

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  
KHOA CÔNG NGHỆ THÔNG TIN  
-----o0o-----

**Thạc Bình Cường**

**Bài giảng điện tử môn học**

**KIỂM THỬ VÀ BẢO ĐẢM  
CHẤT LƯỢNG PHẦN MỀM**

<b>MỞ ĐẦU.....</b>	<b>4</b>
<b>CHƯƠNG 1: CÁC KHÁI NIỆM .....</b>	<b>5</b>
1.1. Các định nghĩa.....	5
1.2. Vòng đời của việc kiểm nghiệm (testing life cycle):.....	6
1.3. Phân loại kiểm nghiệm: .....	7
1.4. Sự tương quan giữa các công đoạn xây dựng phần mềm và loại kiểm nghiệm: Mô hình chữ V.....	8
1.5. Sơ lược các kỹ thuật và công đoạn kiểm nghiệm:.....	9
<b>CHƯƠNG 2: KIỂM CHỨNG VÀ XÁC NHẬN (V &amp; V ) .....</b>	<b>13</b>
2.1. Kiểm chứng và hợp lệ hoá.....	13
2.1.1. Tổ chức việc kiểm thử phần mềm .....	14
2.1.2. Chiến lược kiểm thử phần mềm .....	15
2.1.3. Tiêu chuẩn hoàn thành kiểm thử .....	17
2.2. Phát triển phần mềm phòng sạch (cleanroom software development).....	18
2.2.1. Nghệ thuật của việc gỡ rối.....	18
2.2.2. Tiến trình gỡ lỗi.....	18
2.2.3. Xem xét tâm lý .....	19
2.2.4. Cách tiếp cận gỡ lỗi .....	19
<b>CHƯƠNG 3: KIỂM THỬ PHẦN MỀM.....</b>	<b>22</b>
3.1. Quá trình kiểm thử.....	22
3.2. Kiểm thử hệ thống .....	24
3.3. Kiểm thử tích hợp .....	25
3.4. Kiểm thử phát hành .....	27
3.5. Kiểm thử hiệu năng .....	31
3.6. Kiểm thử thành phần .....	32
3.7. Kiểm thử giao diện .....	33
3.8. Thiết kế trường hợp thử (Test case design) .....	35
3.9. Tự động hóa kiểm thử (Test automation).....	45
<b>CHƯƠNG 4: CÁC PHƯƠNG PHÁP KIỂM THỬ .....</b>	<b>49</b>
4.1. Phương pháp white-box:.....	50
4.2. Phương pháp black-box:.....	59
<b>CHƯƠNG 5: KIỂM THỬ TÍCH HỢP.....</b>	<b>66</b>
5.1. Tích hợp trên xuống. ....	66
5.2. Tích hợp dưới lên. ....	68
5.3. Kiểm thử nội quy.....	69
5.4. Gợi ý về việc kiểm thử tích hợp .....	71
5.5. Lập tài liệu về kiểm thử tích hợp.....	72
<b>CHƯƠNG 6: KỸ NGHỆ ĐỘ TIN CẬY PHẦN MỀM .....</b>	<b>75</b>
6.1. Giới thiệu.....	75
6.2. Xác nhận tính tin cậy .....	76
6.2.1. Sơ thảo hoạt động .....	78
6.2.2. Dự đoán tính tin cậy .....	79
6.3. Đảm bảo tính an toàn.....	82
6.3.1. Những luận chứng về tính an toàn.....	83
6.3.2. Đảm bảo quy trình .....	86
6.3.3. Kiểm tra tính an toàn khi thực hiện .....	88
6.4. Các trường hợp an toàn và tin cậy được .....	89

