách sử dụng Nunit.	61
4.2.6.1.	Hướng dẫn tạo test case trong Visual studio 2008.	61
4.2.6.2.	Sử dụng Nunit.	65
CHƯƠNG V: CHƯƠNG TRÌNH DEMO.....		70
5.1.	Phát biểu bài toán.	70
5.2.	Đặt vấn đề.....	70
5.3.	Phân tích và thiết kế bài toán.....	70
5.4.	Thiết kế các test case.	71
5.5.	Ứng dụng chương trình	76
5.6.	Tổng kết chương trình demo	80
PHẦN III: PHẦN KẾT LUẬN.....		81
TÀI LIỆU THAM KHẢO		82

This image shows a full page of white paper with horizontal dotted lines, typical of primary school writing paper. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Giáo viên hướng dẫn

PHẦN I: PHẦN MỞ ĐẦU

Kiểm thử phần mềm là khâu sống còn của việc phát triển phần mềm. Hai chữ "kiểm thử" nghe có vẻ đơn giản, nhàn rỗi nhưng khâu này lại giúp cho sản phẩm được hoàn thiện nhằm đáp ứng yêu cầu đặt ra của khách hàng. Sản phẩm hoàn thiện, chất lượng cao sẽ tạo thêm niềm tin và uy tín của công ty với đối tác trong và ngoài nước. Nếu không có khâu kiểm thử phần mềm, tình trạng khách hàng trả lại sản phẩm về cho người phát triển phần mềm đó sẽ xảy ra thường xuyên. Chính vì vậy, tester là vị trí không thể thiếu và công việc này quyết định khá nhiều vào sự thành công chung của dự án phát triển phần mềm.

Việt Nam hiện nay đang được đánh giá sẽ trở thành con hổ trong ngành kiểm thử phần mềm châu Á với lượng nhân công trẻ và nhiều doanh nghiệp đang phát triển theo con đường này. Tại Việt Nam, những ai theo học ngành Công nghệ thông tin đều đa phần là nghĩ ngay đến nghề lập trình vì thế khiến đầu ra của nghề kiểm thử phần mềm có số lượng thấp hơn hẳn so với chuyên môn lập trình viên khiến các nhà tuyển dụng rất vất vả trong việc tìm kiếm nguồn nhân lực cho ngành kiểm thử phần mềm. Nhưng cũng nhờ đó mà những ai định hướng theo nghề tester ngay từ đầu có thể yên tâm có trong tay tấm vé xin việc làm ngay khi vừa tốt nghiệp.

Với xu hướng phát triển ngành kiểm thử phần mềm của Việt Nam nói riêng cũng như của Châu Á nói chung thì việc nhóm sinh viên em chọn đề tài làm về kiểm thử phần mềm: ***“Nghiên cứu về Unit Testing trong C# với NUnit và viết chương trình demo”*** là đúng đắn và hơn hết là hợp thời đại bây giờ.

Đề tài nghiên cứu về kiểm thử phần mềm này thì nhóm sinh viên chúng em có thể hiểu được khái quát về kiểm thử phần mềm, hiểu được về một số công cụ dùng để kiểm thử như NUnit cho dotNet, Junit cho ngôn ngữ Java,...và hiểu được việc thiết kế test – case trong kiểm thử mức đơn vị (Unit test). Hơn hết, em có thể biết thêm một số công cụ kiểm thử tự động như: QuickTestProfessional, LoadRunner, hay Test Complete...

Trong quá trình thực hiện nghiên cứu đề tài, chúng em nhận được sự hướng dẫn tận tình của cô giáo *Lê Thị Thu Hương* – giáo viên trực tiếp hướng dẫn, chúng em còn nhận được sự hướng dẫn của các thầy cô giáo trong bộ môn Công nghệ phần mềm và tất cả các bạn trong bộ môn. Chúng em hy vọng sẽ nhận được sự góp ý của các thầy cô và các bạn

để chúng em có thể hoàn thành tốt đề tài này. Những đóng góp của mọi người sẽ là những kinh nghiệm quý báu giúp em và các bạn trong nhóm có những dự định sau này trong khi làm đồ án tốt nghiệp và sau khi tốt nghiệp.

Một lần nữa em xin chân thành cảm ơn cô giáo *Lê Thị Thu Hương* đã hướng dẫn em và các bạn hoàn thành đề tài nghiên cứu.

Em xin chân thành cảm ơn!

Nhóm sinh viên:

Đỗ Thùy Dung

Nguyễn Thị Huệ

Nguyễn Thị Hương

PHẦN II: PHẦN NỘI DUNG

CHƯƠNG I: KHÁI QUÁT KIỂM THỬ PHẦN MỀM

1.1. Các khái niệm cơ bản trong kiểm thử phần mềm.

1.1.1. Khái niệm về kiểm thử phần mềm.

- **Kiểm thử phần mềm** là quá trình khảo sát một hệ thống hay thành phần dưới những điều kiện xác định, quan sát và ghi lại các kết quả, và đánh giá một khía cạnh nào đó của hệ thống hay thành phần đó. (Theo Bảng chú giải thuật ngữ chuẩn IEEE của Thuật ngữ kỹ nghệ phần mềm- IEEE Standard Glossary of Software Engineering Terminology).

- **Kiểm thử phần mềm** là quá trình thực thi một chương trình với mục đích tìm ra nhiều lỗi. (Theo “The Art of Software Testing” – Nghệ thuật kiểm thử phần mềm).

- **Kiểm thử phần mềm** là hoạt động khảo sát thực tiễn sản phẩm hay dịch vụ phần mềm trong đúng môi trường chúng dự định sẽ được triển khai nhằm cung cấp cho người có lợi ích liên quan những thông tin về chất lượng của sản phẩm hay dịch vụ phần mềm ấy. Mục đích của kiểm thử phần mềm là tìm ra các lỗi hay khiếm khuyết phần mềm nhằm đảm bảo hiệu quả hoạt động tối ưu của phần mềm trong nhiều ngành khác nhau. (Theo Bách khoa toàn thư mở Wikipedia).

- Có thể định nghĩa một cách dễ hiểu như sau: **Kiểm thử phần mềm** là một tiến trình hay một tập hợp các tiến trình được thiết kế để đảm bảo mã hóa máy tính thực hiện theo cái mà chúng đã được thiết kế để làm, và không thực hiện bất cứ thứ gì không mong muốn. Đây là một pha quan trọng trong quá trình phát triển hệ thống, giúp cho người xây dựng hệ thống và khách hàng thấy được hệ thống mới đã đáp ứng yêu cầu đặt ra hay chưa.

1.1.2. Mục đích của kiểm thử phần mềm.

- Tìm ra nhiều lỗi bằng việc đưa ra các dòng thời gian.
- Chứng minh được sản phẩm hoàn thành có những chức năng hay ứng dụng giống với bản đặc tả yêu cầu.
- Tạo ra các test case có chất lượng cao, thực thi hiệu quả...

- Một số lỗi cơ bản trong kiểm thử phần mềm như: lỗi ngay từ khi phân tích yêu cầu, lỗi từ bản đặc tả hệ thống, lỗi trong code, lỗi hệ thống và nguồn tài nguyên hệ thống, lỗi trong vấn đề phần mềm, phần cứng...

1.1.3. Các phương pháp kiểm thử.

- **Kiểm thử tĩnh(Static testing):** Là phương pháp thử phần mềm đòi hỏi phải duyệt lại các yêu cầu và các đặc tả bằng tay, thông qua việc sử dụng giấy, bút để kiểm tra logic, lần từng chi tiết mà không cần chạy chương trình. Kiểu kiểm thử này thường được sử dụng bởi chuyên viên thiết kế người mà viết mã lệnh một mình.

Kiểm thử tĩnh cũng có thể được tự động hóa. Nó sẽ thực hiện kiểm tra toàn bộ bao gồm các chương trình được phân tích bởi một trình thông dịch hoặc biên dịch mà xác nhận tính hợp lệ về cú pháp của chương trình.

- **Kiểm thử động(Dynamic testing):** Là phương pháp kiểm thử thông qua việc dùng máy chạy chương trình để điều tra trạng thái tác động của chương trình. Đó là kiểm thử dựa trên các ca kiểm thử xác định bằng sự thực hiện của đối tượng kiểm thử hay chạy các chương trình. Kiểm thử động là kiểm tra cách thức hoạt động của mã lệnh, tức là kiểm tra sự phản ứng vật lý từ hệ thống tới các biến luôn thay đổi theo thời gian. Trong kiểm thử động, phần mềm phải thực sự được biên dịch và chạy. Kiểm thử động thực sự bao gồm làm việc với phần mềm, nhập các giá trị đầu vào và kiểm tra xem liệu đầu ra có như mong muốn hay không.

Các phương pháp kiểm thử động gồm có kiểm thử mức đơn vị – Unit Tests, kiểm thử tích hợp – Intergration Tests, kiểm thử hệ thống – System Tests, và kiểm thử chấp nhận sản phẩm – Acceptance Tests.

1.1.4. Các chiến lược kiểm thử.

Trong chiến lược kiểm thử, chúng ta có ba chiến lược kiểm thử hay dùng nhất là: kiểm thử hộp đen, kiểm thử hộp trắng, và kiểm thử hộp xám.

1.1.4.1. Kiểm thử hộp đen – Black box.

Một trong những chiến lược kiểm thử quan trọng là kiểm thử hộp đen, hướng dữ liệu, hay hướng vào ra. Kiểm thử hộp đen xem chương trình như là một “hộp đen”. Mục đích của bạn là hoàn toàn không quan tâm về cách xử lý và cấu trúc bên trong của chương

trình. Thay vào đó, tập trung vào tìm các trường hợp mà chương trình không thực hiện theo các đặc tả của nó.

Theo hướng tiếp cận này, dữ liệu kiểm tra được lấy từ các đặc tả.

Các phương pháp kiểm thử hộp đen

- Phân lớp tương đương – *Equivalence partitioning*.
- Phân tích giá trị biên – *Boundary values analysis*.
- Kiểm thử mọi cặp – *All pairs testing*.
- Kiểm thử dựa trên mô hình – *Model based testing*.
- Kiểm thử thăm dò – *Exploratory testing*
- Kiểm thử dựa trên đặc tả - *Specification base testing*.

Kiểm thử dựa trên đặc tả tập trung vào kiểm tra tính thiết thực của phần mềm theo những yêu cầu thích hợp. Do đó, kiểm thử viên nhập dữ liệu vào, và chỉ thấy dữ liệu ra từ đối tượng kiểm thử. Mức kiểm thử này thường xuyên yêu cầu các ca kiểm thử triệt để được cung cấp cho kiểm thử viên mà khi đó có thể xác minh là đối với dữ liệu đầu vào đã cho giá trị đầu ra(hay cách thức hoạt động) có giống với giá trị mong muốn đã được xác định trong ca kiểm thử đó hay không. Kiểm thử dựa trên đặc tả là cần thiết, nhưng không đủ để ngăn chặn những rủi ro chắc chắn.

Ưu, nhược điểm

Kiểm thử hộp đen không có mối liên quan nào tới mã lệnh và kiểm thử viên chỉ rất đơn giản tam niệm là: một mã lệnh phải có lỗi. Sử dụng nguyên tắc “Hãy đòi hỏi và bạn sẽ được nhận”, những kiểm thử viên hộp đen tìm ra lỗi mà những lập trình viên không tìm ra. Nhưng, người ta nói kiểm thử hộp đen “giống như là đi trong bóng tối mà không có đèn vậy”, bởi vì kiểm thử viên không biết các phần mềm được kiểm tra thực sự được xây dựng như thế nào. Đó là lý do mà có nhiều trường hợp mà một kiểm thử viên hộp đen viết rất nhiều ca kiểm thử để kiểm tra một thứ gì đó mà đáng lẽ có thể chỉ cần kiểm tra bằng 1 ca kiểm thử duy nhất, và hoặc một số phần của chương trình không được kiểm tra chút nào.

Do vậy, kiểm thử hộp đen có ưu điểm của “một sự đánh giá khách quan”, mặt khác nó lại có nhược điểm của “thăm dò mù”.

1.1.4.2. *Kiểm thử hộp trắng – White box.*

Là một chiến lược kiểm thử khác, trái ngược hoàn toàn với kiểm thử hộp đen, kiểm thử hộp trắng hay kiểm thử hướng logic cho phép bạn khảo sát cấu trúc bên trong của chương trình. Chiến lược này xuất phát từ dữ liệu kiểm thử bằng sự kiểm thử tính logic của chương trình. Kiểm thử viên sẽ truy cập vào cấu trúc dữ liệu và giải thuật bên trong chương trình (và cả mã lệnh thực hiện chúng).

Các phương pháp kiểm thử hộp trắng.

- Kiểm thử giao diện lập trình ứng dụng – *API testing(application programming interface)*: là phương pháp kiểm thử của ứng dụng sử dụng các API công khai và riêng tư.
- Bao phủ mã lệnh – *Code coverage*: tạo các kiểm tra để đáp ứng một số tiêu chuẩn về bao phủ mã lệnh.
- Các phương pháp gán lỗi – *Fault injection*.
- Các phương pháp kiểm thử hoán chuyển – *Mutation testing methods*.
- Kiểm thử tĩnh - *Static testing*: kiểm thử hộp trắng bao gồm mọi kiểm thử tĩnh.

Phương pháp kiểm thử hộp trắng cũng có thể được sử dụng để đánh giá sự hoàn thành của một bộ kiểm thử mà được tạo cùng với các phương pháp kiểm thử hộp đen. Điều này cho phép các nhóm phần mềm khảo sát các phần của 1 hệ thống ít khi được kiểm tra và đảm bảo rằng những điểm chức năng quan trọng nhất đã được kiểm tra.

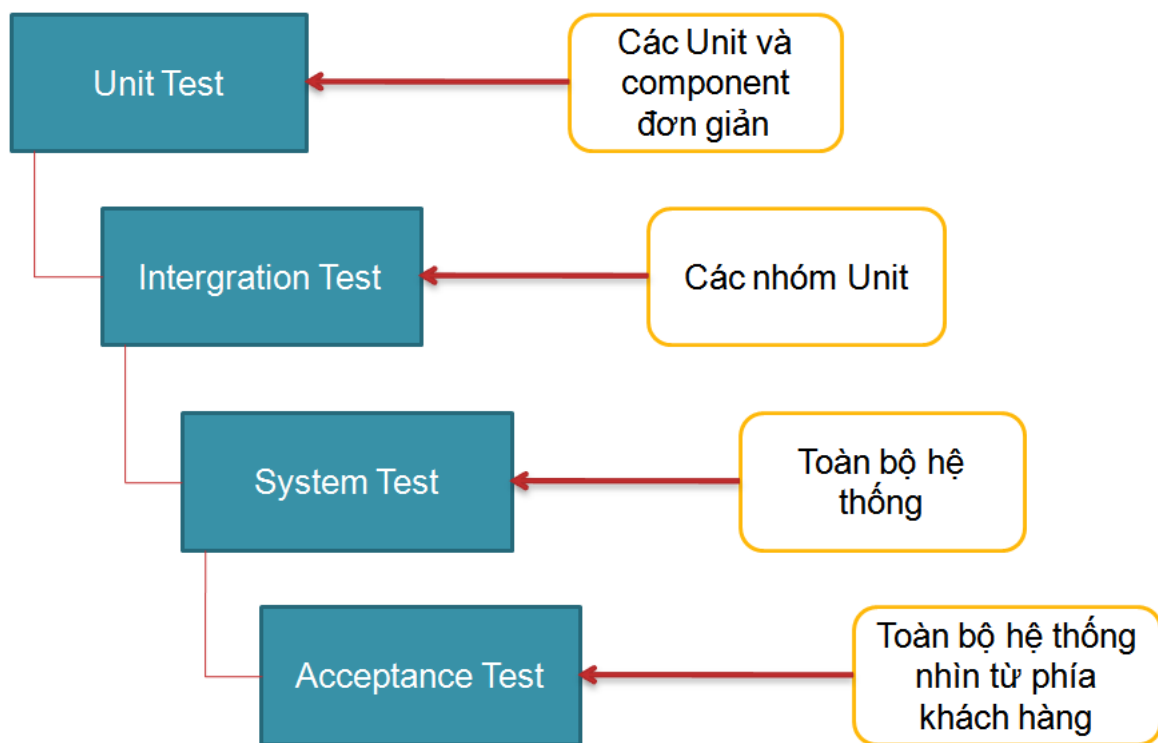
1.1.4.3. *Kiểm thử hộp xám – Gray box testing*

Kiểm thử hộp xám đòi hỏi phải có sự truy cập tới cấu trúc dữ liệu và giải thuật bên trong cho những mục đích thiết kế các ca kiểm thử, nhưng là kiểm thử ở mức người sử dụng hay mức hộp đen. Việc thao tác tới dữ liệu đầu vào và định dạng dữ liệu đầu ra là không rõ ràng, giống như một chiếc “*hộp xám*”, bởi vì đầu vào và đầu ra rõ ràng là ở bên ngoài “*hộp đen*” mà chúng ta vẫn gọi về hệ thống được kiểm tra. Sự khác biệt này đặc

biệt quan trọng khi quản lý kiểm thử tích hợp – *Intergartion testing* giữa 2 modul mã lệnh được viết bởi hai chuyên viên thiết kế khác nhau, trong đó chỉ giao diện là được đưa ra để kiểm thử. Kiểm thử hộp xám có thể cũng bao gồm cả thiết kế đối chiếu để quyết định, ví dụ, giá trị biên hay thông báo lỗi.

1.1.5. Các cấp độ kiểm thử trong kiểm thử phần mềm.

Kiểm thử phần mềm gồm có các cấp độ: Kiểm thử đơn vị, Kiểm thử tích hợp, Kiểm thử hệ thống và Kiểm thử chấp nhận sản phẩm.



Hình 1: Sơ đồ các cấp độ kiểm thử.

1.1.5.1. Kiểm thử đơn vị - Unit test.

a. Định nghĩa

Một đơn vị là một thành phần phần mềm nhỏ nhất mà ta có thể kiểm thử được. Ví dụ, các hàm (*Function*), thủ tục (*Procedure*), lớp (*Class*) hay phương thức (*Method*) đều có thể được xem là Unit.

Vì Unit được chọn để kiểm tra thường có kích thước nhỏ và chức năng hoạt động đơn giản, chúng ta không khó khăn gì trong việc tổ chức kiểm thử, ghi nhận và phân tích kết quả kiểm thử. Nếu phát hiện lỗi, việc xác định nguyên nhân và khắc phục cũng tương đối dễ dàng vì chỉ khoanh vùng trong một đơn thể Unit đang kiểm tra. Một nguyên lý đúc kết từ thực tiễn: thời gian tốn cho Unit Test sẽ được đền bù bằng việc tiết kiệm rất nhiều thời gian, chi phí cho việc kiểm thử và sửa lỗi ở các mức kiểm thử sau đó.

Unit Test thường do lập trình viên thực hiện. Công đoạn này cần được thực hiện càng sớm càng tốt trong giai đoạn viết code và xuyên suốt chu kỳ phát triển phần mềm. Thông thường, Unit Test đòi hỏi kiểm thử viên có kiến thức về thiết kế và code của chương trình

b. Mục đích.

- Đảm bảo thông tin được xử lý và xuất ra là chính xác.
- Trong mỗi tương quan với dữ liệu nhập và chứa năng của Unit.
- Đòi hỏi tất cả các nhánh bên trong phải được kiểm tra phát hiện nhánh sinh lỗi (nhánh đó thường là câu lệnh được thực thi trong một Unit). Ví dụ: chuỗi sau câu lệnh if ... then ...else là một nhánh, đòi hỏi có kỹ thuật, đôi khi dùng thuật toán để chọn lựa
- Phát hiện ra các vấn đề tiềm ẩn hoặc các lỗi kỹ thuật.

c. Yêu cầu.

- Muốn làm được Unit testing thì phải chuẩn bị trước các ca kiểm thử (*Test case*) hoặc các kịch bản kiểm thử (*Test Script*) trong đó phải ghi rõ dữ liệu nhập vào, các bước thực hiện và dữ liệu mong chờ đầu ra của từng testcase.
- Các testcase hay script phải được giữ lại để tái sử dụng.

1.1.5.2. *Kiểm thử tích hợp – Intergration test.*

Integration test là kết hợp các thành phần của một ứng dụng và kiểm thử như một ứng dụng đã hoàn thành. Trong khi Unit Test kiểm tra các thành phần và Unit riêng lẻ thì Intgration Test kết hợp chúng lại với nhau và kiểm tra sự giao tiếp giữa chúng.

Hai mục tiêu chính của Integration Test:

- Phát hiện lỗi giao tiếp giữa các Unit.

- Tích hợp các Unit đơn lẻ thành các hệ thống nhỏ (*Subsystem*) và cuối cùng là nguyên hệ thống hoàn chỉnh (*System*) chuẩn bị cho kiểm thử ở mức hệ thống (*System Test*).

Trong Unit Test, lập trình viên cố gắng phát hiện lỗi liên quan đến chức năng và cấu trúc nội tại của Unit. Có một số phép kiểm thử đơn giản trên giao tiếp giữa Unit với các thành phần liên quan khác, tuy nhiên mọi giao tiếp liên quan đến Unit chỉ thật sự được kiểm tra đầy đủ khi các Unit tích hợp với nhau trong khi thực hiện Integration Test.

Trừ một số ít ngoại lệ, Integration Test chỉ nên thực hiện trên những Unit đã được kiểm tra cẩn thận trước đó bằng Unit Test, và tất cả các lỗi mức Unit đã được sửa chữa. Một số người hiểu sai rằng Unit một khi đã qua giai đoạn Unit Test với các giao tiếp giả lập thì không cần phải thực hiện Integration Test nữa. Thực tế việc tích hợp giữa các Unit dẫn đến những tình huống hoàn toàn khác. Một chiến lược cần quan tâm trong Integration Test là nên tích hợp dần từng Unit. Một Unit tại một thời điểm được tích hợp vào một nhóm các Unit khác đã tích hợp trước đó và đã hoàn tất các đợt Integration Test trước đó. Lúc này, ta chỉ cần kiểm thử giao tiếp của Unit mới thêm vào với hệ thống các Unit đã tích hợp trước đó, điều này sẽ làm cho số lượng can kiểm thử giảm đi rất nhiều, và sai sót sẽ giảm đáng kể.

Có 4 loại kiểm thử trong Integration Test:

- **Kiểm thử cấu trúc (*Structure Test*):** Tương tự White Box Test, kiểm thử cấu trúc nhằm bảo đảm các thành phần bên trong của một chương trình chạy đúng và chú trọng đến hoạt động của các thành phần cấu trúc nội tại của chương trình chẳng hạn các câu lệnh và nhánh bên trong.
- **Kiểm thử chức năng (*Functional Test*):** Tương tự Black Box Test, kiểm thử chức năng chỉ chú trọng đến chức năng của chương trình, mà không quan tâm đến cấu trúc bên trong, chỉ khảo sát chức năng của chương trình theo yêu cầu kỹ thuật.
- **Kiểm thử hiệu năng (*Performance Test*):** Kiểm thử việc vận hành của hệ thống.
- **Kiểm thử khả năng chịu tải (*Stress Test*):** Kiểm thử các giới hạn của hệ thống.

1.1.5.3. *Kiểm thử hệ thống – System test.*

Mục đích System Test là kiểm thử thiết kế và toàn bộ hệ thống (sau khi tích hợp) có thỏa mãn yêu cầu đặt ra hay không.

System Test bắt đầu khi tất cả các bộ phận của phần mềm đã được tích hợp thành công. Thông thường loại kiểm thử này tốn rất nhiều công sức và thời gian. Trong nhiều trường hợp, việc kiểm thử đòi hỏi một số thiết bị phụ trợ, phần mềm hoặc phần cứng đặc thù, đặc biệt là các ứng dụng thời gian thực, hệ thống phân bố, hoặc hệ thống nhúng. Ở mức độ hệ thống, người kiểm thử cũng tìm kiếm các lỗi, nhưng trọng tâm là đánh giá về hoạt động, thao tác, sự tin cậy và các yêu cầu khác liên quan đến chất lượng của toàn hệ thống.

Điểm khác nhau then chốt giữa Integration Test và System Test là System Test chú trọng các hành vi và lỗi trên toàn hệ thống, còn Integration Test chú trọng sự giao tiếp giữa các đơn thể hoặc đối tượng khi chúng làm việc cùng nhau. Thông thường ta phải thực hiện Unit Test và Integration Test để bảo đảm mọi Unit và sự tương tác giữa chúng hoạt động chính xác trước khi thực hiện System Test.