<b>CHƯƠNG 7: KIỂM THỬ PHẦN MỀM TRONG CÔNG NGHIỆP .....</b>	<b>95</b>
7.1. QUY TRÌNH KIỂM TRA PHẦN MỀM CƠ BẢN.....	95
7.2. MÔ HÌNH KIỂM TRA PHẦN MỀM TMM (TESTING MATURITY MODEL).....	99
7.3. Các công cụ kiểm thử (Test tools).....	105
7.3.1. TẠI SAO PHẢI DÙNG TEST TOOL .....	105
7.3.2. KHÁI QUÁT VỀ KTTĐ .....	106
7.3.3. GIỚI THIỆU CÔNG CỤ KTTĐ: QUICKTEST PROFESSIONAL.....	108
7.3.4. Kiểm thử đơn vị với JUnit .....	112
<b>CHƯƠNG 8: ƯỚC LƯỢNG GIÁ THÀNH PHẦN MỀM.....</b>	<b>129</b>
8.1. Giới thiệu .....	129
8.2. Năng suất phần mềm .....	131
8.3. Kỹ thuật ước lượng .....	135
8.4. Mô hình hoá chi phí thuật toán.....	137
8.5. Mô hình COCOMO .....	139
8.6. Mô hình chi phí giải thuật trong kế hoạch dự án.....	147
8.7. Nhân viên và khoảng thời gian của dự án .....	149
<b>CHƯƠNG 9: QUẢN LÝ CHẤT LƯỢNG PHẦN MỀM .....</b>	<b>153</b>
9.1. Chất lượng quá trình và chất lượng sản phẩm:.....	153
9.2. Chất lượng quá trình và chất lượng sản phẩm:.....	155
9.3. Đảm bảo chất lượng và các chuẩn chất lượng.....	156
9.4. Lập kế hoạch chất lượng.....	163
9.5. Kiểm soát chất lượng.....	164
9.6. CMM/CMMi .....	165
9.6.2. Cấu trúc của CMM .....	166
9.6.3. So sánh giữa CMM và CMMi .....	172
<b>CHƯƠNG 10: QUẢN LÝ CẤU HÌNH.....</b>	<b>174</b>
10.1. Giới thiệu .....	174
10.2. Kế hoạch quản trị cấu hình.....	176
11.2. Quản lý việc thay đổi.....	179
11.3. Quản lý phiên bản và bản phát hành.....	183
11.4. Quản lý bản phát hành .....	186
11.5. Xây dựng hệ thống .....	189
11.6. Các công cụ CASE cho quản trị cấu hình .....	190
<b>PHỤ LỤC- CÁC CÂU HỎI ÔN TẬP.....</b>	<b>197</b>
1. Chất lượng và đảm bảo chất lượng phần mềm.....	197
2. Các độ đo đặc trưng chất lượng phần mềm.....	198
3. Kiểm thử phần mềm .....	199
4. Quản lý cấu hình phần mềm.....	201
<b>TÀI LIỆU THAM KHẢO.....</b>	<b>202</b>

## MỞ ĐẦU

Quản lý chất lượng phần mềm là vấn đề không mới nhưng theo một số đánh giá là còn yếu của các công ty phần mềm Việt Nam. Một số công ty trong nước hiện đã đạt các chuẩn quốc tế CMM/CMMI trong nâng cao năng lực và quản lý chất lượng phần mềm, song chỉ đếm được trên đầu ngón tay, và hiện cũng chỉ gói gọn trong vài công ty gia công cho thị trường nước ngoài.

Lâu nay, nói đến chất lượng phần mềm, không ít người nghĩ ngay đến vấn đề là xác định xem phần mềm đó có phát sinh lỗi hay không, có "chạy" đúng như yêu cầu hay không và cuối cùng thường quy về vai trò của hoạt động kiểm thử phần mềm (testing) như là hoạt động chịu trách nhiệm chính.

Với quan điểm của khách hàng, điều này có thể đúng, họ không cần quan tâm nội tình của hoạt động phát triển phần mềm, điều họ cần quan tâm là liệu sản phẩm cuối cùng giao cho họ có đúng hạn hay không và làm việc đúng như họ muốn hay không.

Tuy nhiên theo quan điểm của người phát triển phần mềm, thực tế cho thấy hoạt động kiểm thử phần mềm là quan trọng, nhưng không đủ để đảm bảo sản phẩm sẽ được hoàn thành đúng hạn và đúng yêu cầu. Kiểm thử sau cùng để phát hiện lỗi là điều tất nhiên phải làm, nhưng trong rất nhiều trường hợp, điều đó thường quá trễ và sẽ phải mất rất nhiều thời gian để sửa chữa.

Thực tế cho thấy, để đảm bảo được hai tiêu chí "đơn giản" trên của khách hàng, đòi hỏi tổ chức không chỉ vận hành tốt khâu kiểm thử phần mềm, mà phải tổ chức và duy trì sự hoạt động nhịp nhàng của cả một hệ thống các công việc liên quan đến một dự án phần mềm, từ đây xuất hiện một khái niệm có tên là "hệ thống quản lý chất lượng phần mềm" bao gồm các quy trình được thực thi xuyên suốt chu kỳ phát triển của dự án phần mềm song hành cùng việc kiểm thử phần mềm nhằm đảm bảo chất lượng cho phần mềm khi chuyển giao cho khách hàng.

Với thực tế trên, là những người làm công tác đào tạo mong muốn cung cấp cho sinh viên ngành công nghệ phần mềm - những người sẽ là nguồn nhân lực chủ yếu trong tương lai của các doanh nghiệp phần mềm – những khái niệm, kiến thức và kỹ năng cơ bản ban đầu về kiểm thử phần mềm, về quy trình quản lý chất lượng, đảm bảo chất lượng phần mềm thông qua giáo trình (nội bộ) Kiểm thử và đảm bảo chất lượng phần mềm (Software Testing and Quality Assurance).

Giáo trình này với mục tiêu cung cấp cho sinh viên công nghệ phần mềm có được kiến thức và kỹ năng về việc kiểm thử phần mềm, các công đoạn kiểm thử, các loại kiểm thử, công cụ kiểm thử, xây dựng tài liệu kiểm thử, dữ liệu kiểm thử .... Và xây quy trình đảm bảo chất lượng phần mềm, giới thiệu tổng quan về hệ thống quản lý chất lượng, nguyên tắc, kỹ thuật ... để đảm bảo rằng dự án phần mềm sẽ chuyển giao cho khách hàng đúng hạn, đúng yêu cầu.

Đây là giáo trình sơ khởi, còn nhiều vấn đề chưa đi sâu phân tích và thực hiện, còn mang tính lý thuyết nhiều. Tác giả hy vọng bạn đọc đóng góp ý kiến để phiên bản 2 đáp ứng tốt hơn yêu cầu của nhiều độc giả, của sinh viên và kể cả những người đang công tác tại các phòng phát triển và đảm bảo chất lượng phần mềm.

## CHƯƠNG 1: CÁC KHÁI NIỆM

### 1.1. Các định nghĩa

“Lỗi phần mềm là chuyện hiển nhiên của cuộc sống. Chúng ta dù cố gắng đến mức nào thì thực tế là ngay cả những lập trình viên xuất sắc nhất cũng không có thể lúc nào cũng viết được những đoạn mã không có lỗi. Tính trung bình, ngay cả một lập trình viên loại tốt thì cũng có từ 1 đến 3 lỗi trên 100 dòng lệnh. Người ta ước lượng rằng việc kiểm tra để tìm ra các lỗi này chiếm phân nửa khối lượng công việc phải làm để có được một phần mềm hoạt động được”. (*Software Testing Techniques, Second Edition, by Boris Beizer, Van Nostrand Reinhold, 1990, ISBN 1850328803*).

**Trên đây là một nhận định về công việc kiểm nghiệm (testing) chương trình.**

Thật vậy, ngày nay càng ngày các chương trình (các phần mềm) càng trở lên phức tạp và đồ sộ. Việc tạo ra một sản phẩm có thể bán được trên thị trường đòi hỏi sự nỗ lực của hàng chục, hàng trăm thậm chí hàng ngàn nhân viên. Số lượng dòng mã lên đến hàng triệu. Và để tạo ra một sản phẩm thì không phải chỉ do một tổ chức đứng ra làm từ đầu đến cuối, mà đòi hỏi sự liên kết, tích hợp của rất nhiều sản phẩm, thư viện lập trình, ... của nhiều tổ chức khác nhau... Từ đó đòi hỏi việc kiểm nghiệm phần mềm càng ngày càng trở nên rất quan trọng và rất phức tạp.

Song song với sự phát triển các công nghệ lập trình, các ngôn ngữ lập trình... thì các công nghệ và kỹ thuật kiểm nghiệm phần mềm ngày càng phát triển và mang tính khoa học. Bài tiểu luận này với mục đích là tập hợp, nghiên cứu, phân tích các kỹ thuật, các công nghệ kiểm nghiệm phần mềm đang được sử dụng và phát triển hiện nay.

#### 1.1.1. Định nghĩa:

Việc kiểm nghiệm là quá trình thực thi một chương trình với mục đích là tìm ra lỗi. (Glen Myers)

*Giải thích theo mục đích:*

Việc thử nghiệm hiển nhiên là nói đến các lỗi (error), sai sót (fault), hỏng hóc (failure) hoặc các hậu quả (incident). Một phép thử là một cách chạy phần mềm theo các trường hợp thử nghiệm với mục tiêu là:

- Tìm ra sai sót.
- Giải thích sự hoạt động chính xác.