Sau khi hoàn thành Integration Test, một hệ thống phần mềm đã được hình thành cùng với các thành phần đã được kiểm tra đầy đủ. Tại thời điểm này, lập trình viên hoặc kiểm thử viên bắt đầu kiểm thử phần mềm như một hệ thống hoàn chỉnh. Việc lập kế hoạch cho System Test nên bắt đầu từ giai đoạn hình thành và phân tích các yêu cầu.

System Test kiểm thử cả các hành vi chức năng của phần mềm lẫn các yêu cầu về chất lượng như độ tin cậy, tính tiện lợi khi sử dụng, hiệu năng và bảo mật. Mức kiểm thử này đặc biệt thích hợp cho việc phát hiện lỗi giao tiếp với phần mềm hoặc phần cứng bên ngoài, chẳng hạn các lỗi "tắc nghẽn" (deadlock) hoặc chiếm dụng bộ nhớ. Sau giai đoạn System Test, phần mềm thường đã sẵn sàng cho khách hàng hoặc người dùng cuối cùng kiểm thử chấp nhận sản phẩm (*Acceptance Test*) hoặc dùng thử (*Alpha/Beta Test*).

Đòi hỏi nhiều công sức, thời gian và tính chính xác, khách quan, System Test thường được thực hiện bởi một nhóm kiểm thử viên hoàn toàn độc lập với nhóm phát triển dự án. Bản thân System Test lại gồm nhiều loại kiểm thử khác nhau, phổ biến nhất gồm:

- **Kiểm thử chức năng (Functional Test):** Bảo đảm các hành vi của hệ thống thỏa mãn đúng yêu cầu thiết kế.
- **Kiểm thử hiệu năng (Performance Test):** Bảo đảm tối ưu việc phân bổ tài nguyên hệ thống (ví dụ bộ nhớ) nhằm đạt các chỉ tiêu như thời gian xử lý hay đáp ứng câu truy vấn...
- **Kiểm thử khả năng chịu tải (Stress Test hay Load Test):** Bảo đảm hệ thống vận hành đúng dưới áp lực cao (ví dụ nhiều người truy xuất cùng lúc). Stress Test tập trung vào các trạng thái tới hạn, các "điểm chết", các tình huống bất thường như đang giao dịch thì ngắt kết nối (xuất hiện nhiều trong kiểm tra thiết bị như POS, ATM...)...
- **Kiểm thử cấu hình (Configuration Test).**
- **Kiểm thử bảo mật (Security Test):** Bảo đảm tính toàn vẹn, bảo mật của dữ liệu và của hệ thống.
- **Kiểm thử khả năng phục hồi (Recovery Test):** Bảo đảm hệ thống có khả năng khôi phục trạng thái ổn định trước đó trong tình huống mất tài nguyên hoặc dữ liệu; đặc biệt quan trọng đối với các hệ thống giao dịch như ngân hàng trực tuyến...

Kết luận: Không nhất thiết phải thực hiện tất cả các kiểm thử trên mà phụ thuộc vào yêu cầu hệ thống, tùy vào khả năng và thời gian của dự án, khi lập kế hoạch thì người quản lý sẽ quy định kiểm thử theo những loại nào.

1.1.5.4. *Kiểm thử chấp nhận – Acceptance test.*

Thông thường, sau giai đoạn System Test là Acceptance Test, được khách hàng thực hiện (hoặc ủy quyền cho một nhóm thứ ba thực hiện). Mục đích của Acceptance Test là để chứng minh phần mềm thỏa mãn tất cả yêu cầu của khách hàng và khách hàng chấp nhận sản phẩm (và trả tiền thanh toán hợp đồng).

Acceptance Test có ý nghĩa hết sức quan trọng, mặc dù trong hầu hết mọi trường hợp, các phép kiểm thử của System Test và Acceptance Test gần như tương tự, nhưng bản chất và cách thức thực hiện lại rất khác biệt.

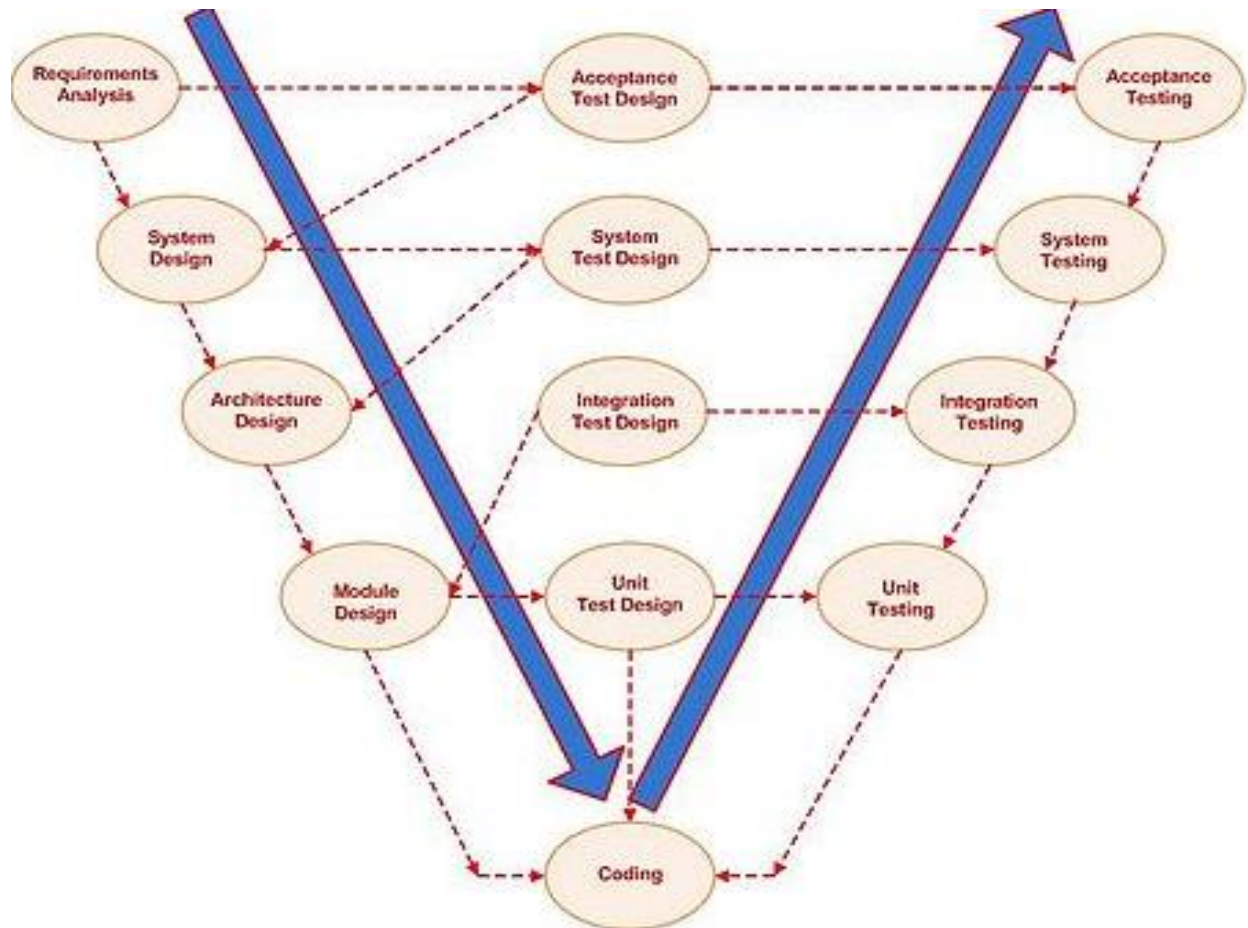
Đối với những sản phẩm dành bán rộng rãi trên thị trường cho nhiều người sử dụng, thông thường sẽ thông qua hai loại kiểm thử gọi là kiểm thử Alpha – **Alpha Test** và kiểm

thử Beta – **Beta Test**. Với Alpha Test, người dùng kiểm thử phần mềm ngay tại nơi phát triển phần mềm, lập trình viên sẽ ghi nhận các lỗi hoặc phản hồi, và lên kế hoạch sửa chữa. Với Beta Test, phần mềm sẽ được gửi tới cho người dùng để kiểm thử ngay trong môi trường thực, lỗi hoặc phản hồi cũng sẽ gửi ngược lại cho lập trình viên để sửa chữa.

Thực tế cho thấy, nếu khách hàng không quan tâm và không tham gia vào quá trình phát triển phần mềm thì kết quả Acceptance Test sẽ sai lệch rất lớn, mặc dù phần mềm đã trải qua tất cả các kiểm thử trước đó. Sự sai lệch này liên quan đến việc hiểu sai yêu cầu cũng như sự mong chờ của khách hàng. Ví dụ đôi khi một phần mềm xuất sắc vượt qua các phép kiểm thử về chức năng thực hiện bởi nhóm thực hiện dự án, nhưng khách hàng khi kiểm thử sau cùng vẫn thất vọng vì bố cục màn hình nghèo nàn, thao tác không tự nhiên, không theo tập quán sử dụng của khách hàng v.v...

Gắn liền với giai đoạn Acceptance Test thường là một nhóm những dịch vụ và tài liệu đi kèm, phổ biến như hướng dẫn cài đặt, sử dụng v.v... Tất cả tài liệu đi kèm phải được cập nhật và kiểm thử chặt chẽ.

1.1.5.5. *Mô hình chữ V trong kiểm thử phần mềm*



Hình 2: Mô hình chữ V

- Các bước tiến hành của mô hình chữ V

- Sau khi đã có yêu cầu của khách hàng ta thực hiện đồng thời việc thiết kế hệ thống và bản kiểm thử cho người dùng (user acceptance testing) dựa trên các yêu cầu đó.
- Khi hoàn thành được bản thiết kế hệ thống, ta vừa thực hiện bảng kiểm thử hệ thống (system testing) và vừa làm thiết kế kiến trúc phần mềm.
- Sau khi có được thiết kế kiến trúc ta chuyển sang thiết kế các module. Từ các thiết kế module ta vừa làm bản thiết kế các unit test đồng thời bắt đầu coding.
- Sau giai đoạn coding thì các công đoạn sau bao gồm **unit test**, **integration test**, **system test** và **acceptance testing** được thực hiện lần lượt dựa trên các thiết kế đã thực hiện sẵn trước đó trong giai đoạn phát triển phần mềm ban đầu.

- Ưu điểm trong mô hình chữ V

- Có thể làm 1 số việc song song. Ví dụ : Nếu làm yêu cầu đúng thì có thể làm song song với việc thiết kế test .
 - Đạt được phần mềm chất lượng, các pha tương thích với nhau, hỗ trợ cho nhau.
 - Các hoạt động kiểm thử được chú trọng và thực hiện song song với các hoạt động liên quan đến đặc tả yêu cầu và thiết kế. Hay nói cách khác, mô hình này khuyến khích các hoạt động liên quan đến kế hoạch kiểm thử được tiến hành sớm trong chu kỳ phát triển, không phải đợi đến lúc kết thúc giai đoạn hiện thực
- **Nhược điểm trong mô hình chữ V**
 - Đòi hỏi tất cả yêu cầu phần mềm phải được xác định rõ ràng ngay từ đầu dự án.
 - Pha sau thực chỉ được thực hiện khi pha trước kết thúc, không thể quay ngược trở lại pha trước.
 - Người sử dụng không có cơ hội tham gia trong suốt thời gian của các giai đoạn trung gian từ thiết kế cho đến kiểm thử

1.1.6. Một số cấp độ kiểm thử khác.

Ngoài các cấp độ trên, còn một số cấp độ kiểm thử khác như:

- **Kiểm thử hồi quy – Regression Testing:**

Theo chuẩn IEEE610.12-90, kiểm thử hồi quy là “sự kiểm tra lại có lựa chọn của một hệ thống hay thành phần để xác minh là những sự thay đổi không gây ra những hậu quả không mong muốn...”. Trên thực tế, quan niệm này là chỉ ra rằng phần mềm mà đã qua được các kiểm tra trước đó vẫn có thể được kiểm tra lại. Beizer định nghĩa đó là sự lặp lại các kiểm tra để chỉ ra rằng cách hoạt động của phần mềm không bị thay đổi, ngoại trừ tới mức như yêu cầu. Hiển nhiên là sự thỏa hiệp phải được thực hiện giữa sự đảm bảo được đưa ra bởi kiểm thử hồi quy mỗi lần thực hiện một sự thay đổi và những tài nguyên được yêu cầu thực hiện điều đó.

- **Kiểm thử tính đúng – Correctness testing:**

Tính đúng đắn là yêu cầu tối thiểu của phần mềm, là mục đích chủ yếu của kiểm thử. Kiểm thử tính đúng sẽ cần một kiểu người đáng tin nào đó, để chỉ ra cách hoạt động đúng đắn từ cách hoạt động sai lầm. Kiểm thử viên có thể biết hoặc không biết các chi tiết bên

trong của các modun phần mềm được kiểm thử, ví dụ luồng điều khiển, luồng dữ liệu,... Do đó, hoặc là quan điểm hộp trắng, hoặc là quan điểm hộp đen có thể được thực hiện trong kiểm thử phần mềm.

1.2. Nguyên tắc trong kiểm thử phần mềm.

Để kiểm thử đạt hiệu quả thì khi tiến hành kiểm thử phần mềm cần phải tuân thủ một số quy tắc sau:

Quy tắc 1: Một phần quan trọng của 1 ca kiểm thử là định nghĩa của đầu ra hay kết quả mong muốn.

Quy tắc 2: Lập trình viên nên tránh tự kiểm tra chương trình của mình.

Quy tắc 3: Nhóm lập trình không nên kiểm thử chương trình của chính họ.

Quy tắc 4: Kiểm tra thấu đáo mọi kết quả của mỗi kiểm tra.

Quy tắc 5: Các ca kiểm thử phải được viết cho các trạng thái đầu vào không hợp lệ và không mong muốn, cũng như cho các đầu vào hợp lệ và mong muốn.

Quy tắc 6: Khảo sát 1 chương trình để xem liệu chương trình có thực hiện cái mà nó cần thực hiện chỉ là 1 phần, phần còn lại là xem liệu chương trình có thực hiện cái mà nó không cần phải thực hiện hay không.

Quy tắc 7: Tránh các ca kiểm thử băng quơ trừ khi chương trình thực sự là 1 chương trình băng quơ.

Quy tắc 8: Không dự kiến kết quả của kiểm thử theo giả thiết ngầm là không tìm thấy lỗi.

Quy tắc 9: Xác suất tồn tại lỗi trong 1 đoạn chương trình là tương ứng với số lỗi đã tìm thấy trong đoạn đó.

Quy tắc 10: Kiểm thử là 1 nhiệm vụ cực kỳ sáng tạo và có tính thử thách trí tuệ.

CHƯƠNG II: UNIT TESTING

2.1. Tổng quan về Unit test

2.1.1. Định nghĩa về Unit testing.

- Một Unit là một thành phần nhỏ nhất mà ta có thể kiểm tra được. Vì vậy, các hàm(*function*), thủ tục (*Proceduce*), các lớp (*Class*) hoặc các phương thức (*Method*) đều có thể coi là một Unit. Vì Unit được chọn để kiểm thử thường có kích thước nhỏ và đơn giản.

- Kiểm thử đơn vị sẽ được thực hiện đối với một hệ thống được kiểm thử(SUT)

- SUT(System Under Test) là hệ thống được kiểm thử và một số người thích sử dụng CUT (class under test – lớp theo kiểm thử; code under test – mã theo kiểm thử). Khi kiểm thử một cái gì đó thì có thể sử dụng một cái như SUT.

- Đặc điểm của một Unit test: đóng vai trò như người sử dụng đầu tiên của hệ thống. Chỉ có giá trị khi chúng có thể phát hiện các vấn đề tiềm ẩn hay lỗi kỹ thuật.

- Ứng dụng của Unit test: kiểm tra được tất cả các hàm, thủ tục, sự kiện, thuộc tính; kiểm tra các trạng thái và ràng buộc của đối tượng ở mức sâu hơn, nơi mà không thể truy cập được; kiểm tra được các quá trình và các khung làm việc(work flow).

2.1.2. Mục đích

- Đảm bảo thông tin xử lý và xuất ra là chính xác.
- Trong mối tương quan với dữ liệu nhập và chức năng của một Unit.
- Đòi hỏi tất cả các nhánh bên trong phải được kiểm tra phát hiện nhánh sinh lỗi(nhánh đó thường là câu lệnh để thực thi trong một Unit). Ví dụ: Chuỗi câu lệnh If ...then ...else là một nhánh. Đòi hỏi có kỹ thuật, đôi khi dùng thuật toán để chọn lựa.
- Phát hiện ra các vấn đề tiềm ẩn hoặc các lỗi kỹ thuật.

2.1.3. Yêu cầu.

- Muốn làm được Unit testing thì phải chuẩn bị trước các ca kiểm thử (Test case), các kịch bản kiểm thử (Test script, trong đó phải ghi rõ dữ liệu nhập vào, các bước thực hiện một ca kiểm thử hay một kịch bản kiểm thử và dữ liệu mong chờ đầu ra.
- Các Test case, Test script được giữ lại để tái sử dụng.

2.1.4. Người thực hiện Unit test.

Khi thực hiện kiểm thử mức đơn vị thì công đoạn này là do người lập trình viên (*Developer*) hay kỹ sư kiểm thử (*Test Engineer*). Vì khi xây dựng được một test case để có thể tìm ra được lỗi là nhiều nhất thì người thực hiện phải biết về ngôn ngữ lập trình, có khả năng đọc và hiểu các đoạn code.

2.1.5. Vòng đời của một Unit test.

Có ba trạng thái cơ bản:

- Trạng thái lỗi (Fail status).
- Trạng thái tạm ngừng thực hiện (Ignore status).
- Trạng thái làm việc (Pass Status).

Chú ý: Mỗi trạng thái của Unit test được thể hiện bằng một màu khác nhau:

- Fail: màu đỏ
- Ignore: màu vàng
- Pass: màu xanh.

Như vậy: Unit test chỉ thực sự mang lại hiệu quả khi:

- Được vận hành nhiều lần.
- Tự động hoàn toàn.
- Độc lập với các Unit khác.

2.1.6. Lợi ích của Unit test.

- Tạo ra môi trường lý tưởng để kiểm tra bất cứ đoạn mã nào, có khả năng thăm dò và phát hiện lỗi chính xác, duy trì sự ổn định của toàn bộ phần mềm và giúp tiết kiệm thời gian so với công việc gỡ rối truyền thống.
- Phát hiện thuật toán thực thi không hiệu quả, các thủ tục chạy vượt quá giới hạn thời gian.
- Phát hiện các vấn đề thiết kế, xử lý hệ thống, các mô hình thiết kế.
- Tạo hàng rào an toàn cho các khối mã. Bất cứ sự thay đổi nào cũng có thể tác động tới hàng rào này và thông báo những nguy hiểm tiềm tàng.

- Phát hiện lỗi nghiêm trọng có thể xảy ra trong những tình huống hẹp.

Như vậy: Unit test là một môi trường lý tưởng để tiếp cận API bên ngoài một cách tốt nhất. Sẽ rất nguy hiểm nếu chúng ta ứng dụng ngay các thư viện này mà không kiểm tra kỹ lưỡng công dụng của các thủ tục trong thư viện. Dành thời gian viết các Unit test kiểm tra từng thủ tục là phương pháp tốt nhất khẳng định sự hiệu đúng đắn về cách sử dụng thư viện đó. Ngoài ra, Unit test cũng được sử dụng để phát hiện sự khác biệt giữa phiên bản mới và phiên bản cũ của cùng một thư viện.

2.1.7. Tác dụng của Unit test.

- Giải phóng chuyên viên QA (*Quality Assurance*) khỏi các công việc phức tạp.
- Tăng sự tự tin khi hoàn thành một công việc. Chúng ta thường có cảm giác không chắc chắn về các đoạn mã của mình như liệu các lỗi có quay lại không, hoạt động của module hiện hành có bị tác động không hoặc liệu công việc hiệu chỉnh mã có gây hư hỏng không...
- Là công cụ để đánh giá năng lực của bạn. Số lượng các tình huống kiểm tra(*test case*) chuyển trạng thái “*pass*” sẽ thể hiện tốc độ làm việc, năng suất của bạn.

2.1.8. Chiến lược viết mã hiệu quả với Unit test.

- Phân tích các tình huống có thể xảy ra đối với mã. Đừng bỏ qua các tình huống tồi tệ nhất có thể xảy ra, thí dụ dữ liệu nhập làm một kết nối cơ sở dữ liệu thất bại, ứng dụng bị treo vì một phép toán chia cho không, các thủ tục đưa ra lỗi ngoại lệ sai có thể phá hỏng ứng dụng một cách bí ẩn...
- Mọi Unit test phải bắt đầu với trạng thái “*fail*” và chuyển trạng thái “*pass*” sau một số thay đổi hợp lý đối với mã chính.
- Mỗi khi viết một đoạn mã quan trọng, hãy viết các Unit test tương ứng cho đến khi bạn không thể nghĩ thêm tình huống nào nữa.
- Nhập số lượng đủ lớn các giá trị đầu vào để phát hiện các điểm sau:
 - Nhập giá trị hợp lệ → kết quả trả về cũng hợp lệ.
 - Nhập giá trị không hợp lệ → kết quả trả về cũng không hợp lệ.

- Sớm nhận biết các đoạn mã không ổn định và có nguy cơ gây lỗi cao, viết Unit test tương ứng để không chề.
- Ứng với mỗi đối tượng nghiệp vụ (business object) hoặc đối tượng truy cập dữ liệu (data access object), nên tạo ra một lớp kiểm tra riêng vì những lỗi nghiêm trọng có thể phát sinh từ các đối tượng này.
- Để ngăn chặn các lỗi có thể phát sinh trở lại thực thi tự động tất cả Unit test mỗi khi có một sự thay đổi quan trọng, hãy làm công việc này mỗi ngày. Các Unit test lỗi cho chúng ta biết thay đổi nào là nguyên nhân gây lỗi.
- Để tăng hiệu quả và giảm rủi ro khi viết các Unit test, cần sử dụng nhiều phương thức kiểm tra khác nhau. Hãy viết càng đơn giản càng tốt.
- Cuối cùng, viết Unit test cũng đòi hỏi sự nỗ lực, kinh nghiệm và sự sáng tạo như viết phần mềm.

Unit test chỉ thực sự mang lại lợi ích nếu chúng ta đặt vấn đề chất lượng phần mềm lên hàng đầu hơn là chỉ nhằm kết thúc công việc đúng thời hạn

2.2. Sử dụng Unit test với mô hình đối tượng ảo (Mock Object)

Trong Unit Test, mỗi một đối tượng hay một phương thức riêng lẻ được kiểm tra tại một thời điểm và chúng ta chỉ quan tâm đến các trách nhiệm của chúng có được thực hiện đúng hay không. Tuy nhiên trong các dự án phần mềm phức tạp thì Unit Test không còn là quy trình riêng lẻ, nhiều đối tượng (đơn vị chương trình) không làm việc độc lập mà tương tác với các đối tượng khác như kết nối mạng, cơ sở dữ liệu hay dịch vụ web. Như vậy công việc kiểm nghiệm có thể bị trì hoãn gây tác động xấu đến quy trình phát triển chung. Để giải quyết các vấn đề này người ta đưa ra mô hình “Mock Object” hay đối tượng ảo (hoặc đối tượng giả).

2.2.1. Định nghĩa

Mock object (MO) là một đối tượng ảo mô phỏng các tính chất và hành vi giống hệt như đối tượng thực được truyền vào bên trong khối mã đang vận hành nhằm kiểm tra tính đúng đắn của các hoạt động bên trong.

2.2.2. Đặc điểm

- Đơn giản hơn đối tượng thực nhưng vẫn giữ được sự tương tác với các đối tượng khác.
- Không lặp lại nội dung đối tượng thực.
- Cho phép thiết lập các trạng thái riêng trợ giúp kiểm tra.

2.2.3. Lợi ích

- Đảm bảo công việc kiểm nghiệm không bị gián đoạn bởi các yếu tố bên ngoài, giúp các chuyên viên tập trung vào một chức năng nghiệp vụ cụ thể, từ đó tạo ra UT vận hành nhanh hơn.
- Giúp tiếp cận hướng đối tượng tốt hơn. Nhờ MO chúng ta có thể phát hiện interface cần tách ở một số lớp.
- Dễ dàng cho việc kiểm nghiệm. Thay vì gọi các đối tượng thực vận hành nặng nề, chúng ta có thể gọi các MO đơn giản hơn để kiểm tra nhanh liên kết giữa các thủ tục, công việc kiểm nghiệm có thể tiến hành nhanh hơn.