(Paul Jorgensen)

#### 1.1.2. Các thuật ngữ:

- Lỗi (Error):
  - Là các lỗi lầm do con người gây ra.
- Sai sót (Fault):
  - Sai sót gây ra lỗi. Có thể phân loại như sau:

- Sai sót do đưa ra dư thừa – chúng ta đưa một vài thứ không chính xác vào mô tả yêu cầu phần mềm.
- Sai sót do bỏ sót – Người thiết kế có thể gây ra sai sót do bỏ sót, kết quả là thiếu một số phần đáng ra phải có trong mô tả yêu cầu phần mềm.
- **Hỏng hóc (Failure):**
  - Xảy ra khi sai sót được thực thi. (Khi thực thi chương trình tại các nơi bị sai thì sẽ xảy ra trạng thái hỏng hóc).
- **Kết quả không mong đợi, hậu quả (Incident)**
  - Là những kết quả do sai sót đem đến. Hậu quả là các triệu chứng liên kết với một hỏng hóc và báo hiệu cho người dùng biết sự xuất hiện của hỏng hóc.
- **Trường hợp thử (Test case)**
  - Trường hợp thử được liên kết tương ứng với hoạt động của chương trình. Một trường hợp thử bao gồm một tập các giá trị đầu vào và một danh sách các kết quả đầu ra mong muốn.
- **Thẩm tra (Verification)**
  - Thẩm tra là tiến trình nhằm xác định đầu ra của một công đoạn trong việc phát triển phần mềm phù hợp với công đoạn trước đó.
- **Xác nhận (Validation)**
  - Xác nhận là tiến trình nhằm chỉ ra toàn hệ thống đã phát triển xong phù hợp với tài liệu mô tả yêu cầu.

***So sánh giữa Thẩm tra và Xác nhận:***

- **Thẩm tra:** thẩm tra quan tâm đến việc ngăn chặn lỗi giữa các công đoạn.
- **Xác nhận:** xác nhận quan tâm đến sản phẩm cuối cùng không còn lỗi.

**1.2. Vòng đời của việc kiểm nghiệm (testing life cycle):**

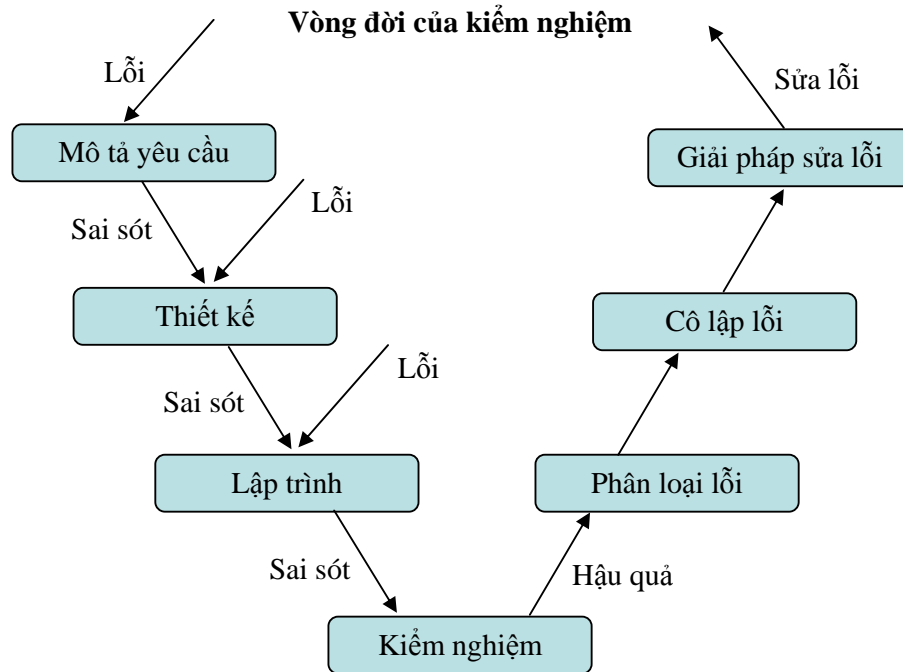
Bảng dưới đây mô tả các công đoạn phát triển một phần mềm và cách khắc phục lỗi.

Lỗi có thể xảy ra trong tất cả các công đoạn từ “Mô tả yêu cầu”, “Thiết kế” đến “Lập trình”.

Từ công đoạn này chuyển sang công đoạn khác thường nảy sinh các sai sót (do dư thừa hoặc thiếu theo mô tả yêu cầu).

Đến công đoạn kiểm nghiệm chúng ta sẽ phát hiện ra các hậu quả (các kết quả không mong muốn).

Quá trình sửa lỗi bao gồm “phân loại lỗi”, “cô lập lỗi” (tìm ra nguyên nhân và nơi gây lỗi), đề ra “giải pháp sửa lỗi” và cuối cùng là khắc phục lỗi.

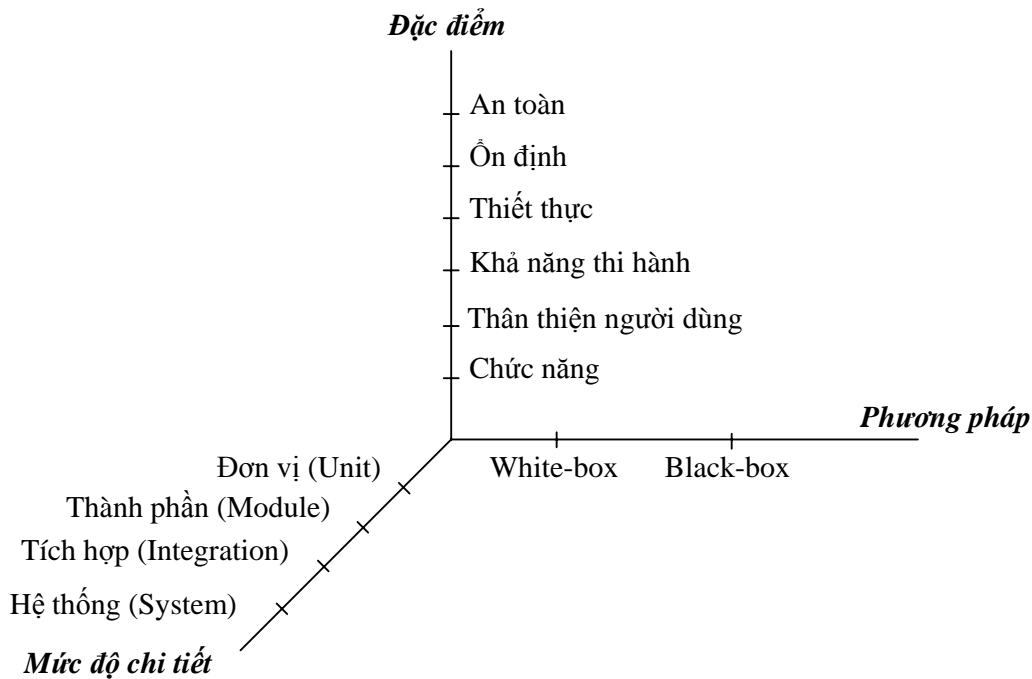


### 1.3. Phân loại kiểm nghiệm:

Có 2 mức phân loại:

- Một là phân biệt theo mức độ chi tiết của các bộ phận hợp thành phần mềm.
  - Mức kiểm tra đơn vị (Unit)
  - Mức kiểm tra hệ thống (System)
  - Mức kiểm tra tích hợp (Integration)
- Cách phân loại khác là dựa trên phương pháp thử nghiệm (thường dùng ở mức kiểm tra đơn vị)
  - Kiểm nghiệm hộp đen (Black box testing) dùng để kiểm tra chức năng.
  - Kiểm nghiệm hộp trắng (White box testing) dùng để kiểm tra cấu trúc.

Hình bên dưới biểu diễn sự tương quan của “các tiêu chí chất lượng phần mềm”, “mức độ chi tiết đơn vị” và “phương pháp kiểm nghiệm”