2.2.4. Phạm vi sử dụng

Mock Object được sử dụng trong các trường hợp sau:

- Cần lập trạng thái giả của một đối tượng thực trước khi các Unit test có liên quan được đưa vào vận hành (Ví dụ: kết nối cơ sở dữ liệu, giả định trạng thái lỗi server...)
- Cần lập trạng thái cần thiết cho một số tính chất nào đó của đối tượng đã bị khoá quyền truy cập (các biến, thủ tục, hàm, thuộc tính riêng được khai báo private). Không phải lúc nào các tính chất của một đối tượng cũng có thể được mở rộng phạm vi truy cập ra bên ngoài vì điều này có thể trực tiếp phá vỡ liên kết giữa các phương thức theo một trình tự sắp đặt trước, từ đó dẫn đến kết quả có thể bị xử lý sai. Tuy nhiên, Mock Object có thể thiết lập các trạng thái giả mà vẫn đảm bảo các yêu cầu ràng buộc, các nguyên tắc đúng đắn và các quan hệ của đối tượng thực.
- Cần kiểm tra một số thủ tục hoặc các biến của thành viên bị hạn chế truy cập. Bằng cách kế thừa Mock Object từ đối tượng thực chúng ta có thể kiểm tra các thành viên đã được bảo vệ (khai báo protected).

- Cần loại bỏ các hiệu ứng phụ của một đối tượng nào đó không liên quan đến Unit Test.
- Cần kiểm nghiệm mã vận hành có tương tác với hệ thống bên ngoài.

2.2.5. Các đối tượng được mô phỏng.

Mock Object mô phỏng các loại đối tượng sau đây:

- Các đối tượng thực mới chỉ được mô tả trên bản thiết kế nhưng chưa tồn tại dưới dạng mã, hoặc các module chưa sẵn sàng cung cấp các dữ liệu cần thiết để vận hành Unit Test.
- Các đối tượng thực có các thủ tục chưa xác định rõ ràng về mặt nội dung (mới chỉ mô tả trong interface) nhưng được đòi hỏi sử dụng gấp trong các Unit Test.
- Các đối tượng thực rất khó cài đặt (thí dụ đối tượng xử lý các trạng thái của server)
- Các đối tượng thực xử lý một tình huống khó xảy ra. Ví dụ lỗi kết nối mạng, lỗi ổ cứng...
- Các đối tượng có các tính chất và hành vi phức tạp, các trạng thái luôn thay đổi và các quan hệ chặt chẽ với nhiều đối tượng khác
- Các đối tượng vận hành chậm chạp. Công việc kiểm tra hiện hành không liên quan đến thao tác xử lý đối tượng này.
- Đối tượng thực liên quan đến giao diện tương tác người dùng. Không người dùng nào có thể ngồi kiểm nghiệm các chức năng hệ thống hết ngày này qua ngày khác. Tuy nhiên bạn có thể dùng Mock Object để mô phỏng thao tác của người dùng, nhờ đó công việc có thể được diễn biến lặp lại và hoàn toàn tự động.

2.2.6. Thiết kế MO

Thông thường, nếu số lượng Mock Object không nhiều, chúng ta có thể tự thiết kế. Nếu không muốn mất nhiều thời gian tự thiết kế một số lượng lớn Mock Object, bạn có thể tải về các công cụ có sẵn thông dụng hiện nay như EasyMock, jMock, Nmock... Các phần mềm này cung cấp nhiều API cho phép xây dựng Mock Object và các kho dữ liệu giả dễ dàng hơn, cũng như kiểm tra tự động các số liệu trong Unit Test. Nói chung, việc thiết kế Mock Object gồm 3 bước chính sau đây:

- Đưa ra interface để mô tả đối tượng. Tất cả các tính chất và thủ tục quan trọng cần kiểm tra phải được mô tả trong interface.
- Viết nội dung cho đối tượng thực dựa trên interface như thông thường.
- Trích interface từ đối tượng thực và triển khai Mock Object dựa trên interface đó.

Lưu ý Mock Object phải được đưa vào quy trình kiểm nghiệm tách biệt. Cách này có thể sinh ra nhiều interface không thực sự cần thiết, có thể làm cho thiết kế ứng dụng trở nên phức tạp. Một cách làm khác là kế thừa một đối tượng đang tồn tại và cố gắng mô phỏng các hành vi càng đơn giản càng tốt, như trả về một dữ liệu giả chẳng hạn. Đặc biệt tránh tạo ra những liên kết mất xích giữa các Mock Object vì chúng có thể làm cho thiết kế Unit Test trở nên phức tạp.

CHƯƠNG III: THIẾT KẾ TEST CASE

3.1. Định nghĩa.

Thiết kế test – case trong kiểm thử phần mềm là quá trình xây dựng các phương pháp kiểm thử có thể phát hiện lỗi, sai sót, khuyết điểm của phần mềm để xây dựng phần mềm đạt tiêu chuẩn.

3.2. Vai trò của việc thiết kế test case.

- Tạo ra các ca kiểm thử tốt nhất có khả năng phát hiện ra lỗi, sai sót của phần mềm một cách nhiều nhất.
- Tạo ra các ca kiểm thử có chi phí rẻ nhất, đồng thời tốn ít thời gian và công sức nhất.

3.3. Quy trình thiết kế test case.

Mục đích của việc thiết kế test case: là khả năng tìm ra nhiều lỗi nhất.

Nếu mà sử dụng một trong hai phương pháp kiểm thử hộp đen hay kiểm thử hộp trắng là không thể nên ta phải kết hợp cả hai phương pháp.

Ta sử dụng kiểm thử hộp đen để kiểm thử các phương pháp thiết kế và sau đó sử dụng phương pháp kiểm thử hộp trắng để kiểm thử tính logic của chương trình.

Chiến lược kết hợp:

Kiểm thử hộp đen	Kiểm thử hộp trắng
1. Phân lớp tương đương	1. Bao phủ câu lệnh.
2. Phân tích giá trị biên	2. Bao phủ quyết định
3. Đồ thị nguyên nhân – kết quả	3. Bao phủ điều kiện
4. Đoán lỗi	4. Bao phủ điều kiện – quyết định.
	5. Bao phủ đa điều kiện

Quy trình thiết kế test case sẽ bắt đầu bằng việc phát triển các ca kiểm thử sử dụng phương pháp hộp đen và sau đó phát triển bổ sung các ca kiểm thử cần thiết với phương pháp hộp trắng.

3.3.1. Phương pháp kiểm thử hộp đen – Black box testing.

3.3.1.1. Phân vùng tương đương

a. Định nghĩa

Phân lớp tương đương là một phương pháp kiểm thử hộp đen chia miền đầu vào của một chương trình thành các lớp dữ liệu, từ đó suy dẫn ra các ca kiểm thử. Phương pháp này cố gắng xác định ra một ca kiểm thử mà làm lộ ra một lớp lỗi, do đó làm giảm tổng số các trường hợp kiểm thử phải được xây dựng.

Thiết kế ca kiểm thử cho phân lớp tương đương dựa trên sự đánh giá về các lớp tương đương với một điều kiện vào. Lớp tương đương biểu thị cho tập các trạng thái hợp lệ hay không hợp lệ đối với điều kiện vào.

Một cách xác định tập con này là để nhận ra rằng 1 ca kiểm thử được lựa chọn tốt cũng nên có 2 đặc tính khác:

- Giảm thiểu số lượng các ca kiểm thử khác mà phải được phát triển để hoàn thành mục tiêu đã định của kiểm thử “hợp lý”.
- Bao phủ một tập rất lớn các ca kiểm thử có thể khác. Tức là, nó nói cho chúng ta một thứ gì đó về sự có mặt hay vắng mặt của những lỗi qua tập giá trị đầu vào cụ thể.

Thiết kế Test-case bằng phân lớp tương đương tiến hành theo 2 bước:

- Xác định các lớp tương đương.
- Xác định các ca kiểm thử.

b. Xác định các lớp tương đương.

Các lớp tương đương được xác định bằng cách lấy mỗi trạng thái đầu vào (thường là 1 câu hay 1 cụm từ trong đặc tả) và phân chia nó thành 2 hay nhiều nhóm.

Mẫu liệt kê các lớp tương đương:

Điều kiện đầu vào	Các lớp tương đương hợp lệ	Các lớp tương đương không hợp lệ

- Điều kiện đầu vào là một giá trị đặc biệt, mảng số hay chuỗi, tập hợp hay điều kiện đúng sai.

- Các lớp tương đương hợp lệ là mô tả các đầu vào hợp lệ của chương trình
- Các lớp tương đương không hợp lệ là mô tả các trạng thái khác của chương trình như: sai, thiếu, không đúng...

c. Nguyên tắc để xác định lớp tương đương.

- Nếu điều kiện đầu vào định rõ giới hạn của một mảng thì chia vùng tương đương thành 3 tình huống:
 - Xác định một lớp tương đương hợp lệ.
 - Xác định hai lớp tương đương không hợp lệ.
- Nếu điều kiện đầu vào là một giá trị xác định thì chia vùng tương đương thành 3 tình huống:
 - Một lớp tương đương hợp lệ.
 - Hai lớp tương đương không hợp lệ.
- Nếu điều kiện đầu vào chỉ định là một tập giá trị thì chia vùng tương đương thành 2 tình huống như sau:
 - Một lớp tương đương hợp lệ.
 - Một lớp tương đương không hợp lệ.
- Nếu điều kiện đầu vào xác định là một kiểu đúng sai thì chia vùng tương đương thành 2 tình huống:
 - Một lớp tương đương hợp lệ.
 - Một lớp tương đương không hợp lệ.

d. Xác định các ca kiểm thử.

Với các lớp tương đương xác định được ở bước trên, bước thứ hai là sử dụng các lớp tương đương đó để xác định các ca kiểm thử. Quá trình này như sau:

1. Gán 1 số duy nhất cho mỗi lớp tương đương.

2. Cho đến khi tất cả các lớp tương đương hợp lệ được bao phủ bởi (hợp nhất thành) các ca kiểm thử. Viết 1 ca kiểm thử mới bao phủ càng nhiều các lớp tương đương đó chưa được bao phủ càng tốt.
3. Cho đến khi các ca kiểm thử của bạn đã bao phủ tất cả các lớp tương đương không hợp lệ. Viết 1 ca kiểm thử mà bao phủ một và chỉ một trong các lớp tương đương không hợp lệ chưa được bao phủ.
4. Lý do mà mỗi ca kiểm thử riêng bao phủ các trường hợp không hợp lệ là vì các kiểm tra đầu vào không đúng nào đó che giấu hoặc thay thế các kiểm tra đầu vào không đúng khác.

Mặc dù việc phân lớp tương đương là rất tốt khi lựa chọn ngẫu nhiên các ca kiểm thử, nhưng nó vẫn có những thiếu sót. Ví dụ, nó bỏ qua các kiểu test – case có lợi nào đó. Hai phương pháp tiếp theo, phân tích giá trị biên và đồ thị nguyên nhân – kết quả, bao phủ được nhiều những thiếu sót này.

e. Ví dụ về phân lớp tương đương.

Cho bài toán như sau:

User:
Password:

Yêu cầu: Thiết kế test case sao cho khi người dùng nhập user vào ô text thì chỉ cho nhập số ký tự [6 – 20].

Bài làm

Do yêu cầu của bài toán chỉ cho phép nhập số ký tự vào trong khi nhập của user nằm [6 - 20] nên ta có tình huống kiểm thử sau:

- Nhập vào một trường hợp hợp lệ: nhập 7 ký tự.
- Nhập vào trường hợp không hợp lệ thứ nhất: nhập 5 ký tự.
- Nhập vào trường hợp không hợp lệ thứ hai: nhập vào 21 ký tự.
- Trường hợp đặc biệt: không nhập gì vào ô text đó (để trống).

Lập bảng các lớp tương đương:

Điều kiện đầu vào	Các lớp tương đương hợp lệ	Các lớp tương đương không hợp lệ
Cho phép nhập số ký tự nằm [6 – 20]	- Nhập vào 7 ký tự	- Nhập vào 5 ký tự - Nhập vào 21 ký tự - Để trống ô đó

3.3.1.2. Phân tích giá trị biên – Boundary Values Analysis

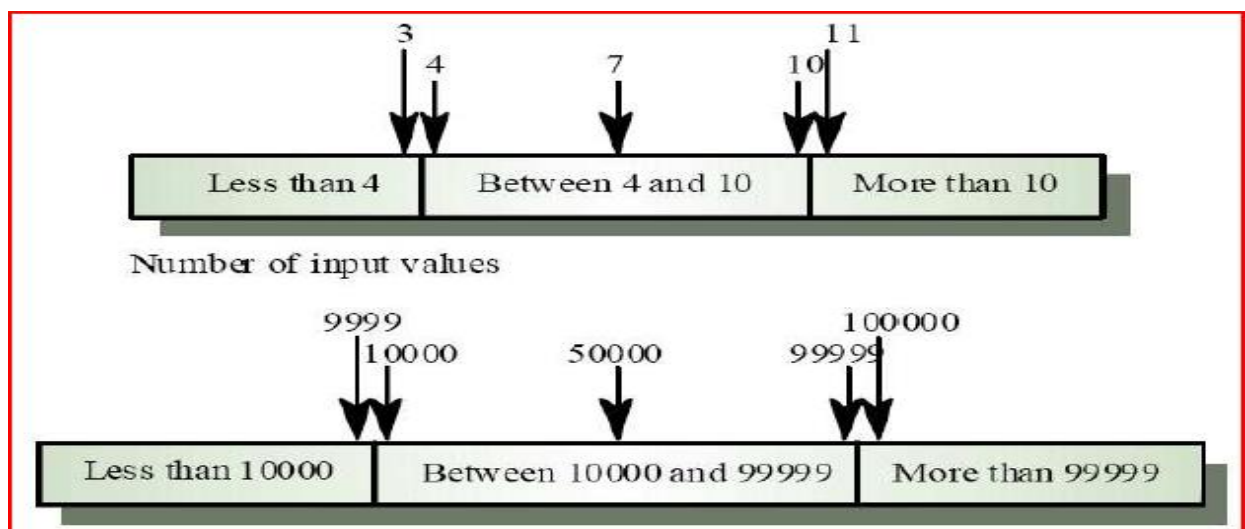
a. Định nghĩa.

Phân tích giá trị biên (*BVA*) là kỹ thuật thiết kế test case và hoàn thành phân vùng tương đương.

Mục tiêu là lựa chọn các test case để thực thi giá trị biên.

Kiểm thử các dữ liệu vào bao gồm:

- Giá trị nhỏ nhất.
- Giá trị gần kề lớn hơn giá trị nhỏ nhất.
- Giá trị bình thường.
- Giá trị gần kề bé hơn giá trị lớn nhất.
- Giá trị lớn nhất.



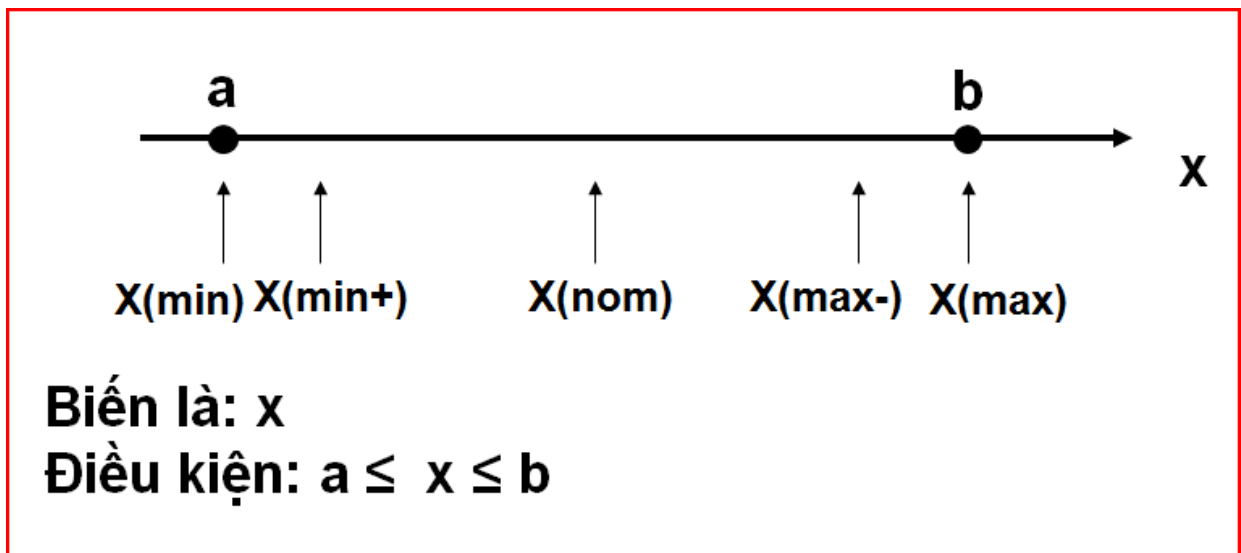
Hình 3: Ví dụ về biểu thị phân tích giá trị biên

b. Nguyên tắc chọn dữ liệu.

- Nếu giá trị đầu vào xác định là một mảng có biên là a và b ($a < b$) thì có thể thiết kế được các test case như sau:

- Biên a
- Biên b
- Giá trị nhỏ hơn biên a
- Giá trị lớn hơn biên b
- Một giá trị nằm giữa a và b.

➤ *Kiểm thử theo giá trị biên với một biến*

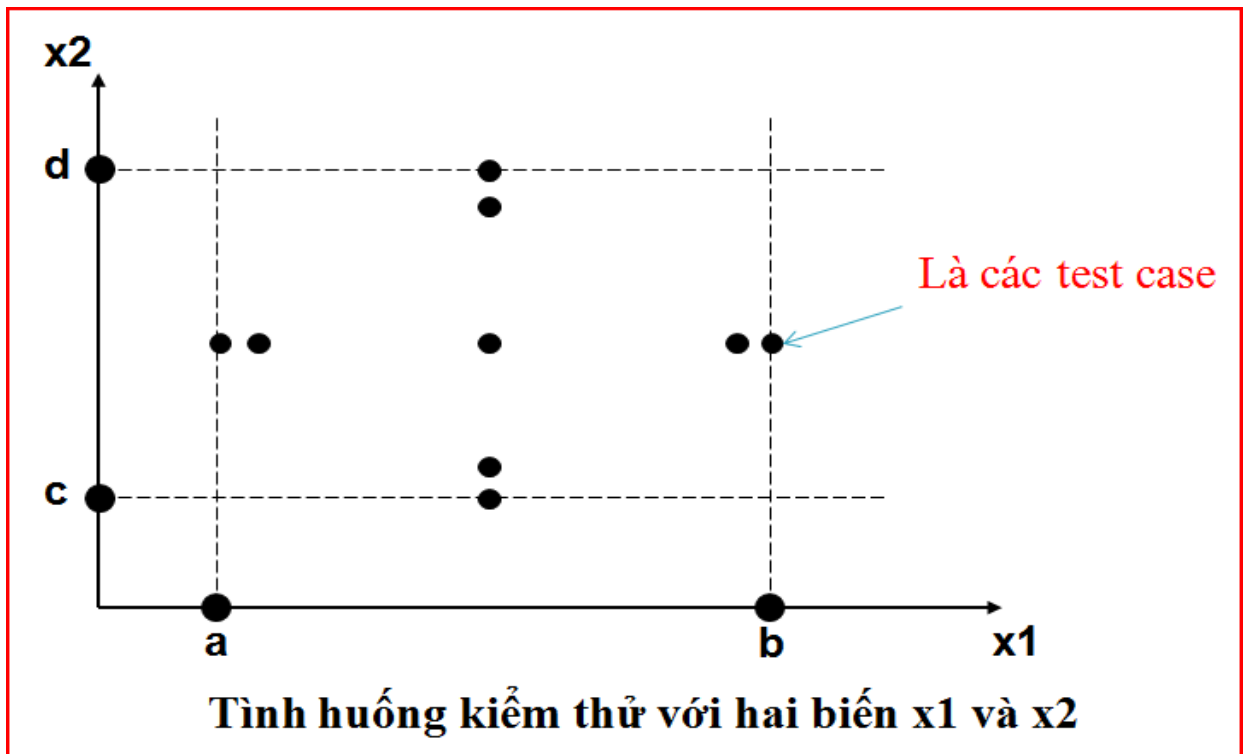


Hình 4: Đồ thị kiểm thử giá trị biên với một biến

Ví dụ: Cho một mảng $[-3, 10]$ ta có thể thiết kế được các test case là:

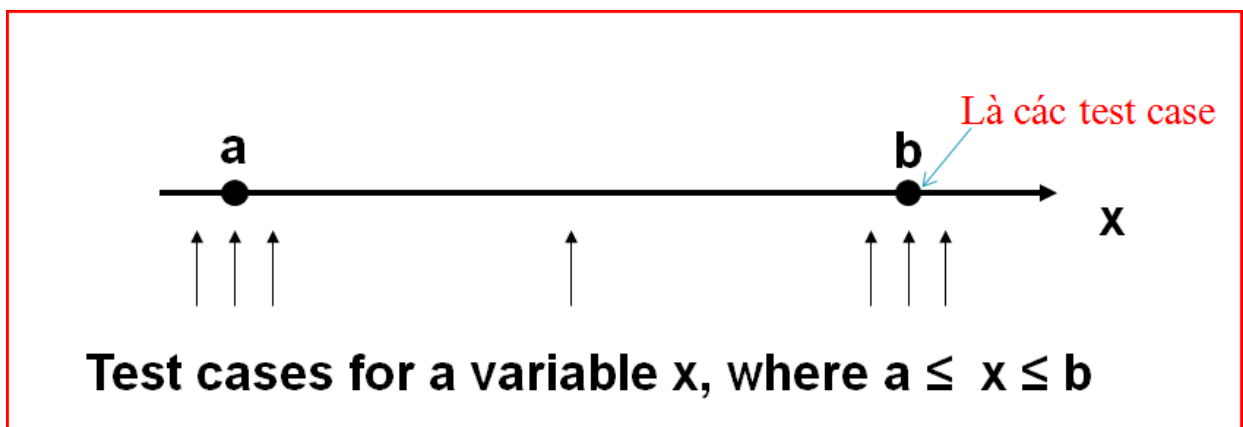
- Giá trị nhỏ nhất: -3
- Giá trị lớn nhất: 10
- Giá trị nhỏ hơn giá trị nhỏ nhất: -4
- Giá trị lớn hơn giá trị lớn nhất: 11
- Giá trị nằm trong -3 và 10: 0

- Kiểm thử theo giá trị biên theo hai biến $x1$ và $x2$.



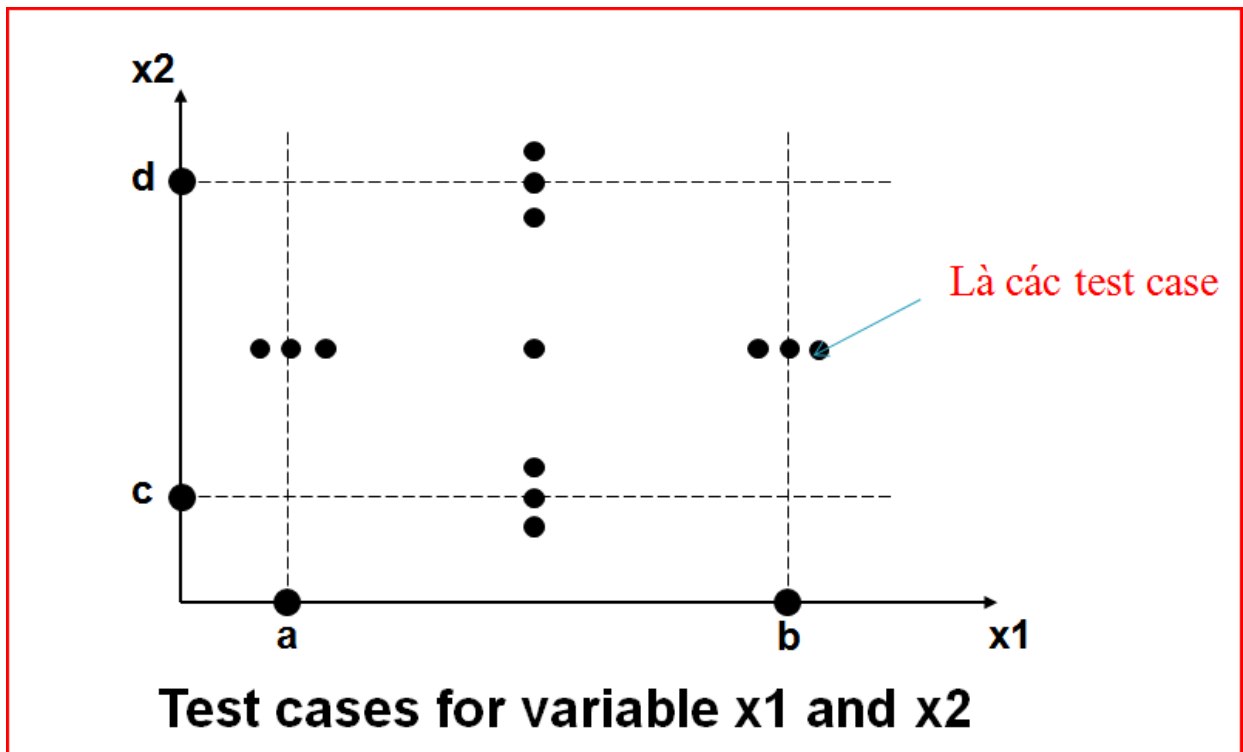
Hình 5: Đồ thị kiểm thử giá trị biên với hai biến

- Kiểm thử theo giá trị biên đầy đủ với một biến $x1$.