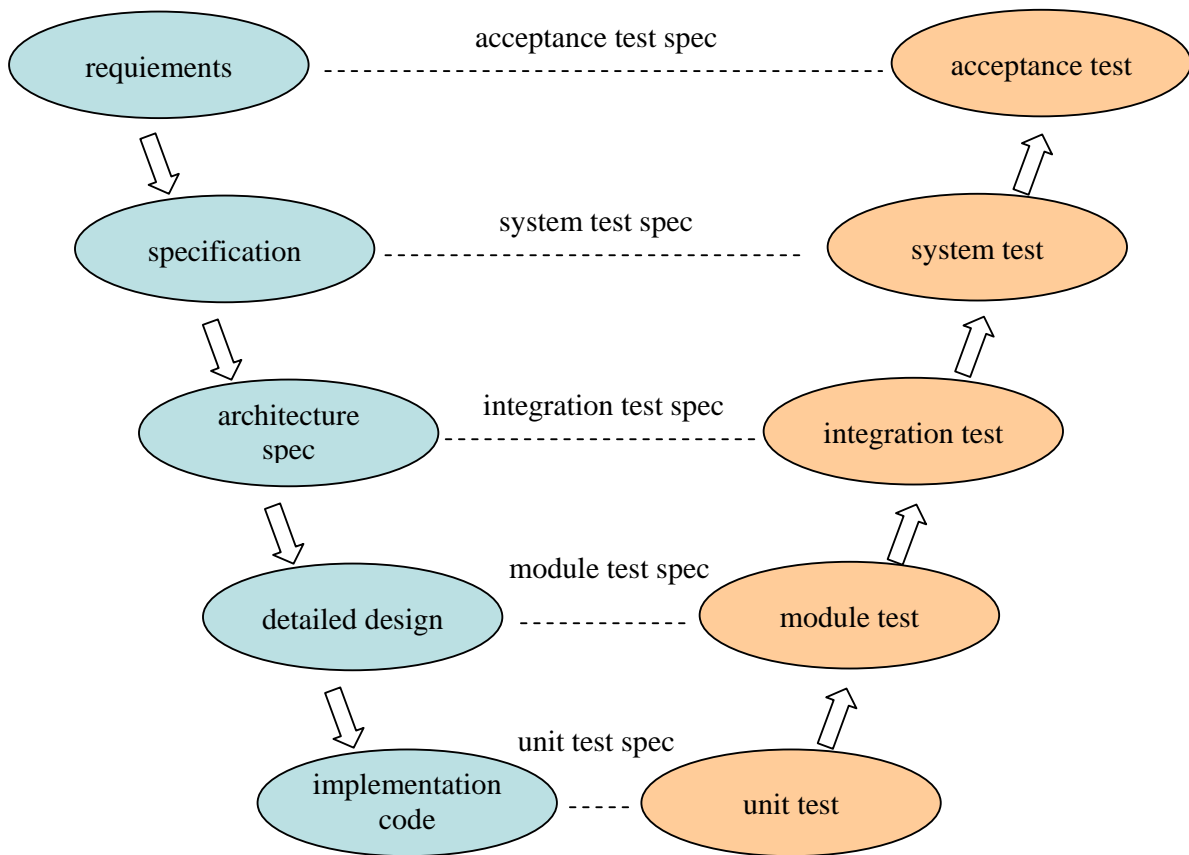
#### 1.4. Sự tương quan giữa các công đoạn xây dựng phần mềm và loại kiểm nghiệm: Mô hình chữ V

Mô hình này nhằm giải thích sự tương quan giữa các công đoạn xây dựng phần mềm và các loại kiểm nghiệm. Ở mỗi công đoạn xây dựng phần mềm sẽ tương ứng với một loại kiểm nghiệm và cần có một hồ sơ kiểm nghiệm tương ứng được thành lập để phục vụ cho việc kiểm nghiệm.

Ví dụ:

- **Công đoạn:** Yêu cầu phần mềm(*requirements*); **Loại kiểm nghiệm:** Kiểm nghiệm chấp nhận (*acceptance test*); **Hồ sơ:** hồ sơ kiểm nghiệm chấp nhận (*acceptance test spec*).
- **Công đoạn:** Mô tả chi tiết phần mềm (*specification*); **Loại kiểm nghiệm:** Kiểm nghiệm hệ thống(*system test*); **Hồ sơ:** hồ sơ kiểm nghiệm hệ thống (*system test spec*).
- **Công đoạn:** Hồ sơ kiến trúc (*architecture spec*); **Loại kiểm nghiệm:** Kiểm nghiệm tích hợp (*integration test*); **Hồ sơ:** hồ sơ kiểm nghiệm tích hợp (*integration test spec*).
- **Công đoạn:** Thiết kế chi tiết (*detailed design*); **Loại kiểm nghiệm:** Kiểm nghiệm khối (*module test*); **Hồ sơ:** hồ sơ kiểm nghiệm khối (*module test spec*).
- **Công đoạn:** Viết mã (*implementation code*); **Loại kiểm nghiệm:** Kiểm nghiệm đơn vị (*unit test*); **Hồ sơ:** hồ sơ kiểm nghiệm đơn vị (*unit test spec*).





### 1.5. Sơ lược các kỹ thuật và công đoạn kiểm nghiệm:

Các kỹ thuật và công đoạn kiểm nghiệm có thể chia như sau:

- Kiểm nghiệm tầm hẹp: kiểm nghiệm các bộ phận riêng rẽ.
  - Kiểm nghiệm hộp trắng (White box testing)
  - Kiểm nghiệm hộp đen (Black box testing)
- Kiểm nghiệm tầm rộng:
  - Kiểm nghiệm bộ phận (Module testing): kiểm nghiệm một bộ phận riêng rẽ.
  - Kiểm nghiệm tích hợp (Integration testing): tích hợp các bộ phận và hệ thống con.
  - Kiểm nghiệm hệ thống (System testing): kiểm nghiệm toàn bộ hệ thống.
  - Kiểm nghiệm chấp nhận (Acceptance testing): thực hiện bởi khách hàng.