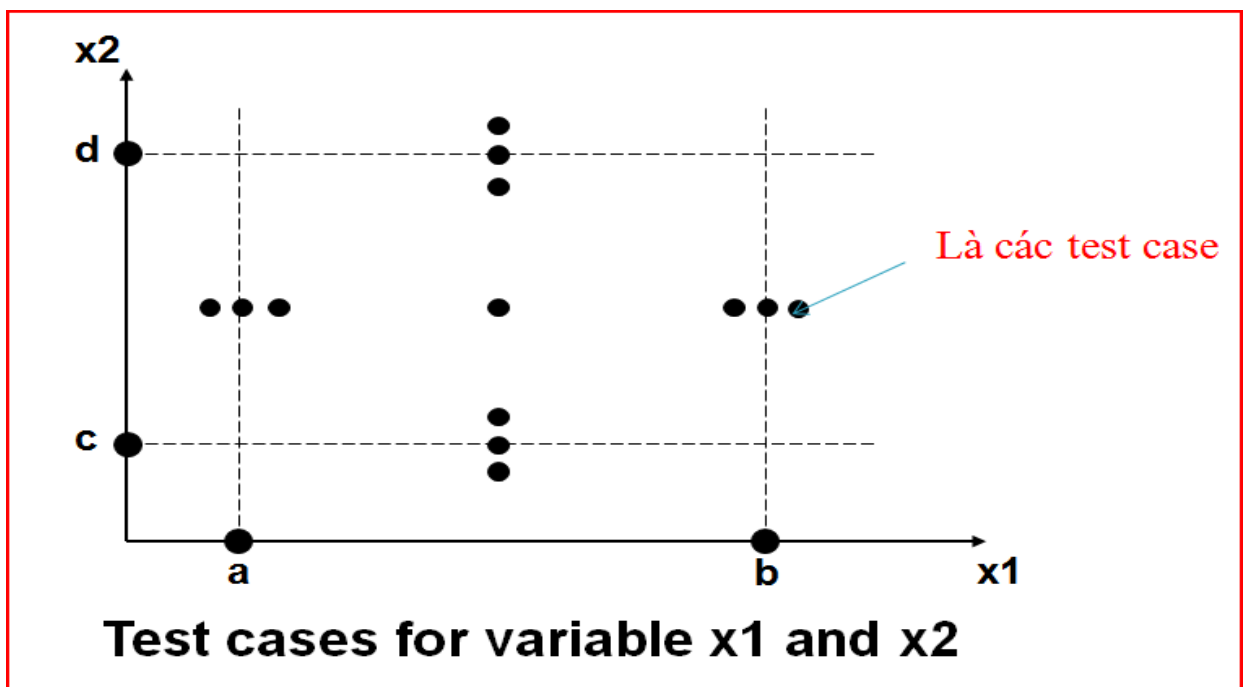
Hình 6: Đồ thị kiểm thử giá trị biên đầy đủ với một biến

- Kiểm thử theo giá trị biên đầy đủ với hai biến $x1$ và $x2$.



Hình 7: Đồ thị kiểm thử giá trị biên đầy đủ với hai biến.

- Kiểm thử theo giá trị biên xấu nhất với hai biến $x1$ và $x2$.



Hình 8: Đồ thị kiểm thử giá trị biên xấu nhất với hai biến.

Kết luận: Số lượng trường hợp kiểm thử phải có: (với n là các biến)

- Kiểm thử theo giá trị biên: $4n + 1$
- Kiểm thử theo giá trị biên đầy đủ: $6n + 1$
- Kiểm thử theo giá trị biên xấu nhất: 5^n
- Nếu miền giá trị đầu vào là một danh sách các giá trị thì có thể thiết kế các test case như sau:

- Giá trị nhỏ nhất.
- Giá trị lớn hơn giá trị nhỏ nhất.
- Giá trị lớn nhất.
- Giá trị nhỏ hơn giá trị lớn nhất.

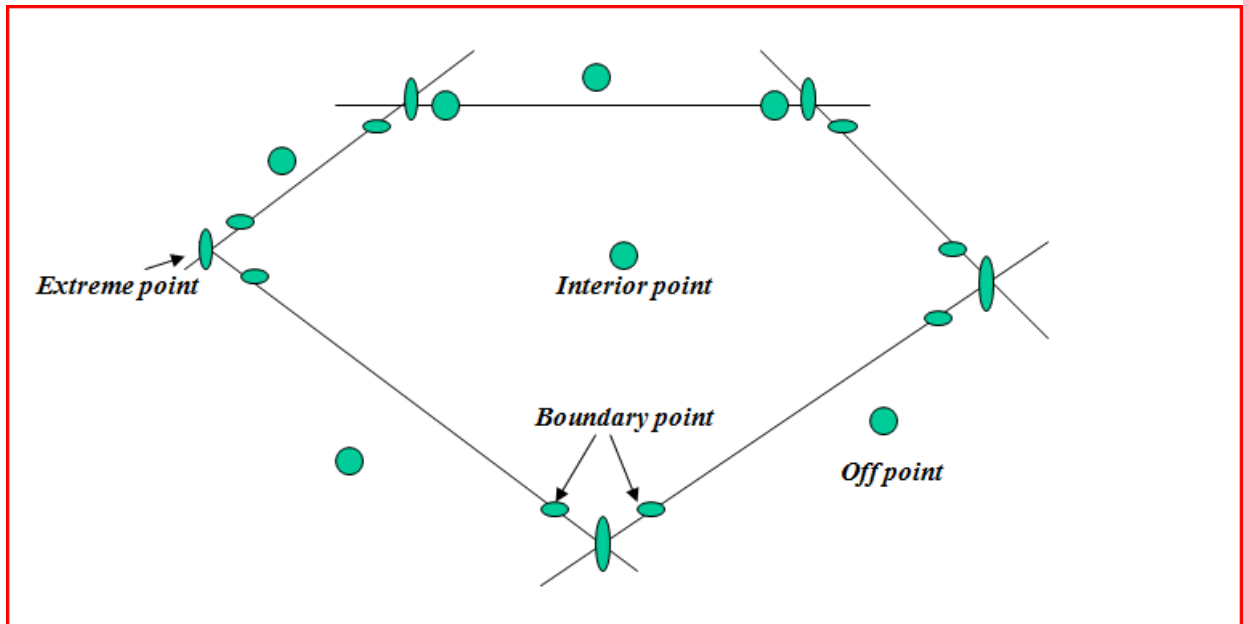
Ví dụ: Cho một danh sách như sau { 3,5,90,100,102 } nên có thể thiết kế như sau:

- Giá trị nhỏ nhất: 3
- Giá trị lớn hơn giá trị nhỏ nhất: 5
- Giá trị lớn nhất: 102
- Giá trị nhỏ hơn giá trị lớn nhất: 100
- Nếu dữ liệu vào là điều kiện ràng buộc số giá trị thì có thể thiết kế được các test case như sau:

- Số giá trị tối thiểu.
- Số giá trị tối đa.
- Và một số giá trị không hợp lệ.

c. Phân loại miền biên.

- Điểm trên biên (*Boundary point*).
- Điểm cực biên (*Extreme point*).
- Điểm ngoài biên (*Off point*).
- Điểm trong biên (*Interior point*).



Hình 9: Phân loại miền biên

d. Ví dụ cho phân tích giá trị biên

Kiểm tra tính hợp lệ của tháng trong năm thì ta có thể thiết kế như sau:

- Giá trị biên: 1 và 12
- Giá trị cận biên ở bên trong: 2 và 11
- Giá trị cận biên ở bên ngoài: 0 và 13

3.3.1.3. Đồ thị nguyên nhân – hệ quả.

a. Định nghĩa.

Một điểm yếu của hai phương pháp trên là chúng không khảo sát sự kết hợp của các trường hợp đầu vào. Việc kiểm tra sự kết hợp đầu vào không phải là một nhiệm vụ đơn giản bởi vì nếu bạn phân lớp tương đương các trạng thái đầu vào, thì số lượng sự kết hợp thường là rất lớn. Nếu bạn không có cách lựa chọn có hệ thống một tập con các trạng thái đầu vào, có lẽ bạn sẽ chọn ra một tập tùy hứng các điều kiện, điều này có thể dẫn tới việc kiểm thử không có hiệu quả.

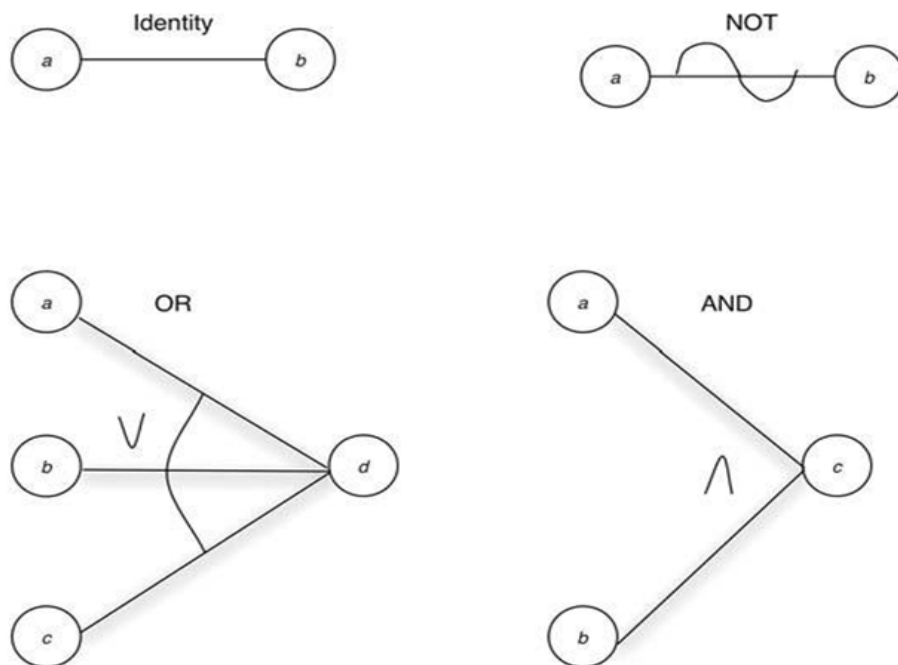
Đồ thị nguyên nhân – kết quả hỗ trợ trong việc lựa chọn một cách có hệ thống tập các ca kiểm thử có hiệu quả cao. Nó có tác động có lợi ảnh hưởng tới việc chỉ ra tình trạng chưa đầy đủ và nhập nhằng trong đặc tả. Nó cung cấp cả cách biểu diễn chính xác cho các điều kiện logic và hành động tương ứng.

Kỹ thuật gồm có 4 bước:

- Xác định điều kiện vào và hành động cho mỗi module cần kiểm định.
- Xác định đồ thị nguyên nhân – kết quả.
- Đồ thị được chuyển thành bảng quyết định.
- Những phần trong bảng quyết định được chuyển thành test case.

b. Các ký hiệu cơ bản.

- Mỗi nút có giá trị là 0 hoặc 1.
- 0 mô tả trạng thái vắng mặt và 1 mô tả trạng thái có mặt.

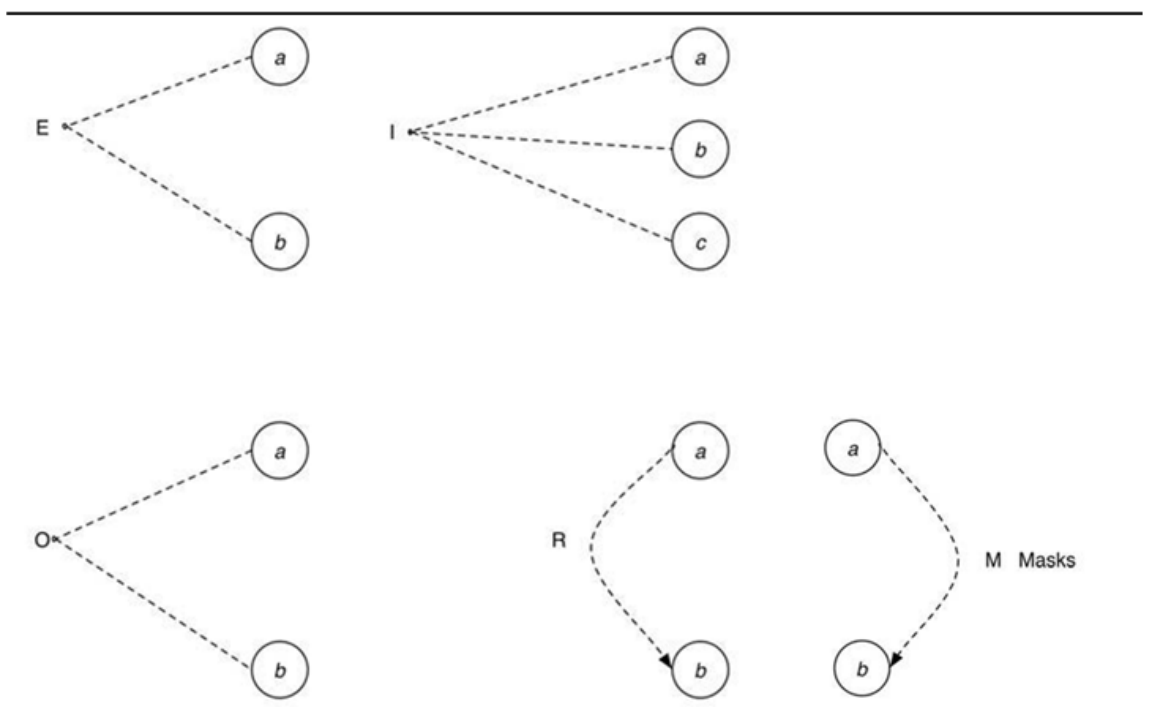


Hình 10: Các ký hiệu của đồ thị nguyên nhân – kết quả.

- Hàm đồng nhất (Identity) nói:
 - Nếu *a* là 1 thì *b* là 1
 - Nếu *a* là 0 thì *b* là 0
- Hàm NOT nói:
 - Nếu *a* là 1 thì *b* là 0

- Nếu a là 0 thì b là 1
- Hàm OR nói: (Cho phép số lượng đầu vào là bất kỳ)
 - Nếu a hoặc b hoặc c là 1 thì d là 1
 - Nếu a hoặc b hoặc c là 0 thì d là 0.
- Hàm AND nói: (Số lượng đầu vào là bất kỳ)
 - Nếu cả a và b là 1 thì c là 1
 - Nếu cả a và b là 0 thì c là 0.

c. Các ký hiệu ràng buộc.



Hình 11: Các ký hiệu ràng buộc trong đồ thị nguyên nhân – kết quả.

- Ràng buộc E (Exclude – loại trừ):
 - Khẳng định tối đa rằng, chỉ có thể a hoặc b là 1 (a,b không đồng thời = 1)
- Ràng buộc I (Include – bao hàm):
 - Khẳng định ít nhất một trong a,b hoặc c phải luôn là 1 (a,b,c không đồng thời bằng 0)
- Ràng buộc O (Only – chỉ một):

- Một và chỉ một a hoặc b là 1
- Ràng buộc R (Request – yêu cầu):
 - Khi a là 1 thì b phải là 1
- Ràng buộc M (Mask – mặt nạ):
 - Nếu kết quả a là 1 thì kết quả b bắt buộc phải là 0.

Bước tiếp theo là tạo bảng quyết định mục vào giới hạn – *limited-entry decision table*. Tương tự với các bảng quyết định, thì nguyên nhân chính là các điều kiện và kết quả chính là các hành động.

Quy trình được sử dụng là như sau:

- Chọn một kết quả để là trạng thái có mặt (1).
- Lăn ngược trở lại đồ thị, tìm tất cả những sự kết hợp của các nguyên nhân (đối tượng cho các ràng buộc) mà sẽ thiết lập kết quả này thành 1.
- Tạo một cột trong bảng quyết định cho mỗi sự kết hợp nguyên nhân.
- Với mỗi sự kết hợp, hãy quy định trạng thái của tất cả các kết quả khác và đặt chúng vào mỗi cột.

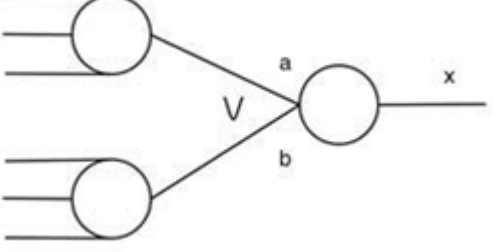
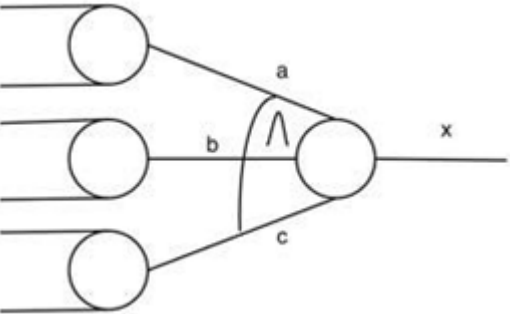
Trong khi biểu diễn bước 2, cần quan tâm các vấn đề sau:

- Khi lăn ngược trở lại qua một nút *or* mà đầu ra của nó là 1, không bao giờ thiết lập nhiều hơn 1 đầu vào cho nút *or* là 1 một cách đồng thời. Điều này được gọi là *path sensitizing – làm nhạy đường đi*. Mục tiêu của nó là để ngăn chặn dò lỗi thất bại vì một nguyên nhân che đi một nguyên nhân khác.
- Khi lăn ngược trở lại qua một nút *and* mà đầu ra của nó là 0, dĩ nhiên, phải liệt kê tất cả các sự kết hợp đầu vào dẫn tới đầu ra 0. Tuy nhiên, nếu bạn đang khảo sát trạng thái mà 1 đầu ra là 0 và một hay nhiều đầu ra khác là 1, thì không nhất thiết phải liệt kê tất cả các điều kiện mà dưới điều kiện đó các đầu vào khác có thể là 1.

Khi lăn ngược trở lại qua một nút *and* mà đầu ra của nó là 0, chỉ cần liệt kê 1 điều kiện trong đó tất cả đầu vào bằng 0. (Nếu nút *and* ở chính giữa của đồ thị như vậy thì tất

cả các đầu vào của nó xuất phát từ các nút trung gian khác, có thể có quá nhiều trạng thái mà trong trạng thái đó tất cả các đầu vào của nó bằng 0.)

Những xem xét được dò theo đồ thị:

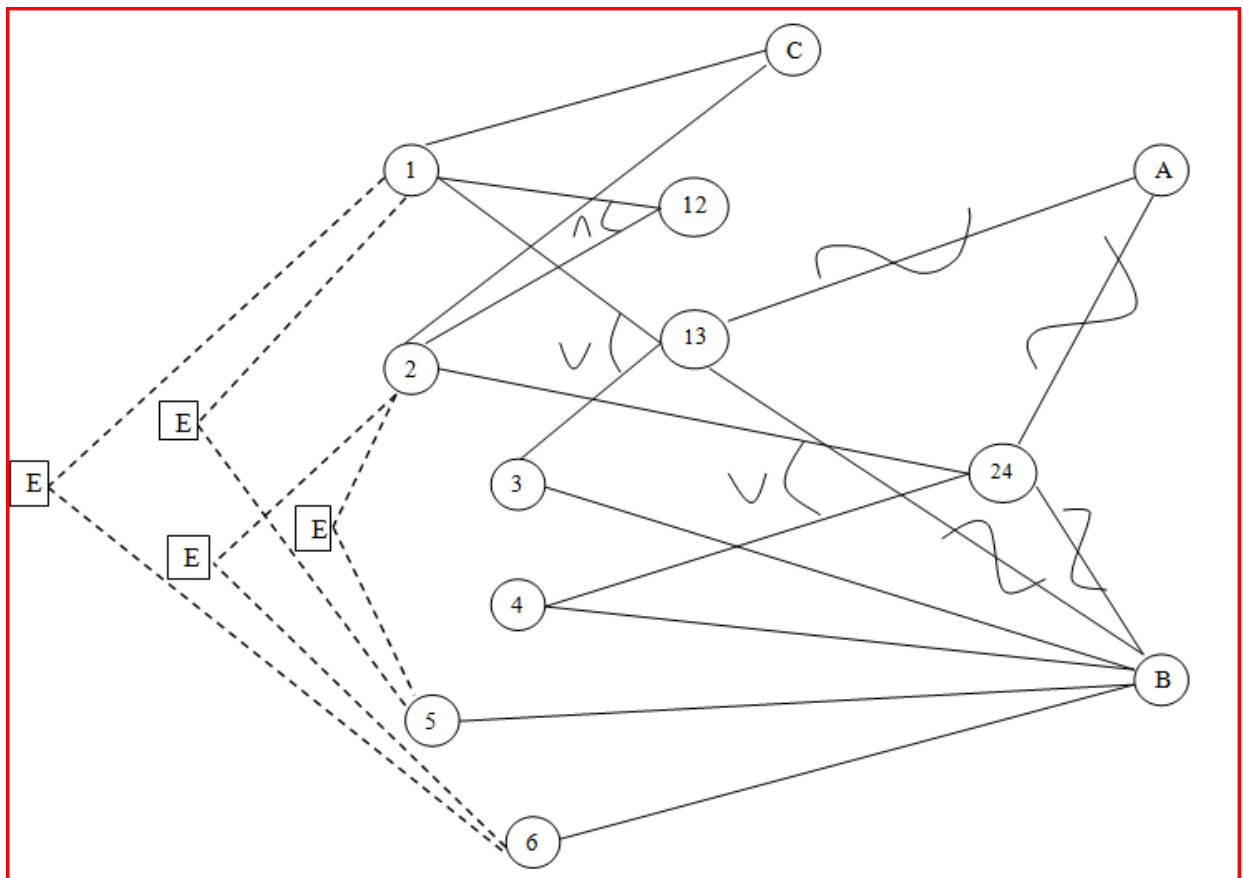
	<ul style="list-style-type: none"> • Nếu $x=1$, không quan tâm về trường hợp $a=b=1$ (sự xem xét thứ 1) • Nếu $x=0$, liệt kê tất cả các trường hợp trong đó $a=b=0$.
	<ul style="list-style-type: none"> • Nếu $x=1$, liệt kê tất cả các trường hợp trong đó $a=b=c=1$. • Nếu $x=0$, bao gồm chỉ 1 trường hợp mà $a=b=c=0$ (sự xem xét 3). Đối với các trạng thái mà abc là 001, 010, 100, 011, 101 và 110, bao gồm chỉ 1 trường hợp mỗi trạng thái (sự xem xét 2).

Hình 12: Các hình biểu diễn dò theo đồ thị

- Ví dụ cho bài toán nhập tháng trong một chương trình. Sử dụng phương pháp đồ thị nguyên nhân – kết quả để thiết kế:
 - Bước 1: Xác định điều kiện nhập vào.

Cause (Điều kiện vào)	Effect (Hành động)
1. Số nguyên ≥ 1 2. Số nguyên ≤ 12 3. Số < 1 4. Số > 12 5. Chuỗi 6. Không phải số nguyên	A: Đúng B: Sai C: Nghi ngờ

- Xây dựng đồ thị. (có sử dụng các ký hiệu cơ bản và cả ký hiệu ràng buộc)



Hình 13: Đồ thị của ví dụ nhập tháng.

Nhận xét:

Vẽ đồ thị nguyên nhân – kết quả là phương pháp tạo các ca kiểm thử có hệ thống mô tả sự kết hợp của các điều kiện. Sự thay đổi sẽ là 1 sự lựa chọn kết hợp không thể dự tính trước, nhưng khi thực hiện như vậy, có vẻ như bạn sẽ bỏ sót nhiều ca kiểm thử “thứ vị” được xác định bằng đồ thị nguyên nhân – kết quả .

Vì vẽ đồ thị nguyên nhân – kết quả yêu cầu chuyển một đặc tả thành một mạng logic Boolean, nó cung cấp một triển vọng khác và sự hiểu biết sâu sắc hơn nữa về đặc tả. Trên thực tế, sự phát triển của 1 đồ thị nguyên nhân – kết quả là cách hay để khám phá sự mơ hồ và chưa đầy đủ trong các đặc tả.

Mặc dù việc vẽ đồ thị nguyên nhân – kết quả tạo ra tập các ca kiểm thử hữu dụng, nhưng thông thường nó không tạo ra tất cả các ca kiểm thử hữu dụng mà có thể được nhận

biết. Ngoài ra, đồ thị nguyên nhân – kết quả không khảo sát thỏa đáng các điều kiện giới hạn. Dĩ nhiên, bạn có thể cố gắng bao phủ các điều kiện giới hạn trong suốt quá trình.

Tuy nhiên, vấn đề trong việc thực hiện điều này là nó làm cho đồ thị rất phức tạp và dẫn tới số lượng rất lớn các ca kiểm thử. Vì thế, tốt nhất là xét 1 sự phân tích giá trị giới hạn tách rời nhau.

Vì đồ thị nguyên nhân – kết quả làm chúng ta mất thời gian trong việc chọn các giá trị cụ thể cho các toán hạng, nên các điều kiện giới hạn có thể bị pha trộn thành các ca kiểm thử xuất phát từ đồ thị nguyên nhân – kết quả. Vì vậy, chúng ta đạt được một tập các ca kiểm thử nhỏ nhưng hiệu quả mà thỏa mãn cả 2 mục tiêu.

3.3.1.4. *Đoán lỗi – Error Guessing*

Một kỹ thuật thiết kế test-case khác là *error guessing* – *đoán lỗi*. Tester được đưa cho 1 chương trình đặc biệt, họ phỏng đoán, cả bằng trực giác và kinh nghiệm, các loại lỗi có thể và sau đó viết các ca kiểm thử để đưa ra các lỗi đó.

Thật khó để đưa ra một quy trình cho kỹ thuật đoán lỗi vì nó là một quy trình có tính trực giác cao và không thể dự đoán trước. Ý tưởng cơ bản là liệt kê một danh sách các lỗi có thể hay các trường hợp dễ xảy ra lỗi và sau đó viết các ca kiểm thử dựa trên danh sách đó. Một ý tưởng khác để xác định các ca kiểm thử có liên đới với các giả định mà lập trình viên có thể đã thực hiện khi đọc đặc tả (tức là, những thứ bị bỏ sót khỏi đặc tả, hoặc là do tình cờ, hoặc là vì người viết có cảm giác những đặc tả đó là rõ ràng). Nói cách khác, bạn liệt kê những trường hợp đặc biệt đó mà có thể đã bị bỏ sót khi chương trình được thiết kế.

3.3.2. *Phương pháp kiểm thử hộp trắng – White box testing.*

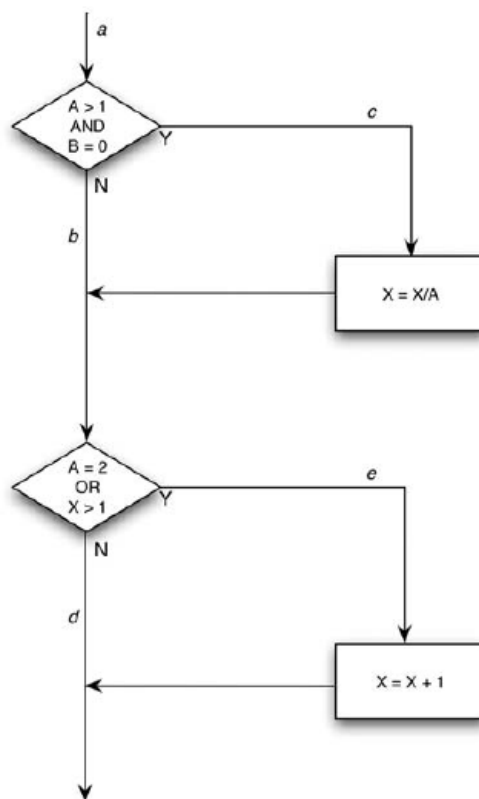
Kiểm thử hộp trắng có liên quan tới mức độ mà các ca kiểm thử thực hiện hay bao phủ tính logic (mã nguồn) của chương trình. Kiểm thử hộp trắng cơ bản là việc thực hiện mọi đường đi trong chương trình, nhưng việc kiểm thử đầy đủ đường đi là một mục đích không thực tế cho một chương trình với các vòng lặp. Các tiêu chuẩn trong kiểm thử bao phủ logic gồm có:

3.3.2.1. *Bao phủ câu lệnh.*

Tư tưởng: Thực hiện mọi câu lệnh trong chương trình ít nhất 1 lần.

Xét ví dụ với đoạn mã lệnh JAVA sau:

```
public void A (int a, int b, int x){
    if (a>1 && b==0) {
        x=x/a;}
    if (a==2||x>1){
        x=x+1;
    }
}
```



Hình 14: Một chương trình nhỏ để kiểm thử

Bạn có thể thực hiện mọi câu lệnh bằng việc viết 1 ca kiểm thử đơn đi qua đường *ace*. Tức là, bằng việc đặt $A=2$, $B=0$ và $X=3$ tại điểm *a*, mỗi câu lệnh sẽ được thực hiện 1 lần (thực tế, X có thể được gán bất kỳ giá trị nào).

Thường tiêu chuẩn này khá kém. Ví dụ, có lẽ nếu quyết định đầu tiên là phép *or* chứ không phải phép *and* thì lỗi này sẽ không được phát hiện. Hay nếu quyết định thứ hai mà bắt đầu với $x>0$, lỗi này cũng sẽ không được tìm ra. Cũng vậy, có 1 đường đi qua chương

trình mà ở đó x không thay đổi (đường đi abd). Nếu đây là 1 lỗi, thì lỗi này có thể không tìm ra. Hay nói cách khác, tiêu chuẩn bao phủ câu lệnh quá yếu đến nỗi mà nó thường là vô ích.

3.3.2.2. Bao phủ quyết định.

Tư tưởng: Viết đủ các ca kiểm thử mà mỗi quyết định có kết luận đúng hay sai ít nhất 1 lần. Nói cách khác, mỗi hướng phân nhánh phải được xem xét kỹ lưỡng ít nhất 1 lần.

Các ví dụ về câu lệnh rẽ nhánh hay quyết định là các câu lệnh switch, do-while, và if-else. Các câu lệnh đa đường GOTO thường sử dụng trong một số ngôn ngữ lập trình như FORTRAN.

Bao phủ quyết định thường thỏa mãn bao phủ câu lệnh. Vì mỗi câu lệnh là trên sự bắt nguồn một đường đi phụ nào đó hoặc là từ 1 câu lệnh rẽ nhánh hoặc là từ điểm vào của chương trình, mỗi câu lệnh phải được thực hiện nếu mỗi quyết định rẽ nhánh được thực hiện. Tuy nhiên, có ít nhất 3 ngoại lệ:

- Những chương trình không có quyết định.
- Những chương trình hay thường trình con/phương thức với nhiều điểm vào. Một câu lệnh đã cho có thể được thực hiện nếu và chỉ nếu chương trình được nhập vào tại 1 điểm đầu vào riêng.
- Các câu lệnh bên trong các ON-unit. Việc đi qua mỗi hướng rẽ nhánh sẽ là không nhất thiết làm cho tất cả các ON-unit được thực thi.

Vì chúng ta đã thấy rằng bao phủ câu lệnh là điều kiện cần thiết, nên một chiến lược tốt hơn là bao phủ quyết định nên được định nghĩa bao hàm cả bao phủ câu lệnh. Do đó, bao phủ quyết định yêu cầu mỗi quyết định phải có kết luận đúng hoặc sai, và mỗi câu lệnh đó phải được thực hiện ít nhất 1 lần.

Phương pháp này chỉ xem xét những quyết định hay những sự phân nhánh 2 đường và phải được sửa đổi cho những chương trình có chứa những quyết định đa đường. Ví dụ, các chương trình JAVA có chứa các lệnh *select (case)*, các chương trình FORTRAN chứa các lệnh số học (ba đường) *if* hoặc các lệnh tính toán hay số học *GOTO*, và các chương

trình COBOL chứa các lệnh *GOTO* biến đổi hay các lệnh *GO-TO-DEPENDING-ON* (các lệnh goto phụ thuộc). Với những chương trình như vậy, tiêu chuẩn này đang sử dụng mỗi kết luận có thể của tất cả các quyết định ít nhất 1 lần và gọi mỗi điểm vào tới chương trình hay thường trình con ít nhất 1 lần.

Trong hình 14, bao phủ quyết định có thể đạt được bởi ít nhất 2 ca kiểm thử bao phủ các đường *ace* và *abd* hoặc *acd* và *abe*. Nếu chúng ta chọn khả năng thứ hai, thì 2 đầu vào test-case là $A=3, B=0, X=3$ và $A=2, B=1, X=1$.

Bao phủ quyết định là 1 tiêu chuẩn mạnh hơn bao phủ câu lệnh, nhưng vẫn khá yếu. Ví dụ, chỉ có 50% cơ hội là chúng ta sẽ tìm ra con đường trong đó x không bị thay đổi (ví dụ, chỉ khi bạn chọn khả năng thứ nhất). Nếu quyết định thứ hai bị lỗi (nếu như đáng lẽ phải nói là $x < 1$ thay vì $x > 1$), lỗi này sẽ không được phát hiện bằng 2 ca kiểm thử trong ví dụ trước.

3.3.2.3. Bao phủ điều kiện

Tư tưởng: Viết đủ các ca kiểm thử để đảm bảo rằng mỗi điều kiện trong một quyết định đảm nhận tất cả các kết quả có thể ít nhất một lần.

Vì vậy, như với bao phủ quyết định, thì bao phủ điều kiện không phải luôn luôn dẫn tới việc thực thi mỗi câu lệnh. Thêm vào đó, trong tiêu chuẩn bao phủ điều kiện, mỗi điểm vào chương trình hay thường trình con, cũng như các ON-unit, được gọi ít nhất 1 lần. Ví dụ, câu lệnh rẽ nhánh do $k=0$ to 50 while ($j+k < \text{quest}$) có chứa 2 điều kiện là $k \leq 50$, và $j+k < \text{quest}$. Do đó, các ca kiểm thử sẽ được yêu cầu cho những tình huống $k \leq 50$, $k > 50$ (để đến lần lặp cuối cùng của vòng lặp), $j+k < \text{quest}$, và $j+k \geq \text{quest}$.

Hình 2.1 có 4 điều kiện: $A > 1$, $B=0$, $A=2$, $X > 1$. Do đó các ca kiểm thử đầy đủ là cần thiết để thúc đẩy những trạng thái mà $A > 1$, $A \leq 1$, $B=0$ và $B \neq 0$ có mặt tại điểm *a* và $A=2$, $A \neq 2$, $X > 1$, $X \leq 1$ có mặt tại điểm *b*. Số lượng đầy đủ các ca kiểm thử thỏa mãn tiêu chuẩn và những đường đi mà được đi qua bởi mỗi ca kiểm thử là:

1. $A=2, B=0, X=4$ *ace*
2. $A=1, B=1, X=1$ *abd*

Chú ý là, mặc dù cùng số lượng các ca kiểm thử được tạo ra cho ví dụ này, nhưng bao phủ điều kiện thường tốt hơn bao phủ quyết định là vì nó có thể (nhưng không luôn luôn) gây ra mọi điều kiện riêng trong 1 quyết định để thực hiện với cả hai kết quả, trong khi bao phủ quyết định lại không. Ví dụ trong cùng câu lệnh rẽ nhánh: *DO K=0 TO 50 WHILE (J+K<QUEST)* là 1 nhánh 2 đường (thực hiện thân vòng lặp hay bỏ qua nó). Nếu bạn đang sử dụng kiểm thử quyết định, thì tiêu chuẩn này có thể được thỏa mãn bằng cách cho vòng lặp chạy từ *K=0* tới *51*, mà chưa từng kiểm tra trường hợp trong đó mệnh đề *WHILE* bị sai. Tuy nhiên, với tiêu chuẩn bao phủ điều kiện, 1 ca kiểm thử sẽ cần phải cho ra 1 kết quả sai cho những điều kiện *J+K<QUEST*.

Mặc dù nếu mới nhìn thoáng qua, tiêu chuẩn bao phủ điều kiện xem ra thỏa mãn tiêu chuẩn bao phủ quyết định, nhưng không phải lúc nào cũng vậy. Nếu quyết định *IF (A&B)* được kiểm tra, thì tiêu chuẩn bao phủ điều kiện sẽ cho phép bạn viết 2 ca kiểm thử - A đúng, B sai, và A sai, B đúng – nhưng điều này sẽ không làm cho mệnh đề *THEN* của câu lệnh *IF* được thực hiện.

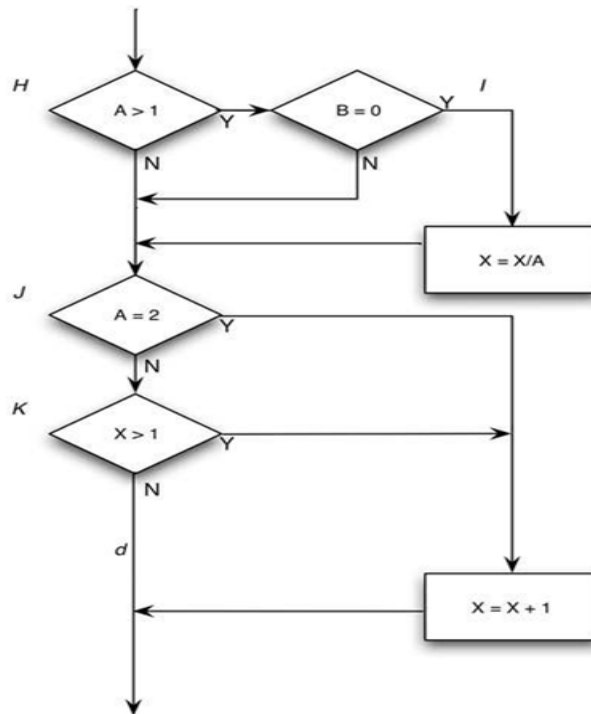
Ví dụ, 2 ca kiểm thử khác:

1. *A=1, B=0, X=3*
2. *A=2, B=1, X=1*

3.3.2.4. Bao phủ quyết định – điều kiện.

Tư tưởng: Thực hiện đủ các ca kiểm thử mà mỗi điều kiện trong 1 quyết định thực hiện trên tất cả các kết quả có thể ít nhất 1 lần, và mỗi điểm vào được gọi ít nhất 1 lần.

Điểm yếu của bao phủ quyết định/điều kiện là mặc dù xem ra nó có thể sử dụng tất cả các kết quả của tất cả các điều kiện, nhưng thường không phải vậy vì những điều kiện chắc chắn đã cản các điều kiện khác.



Hình 15: Mã máy cho chương trình

Biểu đồ tiến trình trong hình 15 là cách 1 trình biên dịch tạo ra mã máy cho chương trình trong Hình 14. Các quyết định đa điều kiện trong chương trình nguồn đã bị chia thành các quyết định và các nhánh riêng vì hầu hết các máy không được chế tạo để có thể thực hiện các quyết định đa điều kiện. Khi đó 1 bao phủ kiểm thử tỉ mỉ hơn xuất hiện là việc sử dụng tất cả các kết quả có thể của mỗi quyết định gốc. Hai ca kiểm thử bao phủ quyết định trước không làm được điều này; chúng không thể sử dụng kết quả *false* của quyết định H và kết quả *true* của quyết định K.

Lí do, như đã được chỉ ra trong hình 15, là những kết quả của các điều kiện trong các biểu thức *and* và *or* có thể cản trở hay ngăn chặn việc ước lượng các quyết định khác. Ví dụ, nếu 1 điều kiện *and* là sai, không cần kiểm tra các điều kiện tiếp theo trong biểu thức. Tương tự như vậy, nếu 1 điều kiện *or* là đúng thì cũng không cần kiểm tra các điều kiện còn lại. Do đó, các lỗi trong biểu thức logic không phải lúc nào cũng được phát hiện bằng các tiêu chuẩn bao phủ điều kiện và bao phủ quyết định/điều kiện.

3.3.2.5. Bao phủ đa điều kiện

Tư tưởng: Viết đủ các ca kiểm thử mà tất cả những sự kết hợp của các kết quả điều kiện có thể trong mỗi quyết định, và tất cả các điểm vào phải được gọi ít nhất 1 lần.

Ví dụ, xét chuỗi mã lệnh sau:

```
NOTFOUND = TRUE;  
DO I=1 TO TABSIZE WHILE (NOTFOUND); /*SEARCH TABLE*/  
    ...searching logic...;  
END;
```

Bốn tình huống để kiểm thử là:

1. $I \leq \text{TABSIZE}$ và NOTFOUND có giá trị đúng (đang duyệt)
2. $I \leq \text{TABSIZE}$ và NOTFOUND có giá trị sai (tìm thấy mục vào trước khi gặp cuối bảng).
3. $I > \text{TABSIZE}$ và NOTFOUND có giá trị đúng (gặp cuối bảng mà không tìm thấy mục vào).
4. $I > \text{TABSIZE}$ và NOTFOUND có giá trị sai (mục vào là cái cuối cùng trong bảng).

Dễ nhận thấy là tập hợp các ca kiểm thử thỏa mãn tiêu chuẩn đa điều kiện cũng thỏa mãn các tiêu chuẩn bao phủ quyết định, bao phủ điều kiện và bao phủ quyết định/điều kiện.

Quay lại hình 14, các ca kiểm thử phải bao phủ 8 sự kết hợp:

1. $A > 1, B = 0$
2. $A > 1, B < > 0$
3. $A \leq 1, B = 0$
4. $A \leq 1, B < > 0$
5. $A = 2, X > 1$
6. $A = 2, X \leq 1$
7. $A < > 2, X > 1$
8. $A < > 2, X \leq 1$

Vì là ca kiểm thử sớm hơn, nên cần chú ý là các trường hợp từ 5 đến 8 biểu diễn các giá trị tại vị trí câu lệnh *IF* thứ hai. Vì *X* có thể thay đổi ở trên câu lệnh *IF* này, nên giá trị cần tại câu lệnh *IF* này phải được sao dự phòng thông qua tính logic để tìm ra các giá trị đầu vào tương ứng.

Những sự kết hợp để được kiểm tra này không nhất thiết ngụ ý rằng cần thực hiện cả 8 ca kiểm thử. Trên thực tế, chúng có thể được bao phủ bởi 4 ca kiểm thử. Các giá trị đầu vào kiểm thử, và sự kết hợp mà chúng bao phủ, là như sau:

$A=2, B=0, X=4$	Bao phủ trường hợp 1, 5
$A=2, B=1, X=1$	Bao phủ trường hợp 2, 6
$A=1, B=0, X=2$	Bao phủ trường hợp 3, 7
$A=1, B=1, X=1$	Bao phủ trường hợp 4, 8

Thực tế là việc có 4 ca kiểm thử và 4 đường đi riêng biệt trong hình 14 chỉ là sự trùng hợp ngẫu nhiên. Trên thực tế, 4 ca kiểm thử này không bao phủ mọi đường đi, chúng bỏ qua đường đi *acd*. Ví dụ, bạn sẽ cần 8 ca kiểm thử cho quyết định sau mặc dù nó chỉ chứa 2 đường đi:

```
If (x==y&&length(z)==0&&FLAG) {  
    J=1;}  
Else {  
    I=1;}  
}
```

Trong trường hợp các vòng lặp, số lượng các ca kiểm thử được yêu cầu bởi tiêu chuẩn đa điều kiện thường ít hơn nhiều số lượng đường đi.

Tóm lại, đối với những chương trình chỉ chứa 1 điều kiện trên 1 quyết định, thì 1 tiêu chuẩn kiểm thử nhỏ nhất là một số lượng đủ các ca kiểm thử để (1) gọi tất cả các kết quả của mỗi quyết định ít nhất 1 lần và (2) gọi mỗi điểm của mục vào (như là điểm vào hay ON-unit) ít nhất 1 lần, để đảm bảo là tất cả các câu lệnh được thực hiện ít nhất 1 lần. Đối với những chương trình chứa các quyết định có đa điều kiện thì tiêu chuẩn tối thiểu là số lượng đủ các ca kiểm thử để gọi tất cả những sự kết hợp có thể của các kết quả điều kiện trong mỗi quyết định, và tất cả các điểm vào của chương trình ít nhất 1 lần.

CHƯƠNG IV: TÌM HIỂU VỀ NUNIT

4.1. Các công cụ kiểm thử của từng ngôn ngữ kiểm thử.

4.1.1. Junit và J2ME Unit trong Java.

- *Định nghĩa Junit:* Là một framework được dùng cho Unit Test trong Java. JUnit được xây dựng bởi Erich Gamma và Kent Beck, hai người nổi tiếng nhất về lập trình XP. Trong Junit có các Testcase là các lớp của Java, các lớp này bao gồm một hay nhiều phương thức cần kiểm tra và testcase này lại được nhóm với nhau để tạo thành Test Suite.

Mỗi phép thử trong Junit là mootk phương thức public, không có đối số và được bắt đầu bằng chữ Test(testXXX()). Nếu chúng ta không tuân thủ theo quy tắc này thì Junit sẽ không xác định được phương thức test một cách tự động.

- *Lợi ích của Junit:*

- Giúp người lập trình không phải làm đi làm lại những việc kiểm thử nhằm chán bằng cách tạo ra tách biệt mã kiểm thử ra khỏi chương trình,
- Tự động hóa việc tổ chức và thi hành các bộ số liệu kiểm thử.

- *Các phương thức trong JUnit.*

- AssertXXX(): được dùng để kiểm tra các điều kiện khác nhau.
 - Boolean assertEquals(): So sánh hai giá trị để kiểm tra bằng nhau. Phép thử thất bại nếu hai giá trị không bằng nhau.
 - Boolean assertFalse(): Đánh giá biểu thức logic. Phép thử thất bại nếu biểu thức đúng.
 - Boolean assertNotNull(): So sánh tham chiếu của một đối tượng với Null. Phép thử thất bại nếu tham chiếu đối tượng Null.
 - Boolean assertNotSame(): So sánh địa chỉ vùng nhớ của hai tham chiếu hai đối tượng bằng cách sử dụng toán tử ==. Phép thử thất bại trả về nếu cả hai đều tham chiếu đến cùng một đối tượng.
 - Boolean assertNull(): So sánh tham chiếu của một đối tượng với giá trị Null. Phép thử thất bại nếu đối tượng không là Null.

- Boolean `assertTrue()`: Đánh giá một biểu thức logic. Phép thử thất bại nếu biểu thức này sai.
- Void `fail()`: Phương thức này làm cho test hiện tại thất bại, phương thức này thường được sử dụng khi xử lý các ngoại lệ.
- `Setup()` và `Teardown()`: Hai phương thức này là một phần của lớp `junit.framework.TestCase`. Khi sử dụng hai phương thức này sẽ giúp chúng ta tránh được việc trùng mã khi nhiều test cùng chia sẻ nhau ở phần khởi tạo và dọn dẹp các biến. Junit tuân theo các bước sau cho mỗi phương thức test:
 - Gọi phương thức `setUp()` của test case.
 - Gọi phương thức thử.
 - Gọi phương thức `tearDown()` của testcase.

4.1.2. Cpp Unit trong C/C++.

- *Định nghĩa Cpp Unit*: Là một framework dùng cho Unit Test trong ngôn ngữ C++.
- *Lợi ích của Cpp Unit*:
 - Giúp cho việc kiểm thử tự động các module (class).
 - Được chuyển từ Junit trong ngôn ngữ Java.
 - Chạy trên Window/ Unix/ Solaris
- *Một số lệnh được sử dụng*:
 - `CppUnit_Assert(condition)`: Khẳng định rằng điều kiện là đúng.
 - `CppUnit_Assert_Message(message, condition)`: Khẳng định với người sử dụng thông điệp quy định.
 - `CppUnit_Fail(message)`: Sai với thông điệp đã quy định.
 - `CppUnit_Assert_Equal(expected, actual)`: Khẳng định có hai giá trị là bằng nhau.

- `CppUnit_Assert_Equal_Message(message, expected, actual)`: Khẳng định hai giá trị đó là bằng nhau, cung cấp thông tin bổ sung về sự thất bại.
- `CppUnit_Assert_Doubles_Equal(expected, actual, delta)`: Lệnh để so sánh giá trị nguyên thủy.

4.1.3. Vb Unit trong Visual Basic.

- *Định nghĩa VbUnit*: Là một framework giúp cho việc viết và chạy tự động cho Unit Test trong ngôn ngữ Visual Basic và đối tượng COM.
- *Lợi ích của VbUnit*:
 - Mục tiêu của chương trình là tạo ra các thông số chính xác.
 - Ngay lập tức phản hồi nếu mã của bạn đang làm việc
 - Việc thay đổi lớn về mã có thể tự tin hơn.
 - Kiểm soát lỗi.
 - Chia sẻ quyền sở hữu về mã.