### 1.5.1. Các loại kiểm nghiệm tầm hẹp:

Các loại kiểm nghiệm này được thực hiện để kiểm nghiệm đến các đơn vị (unit) hoặc các khối chức năng (module).

#### **a. Kiểm nghiệm hộp trắng (white-box testing)**

Còn gọi là kiểm nghiệm cấu trúc. Kiểm nghiệm theo cách này là loại kiểm nghiệm sử dụng các thông tin về cấu trúc bên trong của ứng dụng. Việc kiểm nghiệm này dựa trên quá trình thực hiện xây dựng phần mềm.

Tiêu chuẩn của kiểm nghiệm hộp trắng phải đáp ứng các yêu cầu như sau:

- *Bao phủ dòng lệnh*: mỗi dòng lệnh ít nhất phải được thực thi 1 lần
- *Bao phủ nhánh*: mỗi nhánh trong sơ đồ điều khiển (control graph) phải được đi qua một lần.
- *Bao phủ đường*: tất cả các đường (path) từ điểm khởi tạo đến điểm cuối cùng trong sơ đồ dòng điều khiển phải được đi qua.

#### **b. Kiểm nghiệm hộp đen (black-box testing)**

Còn gọi là kiểm nghiệm chức năng. Việc kiểm nghiệm này được thực hiện mà không cần quan tâm đến các thiết kế và viết mã của chương trình. Kiểm nghiệm theo cách này chỉ quan tâm đến chức năng đã đề ra của chương trình. Vì vậy kiểm nghiệm loại này chỉ dựa vào bản mô tả chức năng của chương trình, xem chương trình có thực sự cung cấp đúng chức năng đã mô tả trong bản chức năng hay không mà thôi.

Kiểm nghiệm hộp đen dựa vào các định nghĩa về chức năng của chương trình. Các trường hợp thử nghiệm (test case) sẽ được tạo ra dựa nhiều vào bản mô tả chức năng chứ không phải dựa vào cấu trúc của chương trình.

#### **c. Vấn đề kiểm nghiệm tại biên:**

Kiểm nghiệm biên (boundary) là vấn đề được đặt ra trong cả hai loại kiểm nghiệm hộp đen và hộp trắng. Lý do là do lỗi thường xảy ra tại vùng này.

Ví dụ:

*if  $x > y$  then S1 else S2*

Với điều kiện bao phủ, chỉ cần 2 trường hợp thử là  $x > y$  và  $x \leq y$ .

Với kiểm nghiệm đường biên thì kiểm tra với các trường hợp thử là  $x > y$ ,  $x < y$ ,  $x = y$

#### **Các loại kiểm nghiệm tầm rộng:**

Việc kiểm nghiệm này thực hiện trên tầm mức lớn hơn và các khía cạnh khác của phần mềm như kiểm nghiệm hệ thống, kiểm nghiệm sự chấp nhận (của người dùng)...

### **a. Kiểm nghiệm Module (Module testing)**

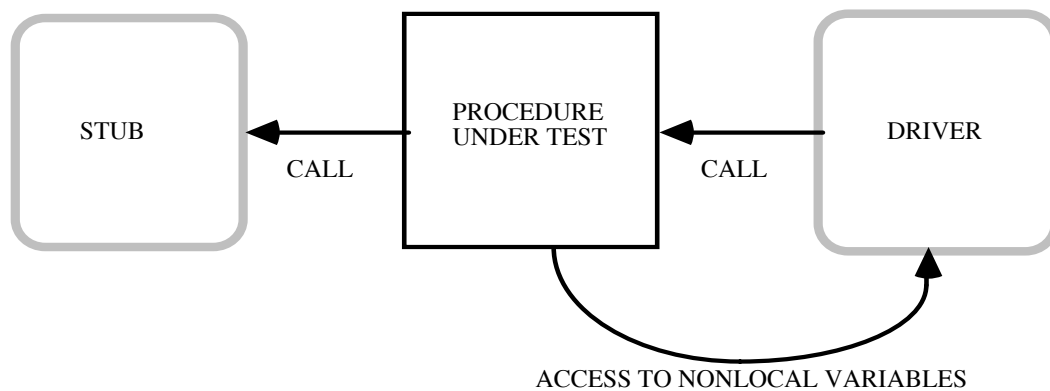
Mục đích: xác minh module đưa ra đã được xây dựng đúng hay chưa?