4.1.4. PyUnit trong Python.

- *Định nghĩa PyUnit*: Là được thiết kế để làm việc với bất cứ chuẩn nào Python và được thử nghiệm của tác giả trên Unix.
- *Đặc điểm của pyUnit*:
 - Có thể viết bài test dễ dàng.
 - PyUnit dựa trên Junit của Java, và do đó sử dụng một kiến trúc thử nghiệm được chứng minh.
 - PyUnit là một phần của thư viện chuẩn Python 2.1
 - Giới thiệu cách sử dụng được thể hiện nhiều trên các website từ phiên bản đầu tiên vào cuối năm 1999.
 - Đã có hơn 500 lượt tải xuống dùng.
 - PyUnit được sử dụng để thử nghiệm Zope, có lẽ là phần lớn nhất và nổi tiếng nhất của phần mềm Python.

4.1.5. Perl Unit trong Perl.

- *Định nghĩa PerlUnit:* Là một framework để thành lập Unit Test trong ngôn ngữ Perl.

- *Đặc điểm của PerlUnit:*

- Giống giao diện của Junit.
- Cung cấp một giao diện thuận tiện.
- Hỗ trợ tích hợp các thử nghiệm: khai thác công cụ.
- Hỗ trợ cách sử dụng theo kiểu lớp nhỏ và sử dụng không theo kiểu lớp nhỏ.
- Có nền tảng độc lập.
- Thực hiện theo tiêu chuẩn các công ước CPAN module để cài đặt etc.
- Có thể được “phân phối theo các điều khoản tương tự như perl”, tức là phân phối lại theo một trong hai nghệ thuật giấy phép hoặc giấy phép công cộng GNU được phép.

4.2. Nunit trong C#.

4.2.1. Định nghĩa.

Nunit là một framework miễn phí được sử dụng khá rộng rãi trong Unit Testing đối với ngôn ngữ dotnet. (chỉ là Unit testing chứ không phải là các loại Unit khác), đơn vị kiểm nghiệm cho tất cả các ngôn ngữ Net.

Ban đầu được chuyển từ JUnit, phiên bản sản xuất hiện nay là phiên bản 2.5.

NUnit được viết hoàn toàn bằng C# và đã được hoàn toàn thiết kế lại để tận dụng lợi thế của nhiều người.

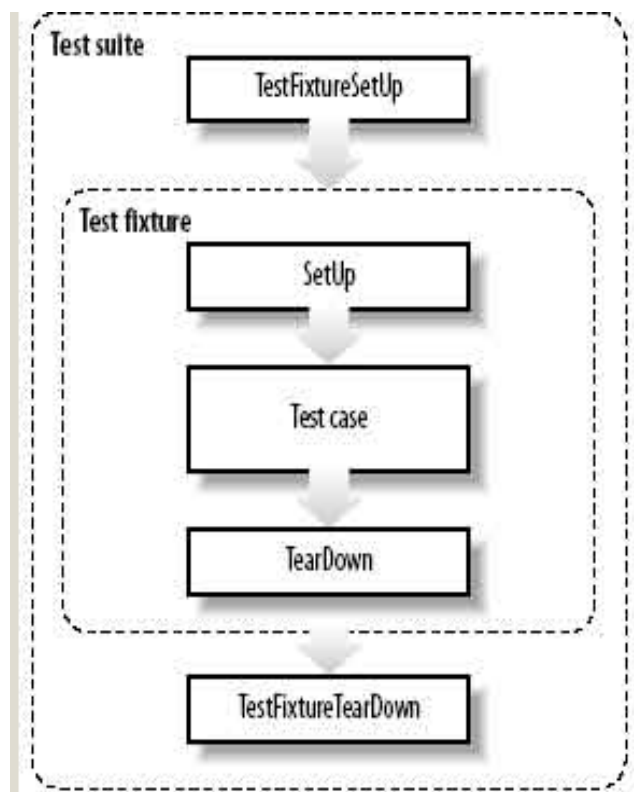
Ngôn ngữ Net cho các thuộc tính tùy chỉnh các tính năng ví dụ và liên quan phản ánh khả năng khác.

4.2.2. Đặc điểm của NUnit.

- Nunit không phải là giao diện tự động kiểm tra.

- Không phải là một ngôn ngữ kịch bản, kiểm tra tất cả các Unit testing được viết bằng .Net và hỗ trợ ngôn ngữ như C#, VC, VB.Net, J#...
- Không phải là công cụ chuẩn.
- Đi qua các bộ phận kiểm tra toàn bộ không có nghĩa là phần mềm được sản xuất sẵn sàng.

4.2.3. Thuộc tính hay dùng trong thư viện *Nunit.Framework*.



- **[TestFixture]**: Dùng để đánh dấu 1 class là test class, những lớp khác không có thuộc tính này sẽ mặc định bị “ignore” khi NUnit test assembly của bạn. ví dụ:

```
using NUnit.Framework;

[TestFixture]
public class MyObjectTestFixture
{
    [Test]
    public void SomeTest()
    { // do some testing :-) }

    public void NotATest()
    { }
}
```

- **[Test]:** Dùng để đánh dấu một phương thức (method) được gọi là test method. Ý nghĩa của nó tương tự như TestFixture nhưng phạm vi ở cấp method. Ví dụ:

```
[Test]
public void SomeUsefulTest()
{
    //do some testing :-)
}
```

- **[SetUp]:** Dùng để đánh dấu một phương thức (method) sẽ gọi trước khi một test case được gọi. Nếu trong một test class có 10 method test, thì mỗi lần một method test chạy thì NUnit sẽ chạy method được đánh dấu với Setup đầu tiên.

- **[TearDown]:** Thuộc tính này ngược với Setup, chạy sau mỗi method test.

- **[TestFixtureSetup]:** Dùng để đánh dấu một class sẽ được gọi trước khi một test case được gọi. Khi một test class được chạy thì method nào được đánh dấu với thuộc tính này sẽ được chạy trước tiên. Ví dụ:

```
[TestFixtureSetUp]
public void MyTFSetUp()
{
    //delete all test files from old tests
    //clear all database tables
}
```

- **[TestFixtureTearDown]:** Ngược với TestFixtureSetup.
- **[ExpectedException]:** Chỉ ra rằng kết quả bình thường của bài test sẽ là một ngoại lệ được đưa vào. Nếu không phải là một ngoại lệ được đưa vào hoặc một ngoại lệ khác được đưa vào thì test đó sẽ thất bại.

```
[Test]
[ExpectedException(typeof(XYZException))]
public void SomeExceptionThrowingTest()
{
    throw new XYZException();
}
```

- **[Ignore]:** Việc thuộc tính nghi ngờ(*Ignore*) có thể được thêm vào một bài test hay một TestFixture. Đánh dấu một test hay một TestFixture như bị bỏ qua sẽ gây ra các bài test không chạy. Giao diện của NUnit sẽ hiển thị các bài test là màu vàng.

```
[Test]
[Ignore("Don't run this at the moment")]
public void DoSomeTesting()
{
    //some test code
}
```


- **[Category]:** Thuộc tính phạm trù (category) cho phép bạn test từng nhóm theo từng phạm trù, bởi đang áp dụng thuộc tính category vào từng test hay testfixture. Có thể chọn để bao gồm hay loại trừ các phạm trù cụ thể khi đang test các unit test.

```
[Test]
[Category("RequiresDatabase")]
public void DBTest()
{
    //do some test on DB functionality
}
```

4.2.4. Phương thức tĩnh hay dùng trong *Nunit.Framework.Assert*

Trong lớp Assert của thư viện Nunit.Framework có một số phương thức tĩnh để có thể khẳng định tính đúng đắn dẫn cho một số điểm trong bài test:

- `Assert.AreEqual (object, object)`: Là kiểm tra sự bằng nhau bởi cách gọi phương thức `Equal` trên đối tượng.
- `Assert.AreEqual(int, int)`: Là so sánh hai giá trị để kiểm tra bằng nhau. Test chấp nhận nếu các giá trị bằng nhau.
- `Assert.AreEqual(float, float, float)`:
- `Assert.AreEqual(double, double, double)`:
- `Assert.Fail()`: Là hữu ích khi có một bộ test. Test sẽ chấp nhận nếu biểu thức sai.
- `Assert.IsTrue(bool)`: Đánh giá một biểu thức luận lý. Test chấp nhận nếu biểu thức đúng.
- `Assert.IsFalse(bool)`: Đánh giá một biểu thức luận lý. Test chấp nhận nếu biểu thức sai.
- `Assert.IsNull(bool)`: So sánh tham chiếu của đối tượng với giá trị null. Test sẽ được chấp nhận nếu tham chiếu là null.

- `Assert.IsNotNull(bool)`: So sánh tham chiếu của một đối tượng null. Test sẽ chấp nhận nếu biểu thức tham chiếu đối tượng khác nulll.
- `Assert.AreSame(object, object)`: Thực thi một tham chiếu bằng trên hai đối tượng là điểm giống như đối tượng.
- `Assert.Ignore()`: dùng để test trạng thái nghi ngờ
- `Assert.IsEmpty()`: Khẳng định một mảng, danh sách hay một bộ nào đó là rỗng.
- `Assert.IsNotEmpty()`: Khẳng định một mảng, danh sách, hay một bộ nào đó là không rỗng.
- ...

4.2.5. Cài đặt Nunit.

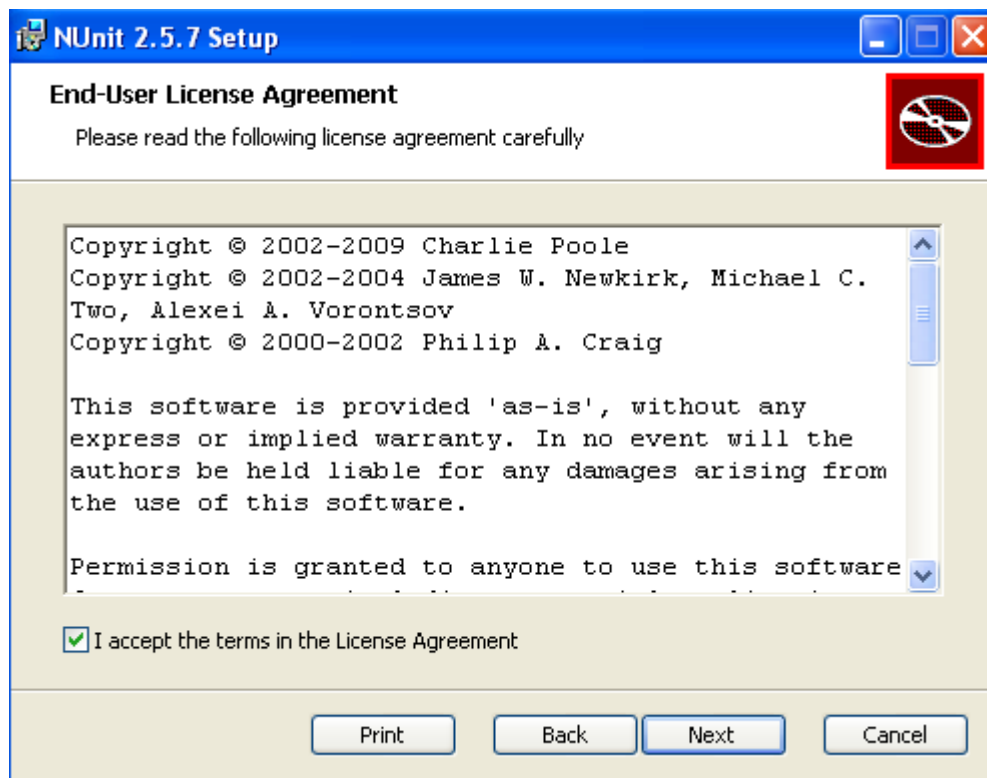
- Để có thể cài đặt được Nunit thì bạn phải có bộ cài của Nunit. Bạn download ở trang chủ của Nunit: <http://www.nunit.org/index.php?p=download>. Ví dụ như phiên bản: *NUnit-2.5.2.9222.msi* hay *NUnit-2.5.7.10213.msi*.

- Các bước cài đặt:

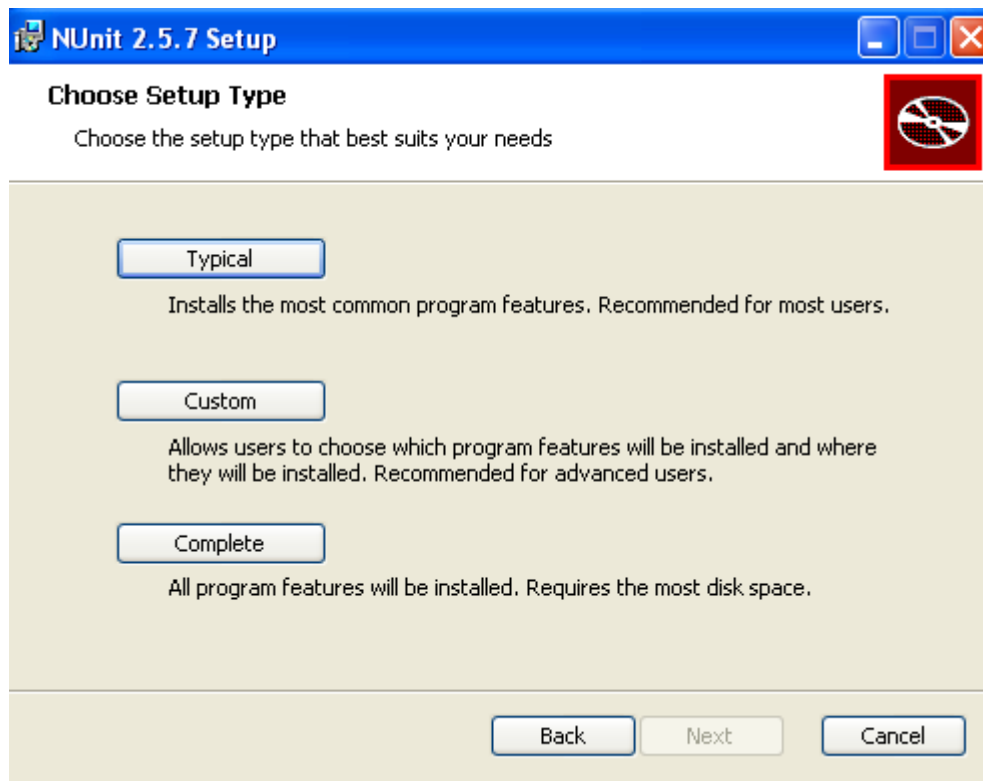
- Bước 1: chạy file Nunit -2.5.2.9222.msi sẽ được như thế này và click Next nhé



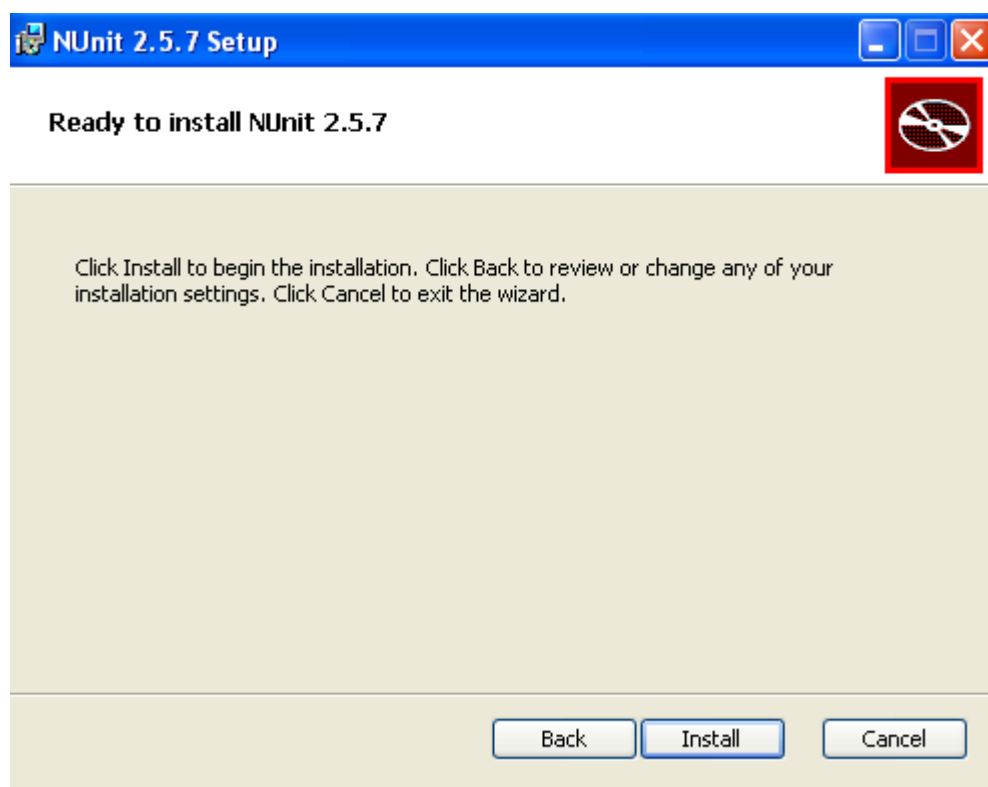
- Bước 2: Tick vào ô “I accept...” và click Next



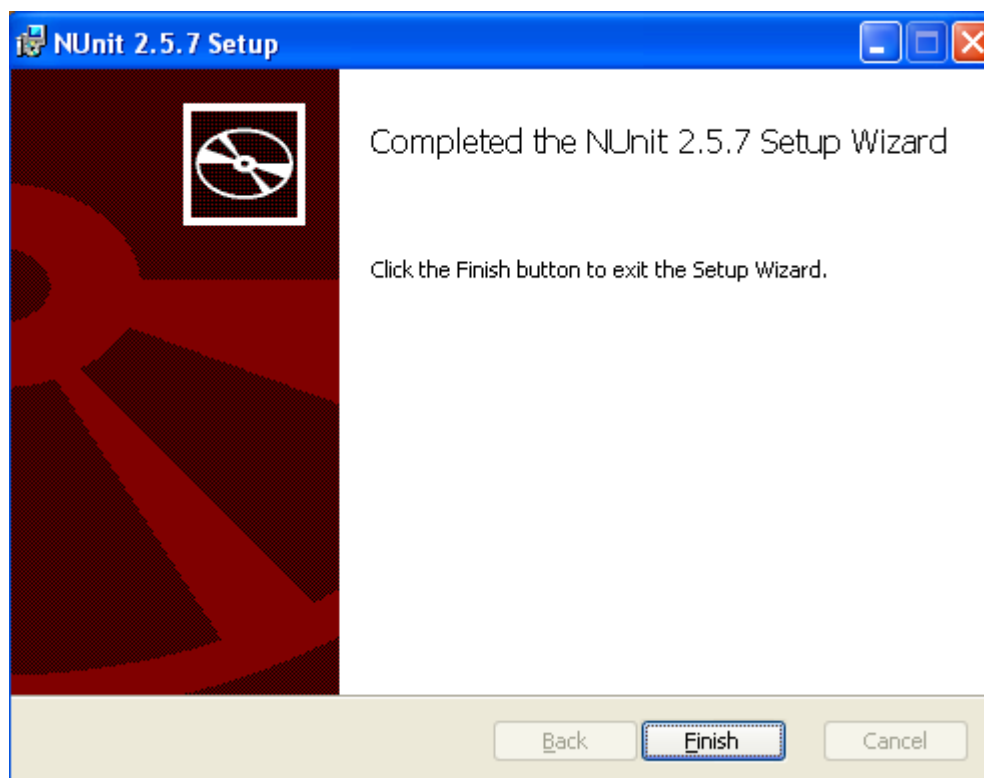
- Chọn vào “Typical” và Click Next



- Bước 4: Click Install và tiếp tục cài đặt.



- Bước 5: quá trình hoàn thành.

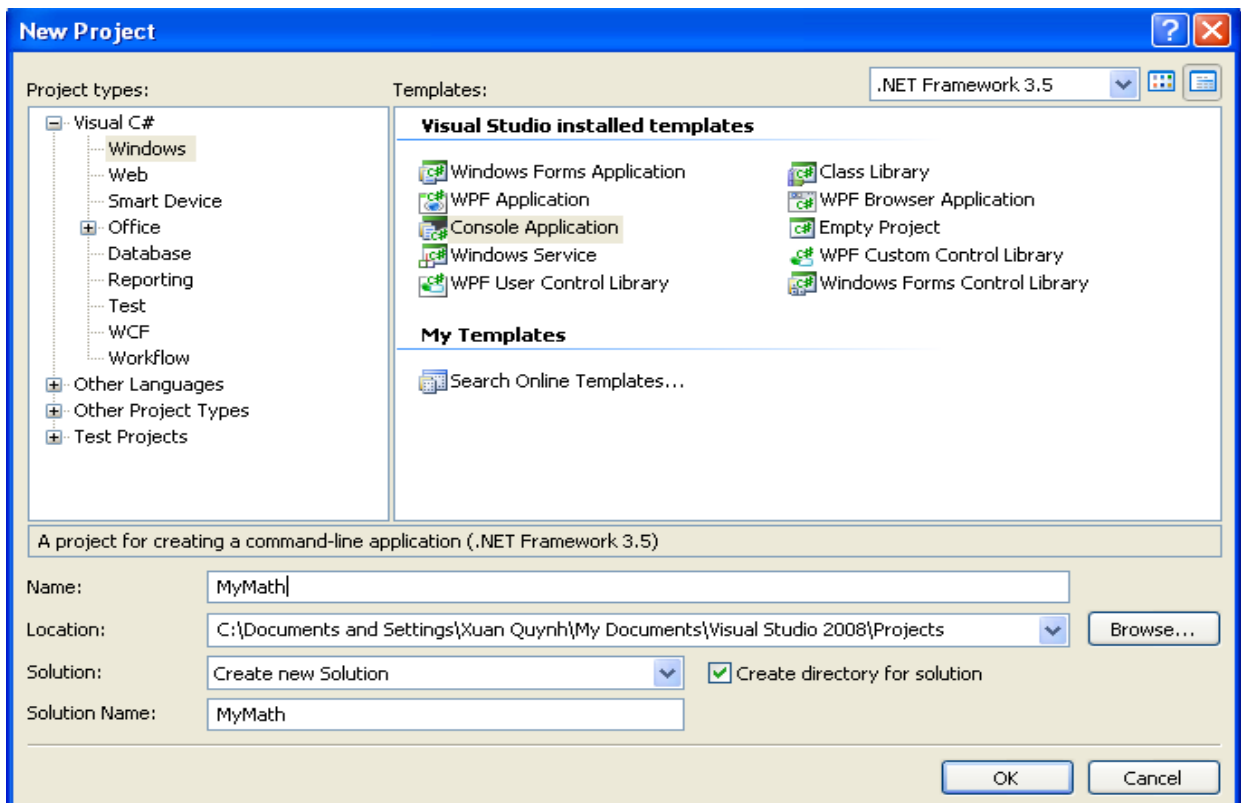


4.2.6. Cách sử dụng Nunit.

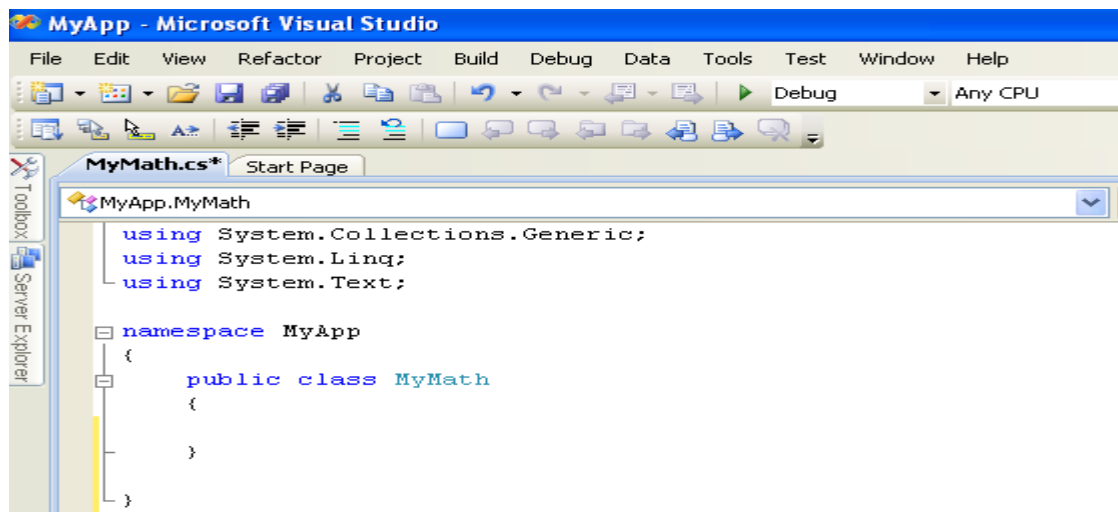
4.2.6.1. Hướng dẫn tạo test case trong Visual studio 2008.

Tạo ra một bộ test:

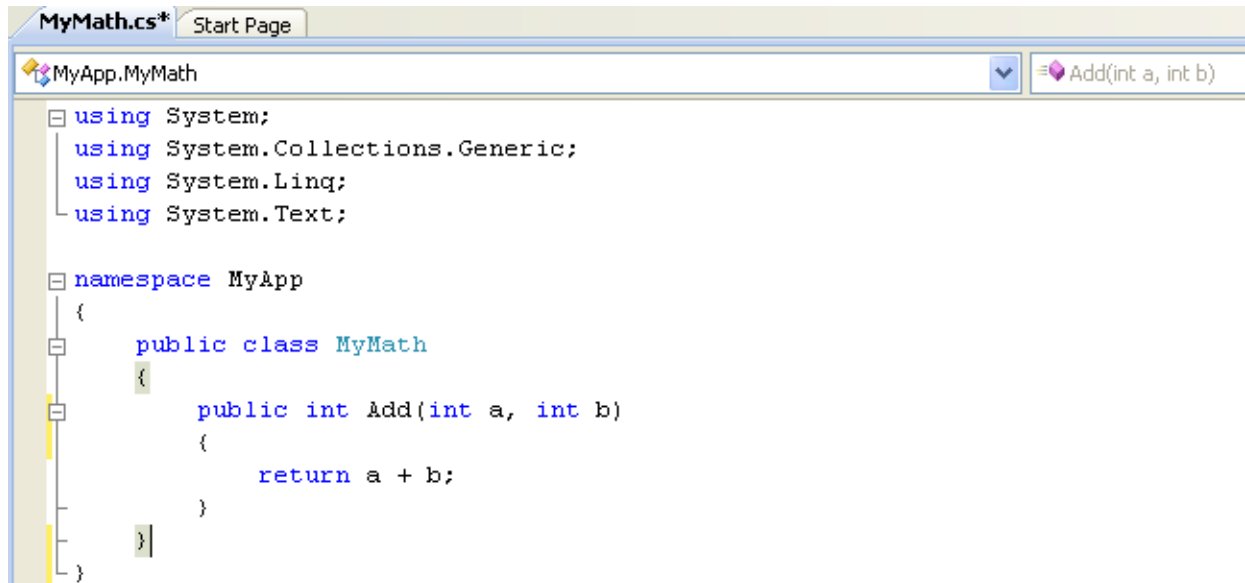
- **Bước 1:** Mở visual studio 2008, tạo ra một Solution mới để dùng cho việc test.
- **Bước 2:** Thêm vào visual C# một project có tên là MyApp vào Solution. Đây là dự án mà bạn có thể chạy thử nghiệm.



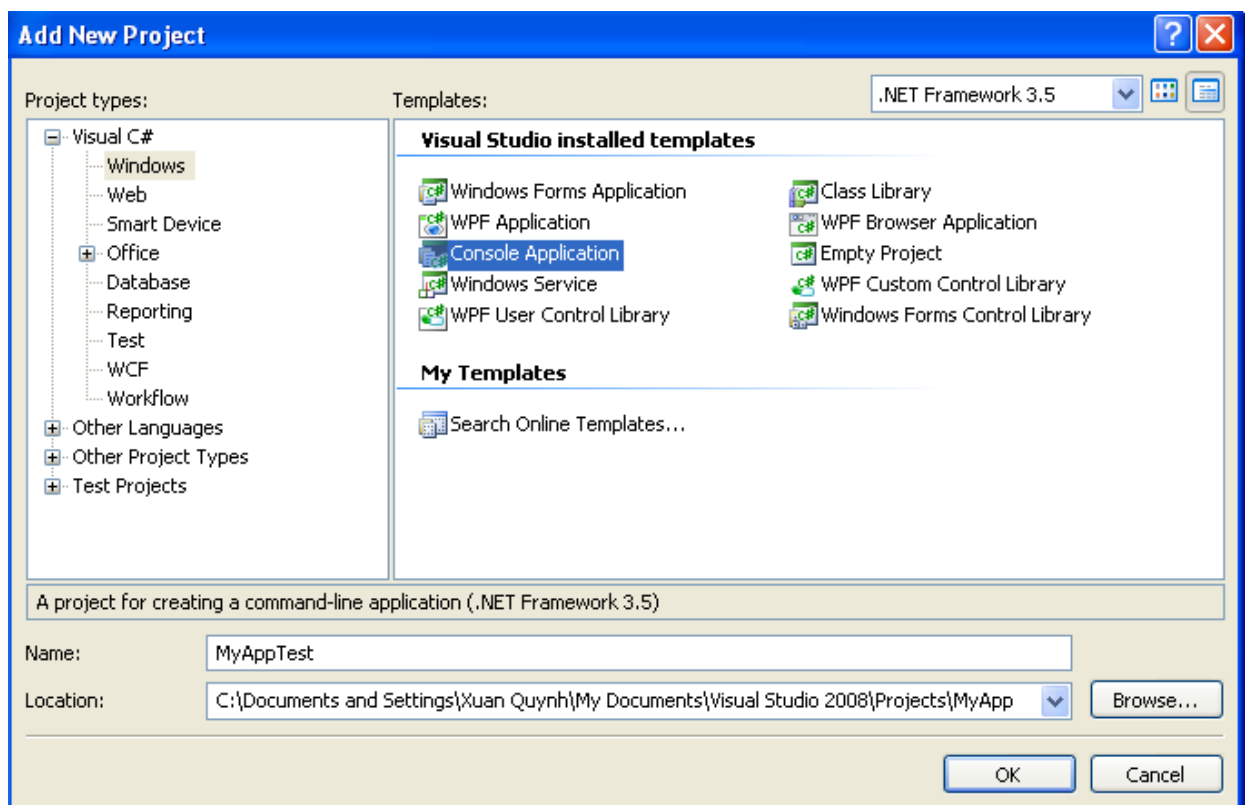
- **Bước 3:** Đổi tên class1 thành MyMath. Thay đổi cách khai báo các lớp:



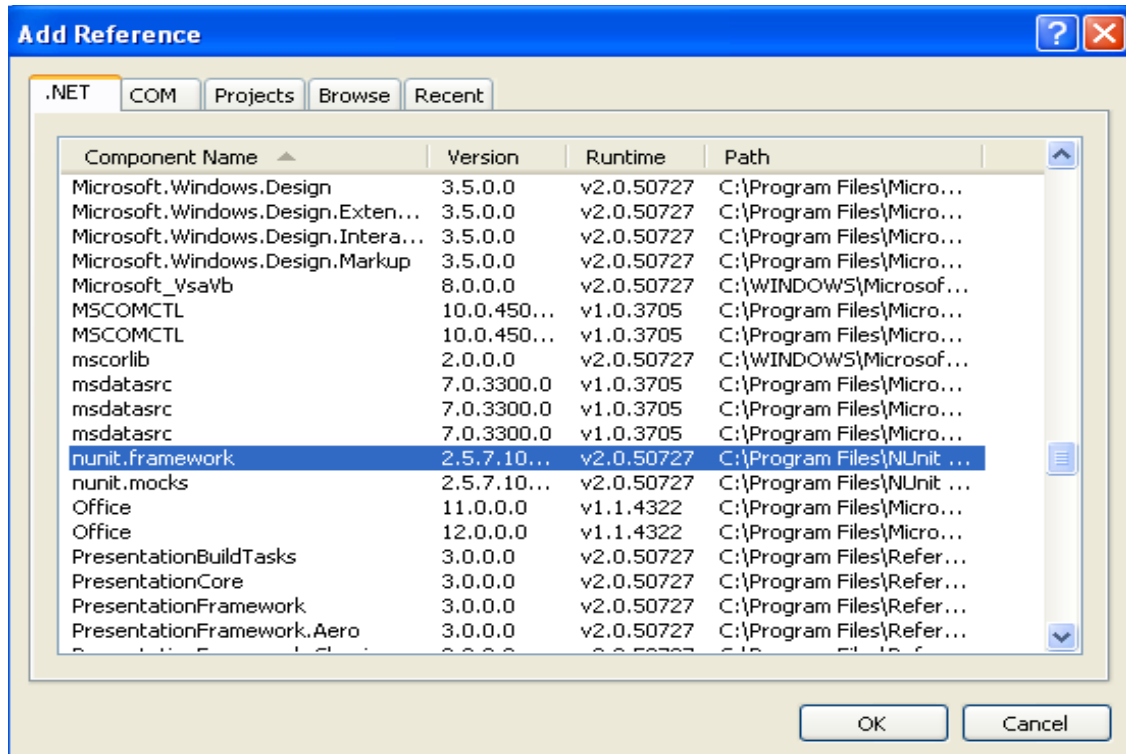
- **Bước 4:** Thêm một phương thức duy nhất là “Add” để cho phép tính tổng của hai số và kết quả trả về là tổng của hai số đó. Hoàn thành lớp đó:



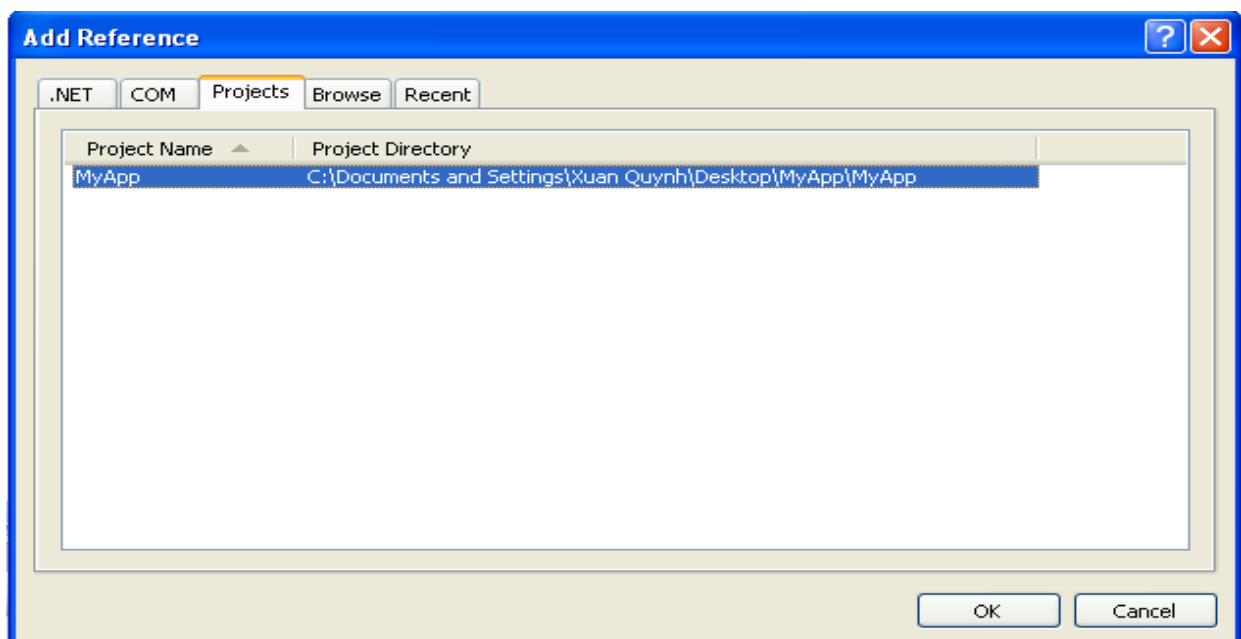
- **Bước 5:** Thêm vào Visual C# một project mới là MyAppTest vào Solution. Đây là dự án sẽ cho phép NUnit kiểm thử. Đó là ý tưởng tốt để sử dụng một tên cho phép các dự án test dễ dàng xác định. (Kích chuột phải vào Solution → Add → New Project → NameProject).



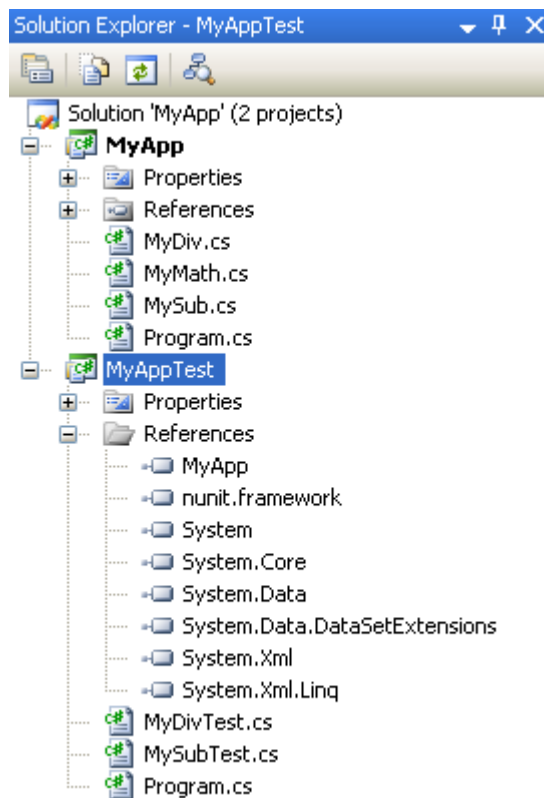
- **Bước 6:** Chọn tham chiếu NUnit framework cho test project MyAppTest; Chọn Project->Add Reference from the Visual Studio .NET IDE menu. In the Add Reference dialog, double-click on *nunit.framework* in the listbox on the .NET tab and click OK.



- **Bước 7:** Thêm một tham chiếu tới các dự án đang được test:MyApp cho dự án test MyAppTest. Select Project →Add Reference. Thêm một tham chiếu tới Nunit Framework: Add reference →Tab Project → Chọn dự án mà đang test →OK.



- **Bước 8:** The Solution Explorer bây giờ như hình vẽ sau:



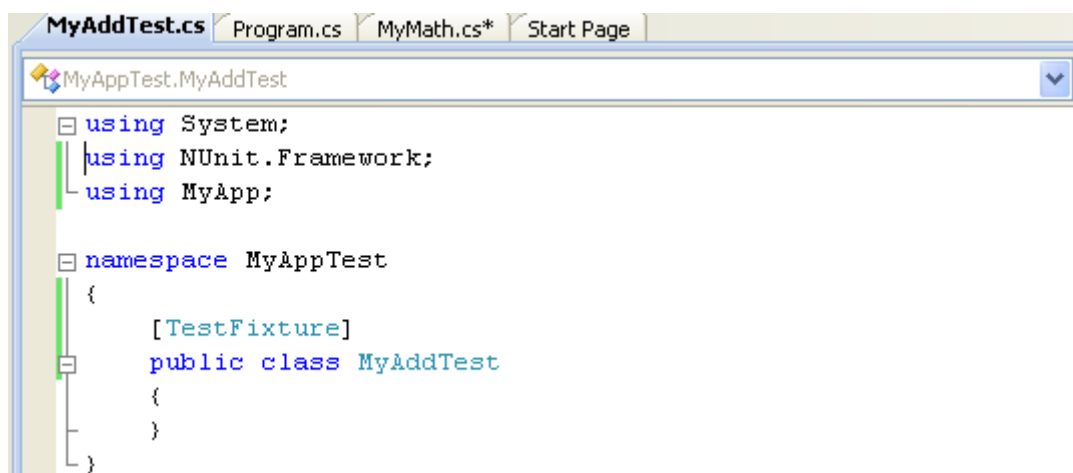
- **Bước 9:** Thêm thư viện Nunit.Framework đến lớp test class1.

Using Nunit.Framework;

- **Bước 10:** Sử dụng namespace MyApp:

Using MyApp;

- **Bước 11:** Thêm một thuộc tính [TestFixture] tới test class MyAddTest để chỉ ra rằng lớp này chứa mã kiểm tra.



- **Bước 12:** Tạo ra một phương thức MyTest trong lớp test. Nhớ phương thức này sử dụng *public* và *void*, và không có đối số truyền vào. Xác định phương thức test bằng việc trang trí nó với thuộc tính [Test].

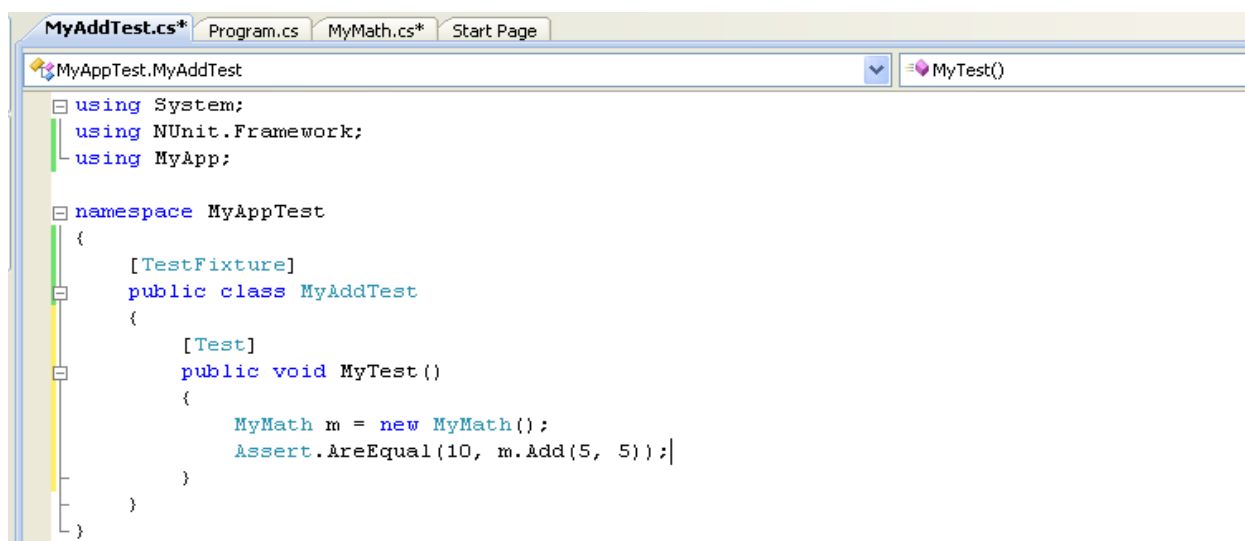
```
[Test]
public void MyTest ()
{
}
```

- **Bước 13:** Viết một test. Trong trường hợp này, kiểm thử phương thức Add trong MyMath.

```
[Test]
public void MyTest ()
{
    MyMath m = new MyMath();
    Assert.AreEqual(10, m.Add(5, 5));
}
```

Đầu ra Tham số
mong đợi truyền vào

- **Hoàn thành một tình huống test.**

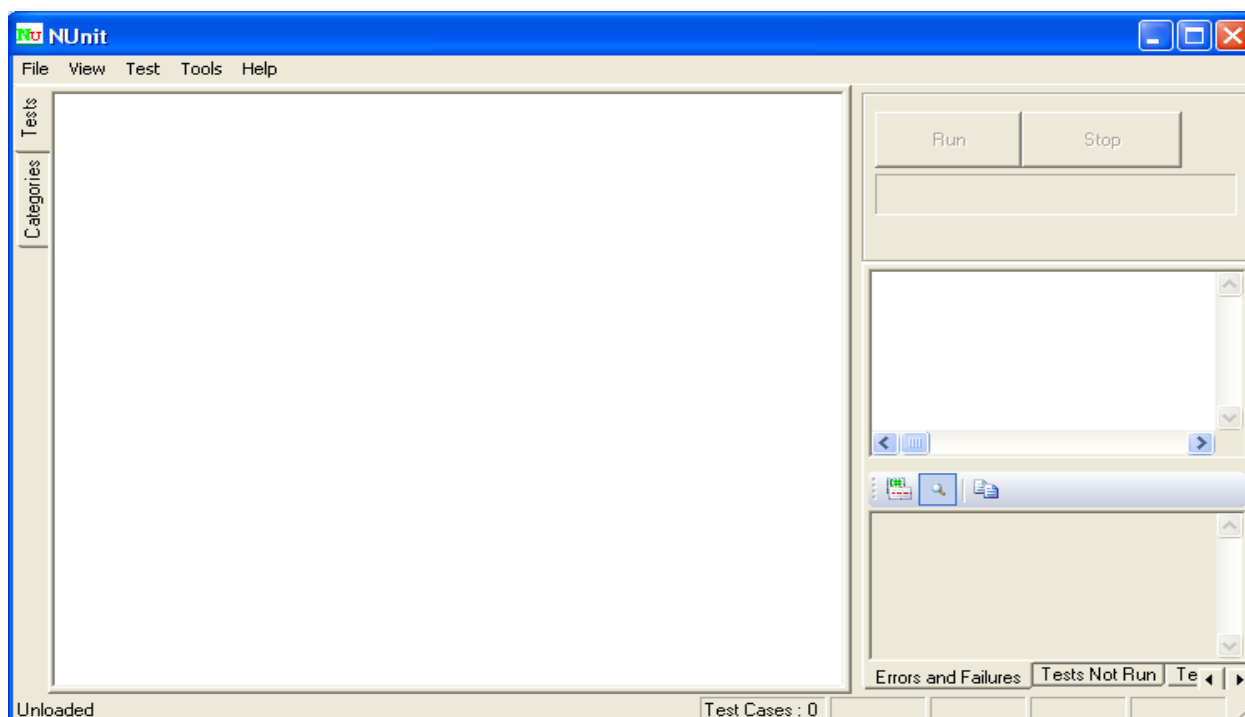


4.2.6.2. Sử dụng Nunit.

Sau khi hoàn thành công việc viết code cho các tình huống test ở trên thì bắt đầu sử dụng NUnit để kiểm tra quá trình viết code tạo ra các testcase ở trên.

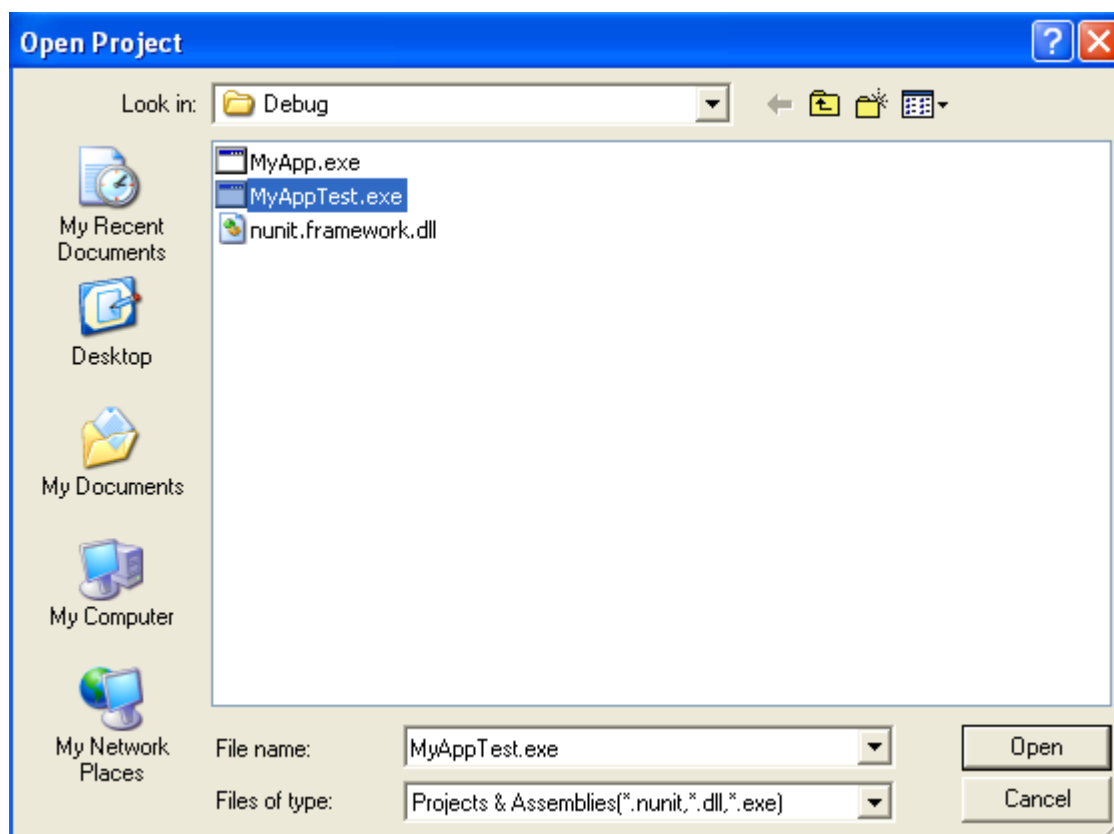
Quá trình test trong Nunit:

- **Bước 1:** Chạy NUnit thì giao diện của NUnit sẽ như sau:

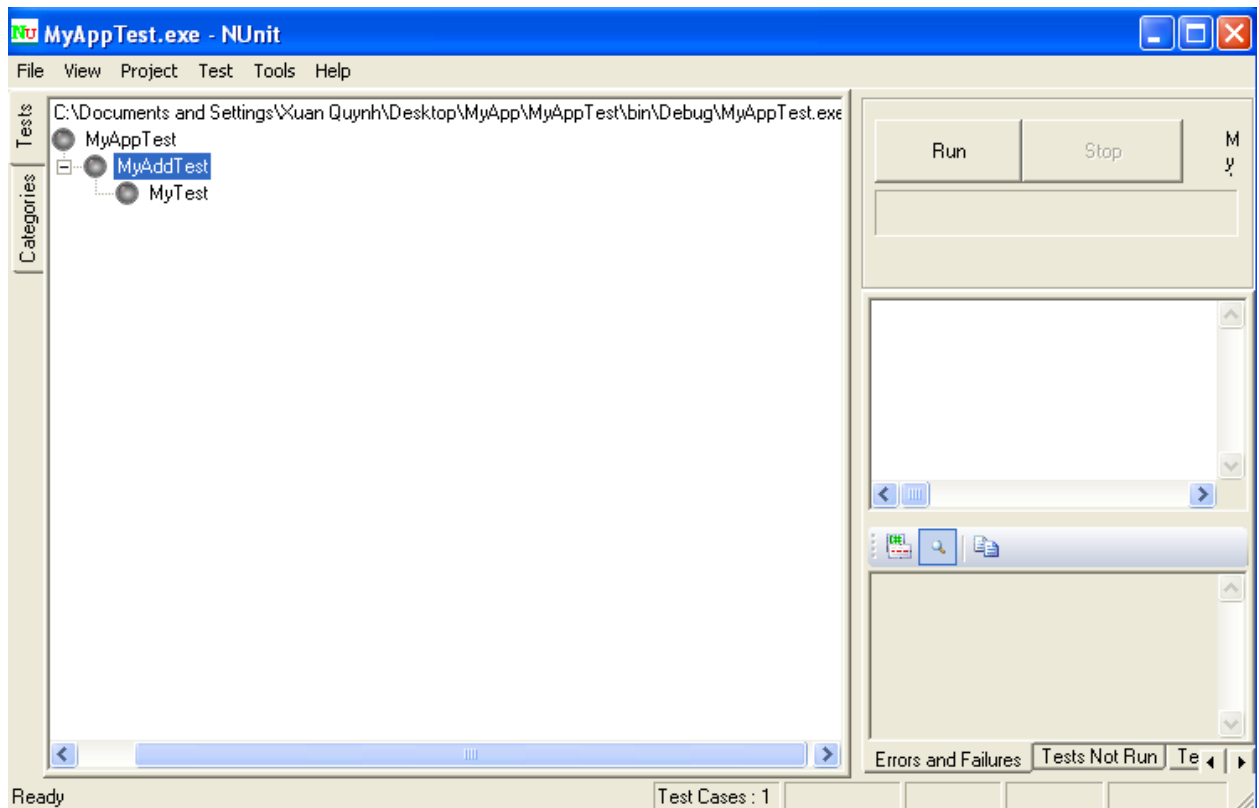


- **Bước 2:** Add project vừa viết vào công cụ Nunit như sau: (trước khi thực hiện công việc Add project thì các bạn phải Built project đó.)

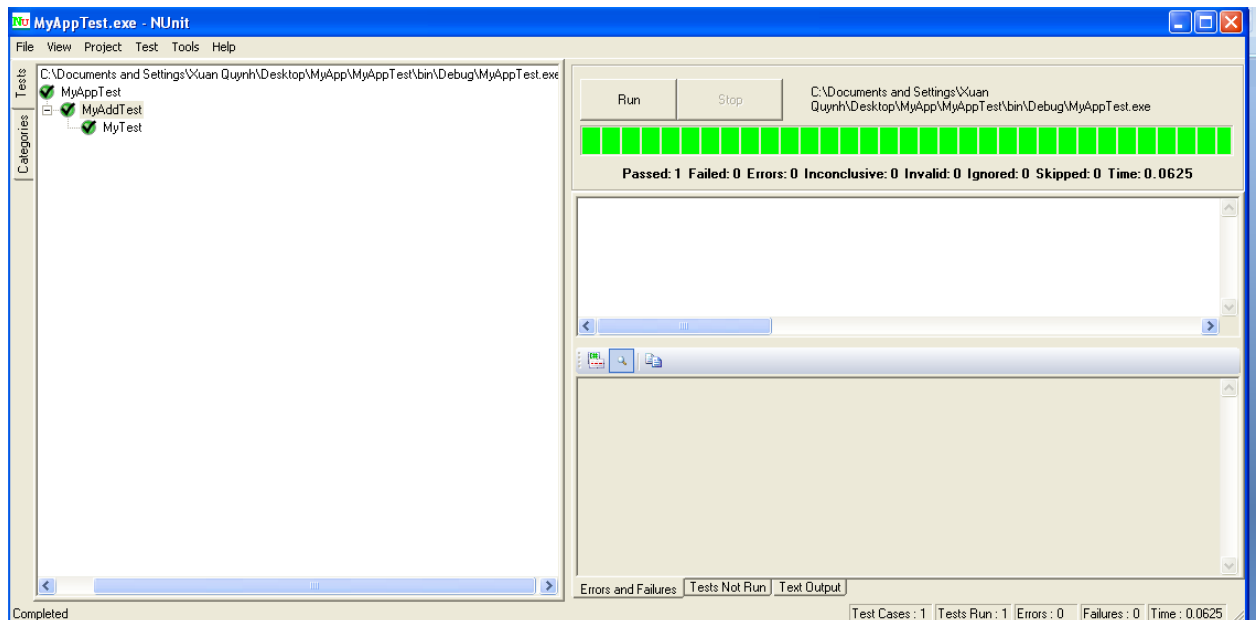
File →Open Project →Tìm tới File MyAppTest.exe(file vừa viết)



- Kích vào MyAppTest.exe → Open. Lúc này giao diện của NUnit như sau:

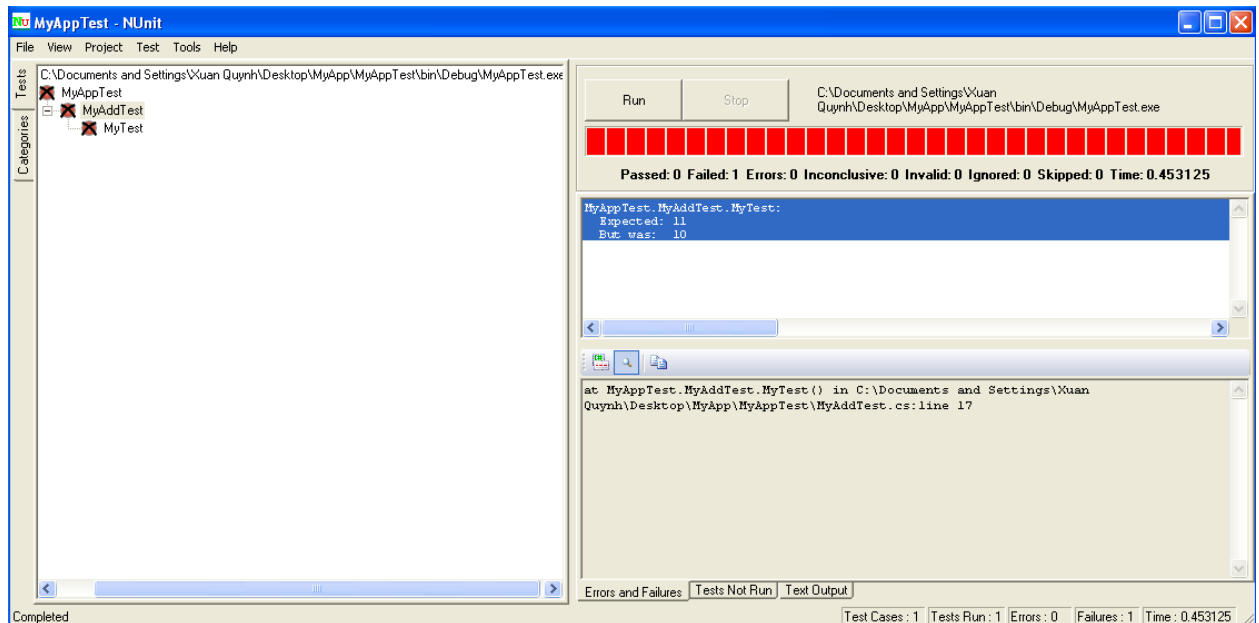


- **Bước 4:** Kiểm tra bằng cách ấn vào Run. Lúc này giao diện của NUnit sẽ hiển thị như sau:



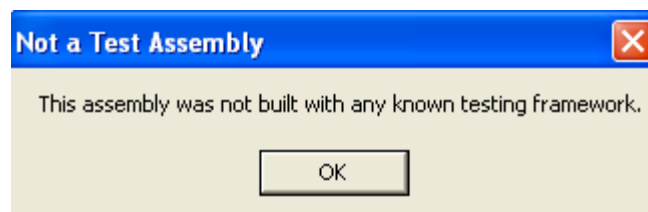
Trong giao diện của NUnit gồm có: MyAppTest là namespace để thực thi, MyAddTest là lớp test, MyTest là phương thức test.

Trường hợp lỗi xảy ra trong khi bạn viết code test case bị sai: Ví dụ như: Bạn đưa sai mong đợi đầu ra của test case đó thì giao diện của NUnit sẽ như sau:



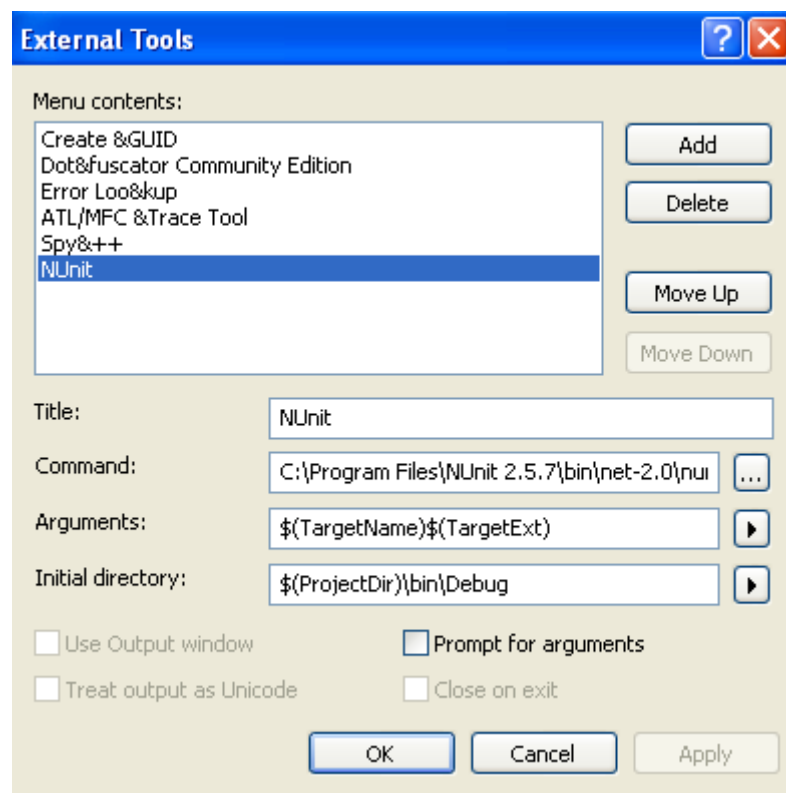
Trong giao diện này có hiển thị: Expected(mong đợi): 11 nhưng But was (thực tế):10.

Chú ý: Có một số trường hợp khi bạn add vào NUnit một file mà nó hiển thị ra như sau:



Cách khắc phục như sau: (add Nunit vào trong Visual Studio)

- Mở project chuẩn bị test ra.
- Thực hiện vào: Menu Tools→ External Tools→Add→...(điền thông số như hình vẽ sau:)



Thực hiện một test case khác cũng làm tương tự như vậy.

CHƯƠNG V: CHƯƠNG TRÌNH DEMO

5.1. Phát biểu bài toán.

Sử dụng công cụ NUnit để kiểm thử các lớp và phương thức của ứng dụng các phép toán trong toán học. Những phép toán trong toán học như cộng, trừ, nhân, chia, hàm lượng giác, một số hàm mũ, hàm căn thức...

5.2. Đặt vấn đề.

- Mỗi sản phẩm phần mềm được tạo ra thì cần phải trải qua những công đoạn của phần mềm như phân tích thiết kế, lập trình, kiểm thử phần mềm.
- Kiểm thử phần mềm là khâu quan trọng trong việc phát triển phần mềm. Thực hiện kiểm tra, thẩm định lại toàn bộ hệ thống của sản phẩm phần mềm sau khi hoàn thiện.
- Ứng dụng những phép toán trong toán học sẽ được kiểm thử bằng cách thủ công như dùng máy tính điện tử số, máy tính tính cá nhân. Bây giờ, thay bằng viết kiểm thử thủ công ta có thể viết các test case để đưa vào ứng dụng rồi sử dụng công cụ để có thể kiểm thử một cách dễ dàng hơn.

5.3. Phân tích và thiết kế bài toán.

- Bài toán kiểm tra các phép toán trong toán học như cộng, trừ, nhân, chia, hàm lượng giác, hàm mũ, hàm căn thức... bằng cách đưa ra các test case và sử dụng công cụ NUnit.
- Với bài toán này thì thiết kế các phép toán như cộng, trừ, nhân, chia, các phép toán lượng giác, phép toán hàm căn, hàm mũ...
 - ❖ Phép cộng: đưa vào phép toán có số hạng thứ nhất, số hạng thứ hai và trả ra kết quả là tổng của hai số hạng. Những kiểu số có thể đưa vào: số nguyên dương, số nguyên âm, số 0, số thập phân âm, số thập phân dương.
 - ❖ Phép trừ: đưa vào phép toán là số trừ, số bị trừ và kết quả trả về là hiệu của phép trừ đó. Những kiểu số có thể đưa vào: số nguyên dương, số nguyên âm, số 0, số thập phân âm, số thập phân dương.
 - ❖ Phép nhân: đưa vào phép toán là số hạng thứ 1, số hạng thứ 2 và kết quả trả về là tích của hai số hạng đó. Những kiểu số có thể đưa vào: số nguyên dương, số nguyên âm, số 0, số thập phân âm, số thập phân dương.

- ❖ Phép chia: đưa vào phép chia là số chia, số bị chia và kết quả trả về là thương của phép chia đó. Những kiểu số có thể đưa vào: số nguyên dương, số nguyên âm, số 0, số thập phân âm, số thập phân dương.
- ❖ Hàm sin: đưa vào phép toán là một số được mặc định là độ(°). Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm cosin: đưa vào phép toán là một số được mặc định là độ(°). Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm tag: đưa vào phép toán là một số được mặc định là độ(°). Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm cotag: đưa vào phép toán là một số được mặc định là độ(°). Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm mũ 2 và hàm mũ 3: đưa vào phép toán một số. Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm x mũ y: đưa vào phép toán là hai số trong đó x là cơ số, y là số mũ. Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm căn bậc 2 và bậc 3: đưa vào phép toán là một số. Những kiểu số có thể đưa vào là kiểu số double.
- ❖ Hàm căn bậc x của y: đưa vào phép toán là hai số. Những kiểu số có thể đưa vào là kiểu số double.

5.4. Thiết kế các test case.

- Sau quá trình phân tích bài toán thì ta thực hiện công việc thiết kế các test case cho từng lớp trong toàn bộ hệ thống.
- Thiết kế test case cho từng hàm toán học.

- ❖ Phép cộng.

	Input	
--	-------	--

STT testcase	Số hạng 1	Số hạng 2	Output
1	10	5	15
2	-10.53	-2	-12.53
3	10	-1.25	8.75
4	0	2	0

❖ Phép trừ.

STT Test case	Input		Output
	Số trừ	Số bị trừ	
1	10.3	5	5.3
2	10.25	12	-1.75
3	-10	-5	-5
4	-5	-10	5
5	10.1	-5.63	15.73
6	-10.31	6.35	-16.66

❖ Phép nhân.

STT Test case	Input		Output
	Số hạng 1	Số hạng 2	
1	2.5	2	5
2	5	-2.5	-12.5
3	-2.5	-3	7.5
4	3	0	0

❖ Phép chia.

STT Testcase	Input		Output
	Số chia	Số bị chia	
1	12.5	5	2.5
2	-10	2.5	-4
3	-12	-3	4
4	0	25	0
5	20	0	NaN

❖ Hàm mũ 2

STT Test case	Input	Output
1	10	100
2	-5.25	27.5625
3	0	0

❖ Hàm mũ 3

STT Test case	Input	Output
1	2.5	15.625
2	-2.6	- 17.576
3	0	0

❖ Hàm x mũ y

	Input	
--	-------	--

STT Testcase	Cơ số x	Lũy thừa y	Output
1	2.5	2	6.25
2	-6	2	36
3	2.5	3	15.625
4	-1.5	3	-3.375
5	1.5	1.5	1.8371173070873836
6	0	4	0
7	12	0	1

❖ Hàm sin

STT Test case	Input (Độ)	Output
1	0	0
2	30	0.5
3	45	Math.Sqrt(2)/2
4	60	Math.Sqrt(3)/2
5	90	1
6	180	0

❖ Hàm cosin

STT Test case	Input (Độ)	Output
1	0	1
2	30	Math.Sqrt(3)/2
3	45	Math.Sqrt(2)/2

4	60	0.5
5	90	0
6	180	-1

❖ Hàm tag

STT Test case	Input (Độ)	Output
1	0	0
2	30	Math.Sqrt(3)/3
3	45	1
4	60	Math.Sqrt(3)
5	90	NaN
6	180	0

❖ Hàm cotag

STT Test case	Input (Độ)	Output
1	0	NaN
2	30	Math.Sqrt(3)
3	45	1
4	60	Math.Sqrt(3)/3
5	90	0
6	180	NaN

❖ Hàm căn bậc 2

STT Test case	Input	Output
1	4	2
2	6.25	2.5
3	0	0

❖ Hàm căn bậc x của y.

STT Test case	Input		Output
	Số x	Căn bậc y	
1	1	2,3	1
2	8	3	2
3	6.25	2	2.5
4	27.9841	4	2.3
5	2.8284271247461900976033774484194	1.5	2
6	0	2	0
7	1	0	1
8	2.3	1	2.3

5.5. Ứng dụng chương trình

- Các hàm cộng, trừ, nhân, chia tương ứng với các lớp: MyAdd, MySub, MyMulti, MyDiv như sau:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Project5
{
    //hàm cộng
    public class MyAdd
    {
        public double Add(double a, double b)
        {
            return a + b;
        }
    }
    //hàm trừ
    public class MySub
    {
        public double Sub(double a, double b)
        {
            return a - b;
        }
    }
    //hàm nhân
    public class MyMulti
    {
        public double Multi(double a, double b)
        {
            return a * b;
        }
    }
    //hàm chia
    public class MyDiv
    {
        public double Div(double a, double b)
        {
            return (double)(a / b);
        }
    }
}
```

- Các hàm mũ như hàm mũ 2, hàm mũ 3, hàm mũ n như sau:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Project5
{
    //hàm mũ 2
    public class MyHamMu
    {
        public double Mu2(double a)
        {
            return Math.Pow(a, 2);
        }
    }
    //hàm mũ 3
    public class MyHamMu3
    {
        public double Mu3(double a)
        {
            return Math.Pow(a, 3);
        }
    }
    //hàm x mũ y
    public class MyHamXMuY
    {
        public double XmuY(double x, double y)
        {
            return Math.Pow(x, y);
        }
    }
}
```

- Các hàm lượng giác như hàm sin, hàm cos, hàm tan, hàm cotan như sau:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Project5
{
    public class MyHamLuongGiac
    {
        public double Sin(double a)
        {
            return (double)Math.Sin(Math.PI * a / 180);
        }
    }
}
```

```

    }
    public class MyCosin
    {
        public double CoSin(double a)
        {
            return Math.Cos(Math.PI * a / 180);
        }
    }
    public class MyTag
    {
        public double Tag(double a)
        {
            return Math.Sin(Math.PI * a / 180) / Math.Cos(Math.PI * a / 180);
        }
    }
    public class MyCoTag
    {
        public double CoTag(double a)
        {
            return Math.Cos(Math.PI * a / 180) / Math.Sin(Math.PI * a / 180);
        }
    }
}

```

- Các hàm căn thức như hàm căn bậc 2, hàm căn bậc n như sau:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Project5
{
    public class MyHamCanThuc
    {
        public double CanBac2(double a)
        {
            return Math.Sqrt(a);
        }
    }
    public class MyCanBacN
    {
        public double CanBacN(double a, double b)
        {
            return Math.Pow(a, (double)(1 / b));
        }
    }
}

```

5.6. Tổng kết chương trình demo

Trong khi viết chương trình demo để có thể dùng công cụ NUnit kiểm thử các thành phần, các lớp, các phương thức ... trong bài toán. NUnit có thể kiểm thử được cơ sở dữ liệu, giao diện,... của một phần mềm. Nhưng do thời gian nghiên cứu còn hạn chế nên em chưa thể dùng công cụ Nunit để kiểm thử được giao diện, hay cơ sở dữ liệu. Nếu trong thời gian tới em vẫn tiếp tục được làm về đề tài này em sẽ hoàn chỉnh hơn việc kiểm thử về nhiều mặt hơn. Còn chương trình demo này, em mới chỉ kiểm tra được các thành phần, lớp của bài toán nhỏ ở mức đơn vị.

Những kết quả đạt được trong chương trình demo:

- Sử dụng được các thuộc tính như: TestFixture, Test... trong thư viện NUnit.Framework
- Sử dụng được lớp Assert và các phương thức trong lớp Assert như: AreEqual, IsNaN, IsNotNull...
- Tạo ra những bộ Test – case cho bài toán cụ thể.

Những việc chưa làm được trong chương trình demo:

- Chưa sử dụng hết được các phương thức trong lớp Assert.
- Chương trình demo ở quy mô nhỏ nên chưa dùng hết các thuộc tính trong thư viện NUnit. Framework.
- ...

PHẦN III:PHẦN KẾT LUẬN

Kiểm thử phần mềm, một hướng đi không còn mới mẻ trên thế giới, nhưng lại là một hướng đi rất mới ở Việt Nam. Nó hứa hẹn một tương lai mới cho các học sinh, sinh viên ngành Công Nghệ Thông Tin.

Qua quá trình tìm hiểu vào xây dựng đề tài này, chúng em đã hiểu thêm nhiều về kiểm thử phần mềm, kiểm thử dự án và sử dụng công cụ để kiểm thử. Trong đó có một số những mặt cần phát huy như:

- Hiểu tổng quan về kiểm thử phần mềm gồm: các khái niệm, tính chất, tác dụng...
- Hiểu và sử dụng được công cụ kiểm thử Nunit.
- Hiểu về các chiến lược kiểm thử, các phương pháp kiểm thử để từ đó xây dựng được các test – case cho bài toán cụ thể là bài toán “***Các phép toán trong toán học***”.
- Có khả năng đọc hiểu tài liệu tiếng anh một cách thành thạo hơn.

Và một số điểm còn hạn chế:

- Kiến thức tìm hiểu về hiểu biết về Nunit còn chưa được mở rộng nên chỉ dừng lại ở việc thiết kế test – case cho bài toán nhỏ là: “***Các phép toán trong toán học***”.

Một lần nữa chúng em xin chân thành cảm ơn tới cô *Lê Thị Thu Hương* – giáo viên hướng dẫn đã nhiệt tình chỉ bảo chúng em trong suốt thời gian qua.

Chúng em xin chân thành cảm ơn!

Sinh viên thực hiện

Đỗ Thùy Dung

Nguyễn Thị Huệ

Nguyễn Thị Hương

TÀI LIỆU THAM KHẢO

- [1]. *Pragmatic Unit Testing In C# with NUnit* – Andrew Hunt and David Thomas
- [2]. *Unit Testing in BlueJ – version 1.0 for BlueJ Version 1.3.0* – Michael Kolling and Mærsk Institute – University of Southern Denmark.
- [3]. *Unit Testing A Guide* – Mark R.Dawson.
- [4]. *Unit testing with Mock Objects* – Tim Mackinnon, Steve Freeman, Philio Craig.
- [5]. *The Art of Unit Testing with Example in .NET* – Roy Osherove.
- [6]. *Software Unit Testing* – Rodney Parkin – IV&V Australia.
- [7]. *Professional Software Testing with Visual Studio 2005* – Team System Tools for Software Developers and Test Engineers Programmer to Programmer.
- [8]. *Preventing Bugs with Unit Testing*.
- [9]. *Một số video hướng dẫn lập Unit test sử dụng công cụ NUnit*.
- [10]....