Vấn đề đặt ra: giả sử module I sử dụng các module H, K. Nhưng các module H và K chưa sẵn sàng. Vậy cách nào để kiểm tra module I một cách độc lập?

Giải pháp đề ra là giả lập môi trường của module H và K.

Thông thường một module có thể gọi một tác vụ (hay một tiến trình) không phải của nó, truy cập các cấu trúc dữ liệu không phải là cục bộ, hay được dùng bởi một module khác.

Hình sau mô tả module được đặt trong môi trường thử nghiệm.



Ghi chú: Driver là module gọi thực thi làm cho module cần kiểm tra hoạt động, nó giả lập các module khác sẽ sử dụng module này. Các tập dữ liệu chia sẻ mà các module khác thiết lập trong thực tế cũng được thiết lập ở driver. Stub là module giả lập các module được module đang kiểm tra sử dụng.

### **b. Kiểm nghiệm tích hợp:**

Là cách kiểm nghiệm bằng cách tích hợp vào hệ thống từng module một và kiểm tra.

Ưu điểm:

- Dễ dàng tìm ra các lỗi vào ngay giai đoạn đầu.
- Dễ dàng khoanh vùng các lỗi (tích hợp n modules, sau đó n + 1 modules).
- Giảm việc sử dụng các stub và Driver

Có thể thực hiện kiểm nghiệm tích hợp theo cả 2 cách bottom-up và top-down tùy thuộc vào mối quan hệ sử dụng lẫn nhau giữa các module.

### **c. Kiểm nghiệm hệ thống:**

Bao gồm một loạt các kiểm nghiệm nhằm xác minh toàn bộ các thành phần của hệ thống được tích hợp một cách đúng đắn.

Mục đích của kiểm nghiệm hệ thống là để đảm bảo toàn bộ hệ thống hoạt động như ý mà khách hàng mong muốn.

Bao gồm các loại kiểm nghiệm sau:

- **Kiểm nghiệm chức năng (Function testing)**

Kiểm tra hệ thống sau khi tích hợp có hoạt động đúng chức năng với yêu cầu đặt ra trong bản mô tả yêu cầu hay không.

Ví dụ: với hệ thống xử lý văn bản thì kiểm tra các chức năng *tạo tài liệu, sửa tài liệu, xóa tài liệu...* có hoạt động hay không.

- **Kiểm nghiệm hiệu suất (Performance testing)**

- **Kiểm nghiệm mức độ đáp ứng (stress testing)**

Thực thi hệ thống với giả thiết là các tài nguyên hệ thống yêu cầu không đáp ứng được về chất lượng, ổn định và số lượng.

- **Kiểm nghiệm cấu hình (configuration testing)**

Phân tích hệ thống với các thiết lập cấu hình khác nhau.

- **Kiểm nghiệm ổn định (robustness testing)**

Kiểm nghiệm dưới các điều kiện không mong đợi ví dụ như người dùng gõ lệnh sai, nguồn điện bị ngắt.

- **Kiểm nghiệm hồi phục (recovery testing)**

Chỉ ra các kết quả trả về khi xảy ra lỗi, mất dữ liệu, thiết bị, dịch vụ... hoặc xóa các dữ liệu hệ thống và xem khả năng phục hồi của nó.

- **Kiểm nghiệm quá tải (overload testing)**

Đánh giá hệ thống khi nó vượt qua giới hạn cho phép. Ví dụ: một hệ thống giao tác (transaction) được yêu cầu thực thi 20 giao tác/giây. Khi đó sẽ kiểm tra nếu 30 giao tác/giây thì như thế nào?

- **Kiểm nghiệm chất lượng (quality testing)**

Đánh giá sự tin tưởng, vấn đề duy tu, tính sẵn sàng của hệ thống. Bao gồm cả việc tính toán thời gian trung bình hệ thống sẽ bị hỏng và thời gian trung bình để khắc phục.

- **Kiểm nghiệm cài đặt (Installation testing)**

Người dùng sử dụng các chức năng của hệ thống và ghi lại các lỗi tại vị trí sử dụng thật sự.

Ví dụ: một hệ thống được thiết kế để làm việc trên tàu thủy phải đảm bảo không bị ảnh hưởng gì bởi điều kiện thời tiết khác nhau hoặc do sự di chuyển của tàu.

#### ***d. Kiểm nghiệm chấp nhận***

Nhằm đảm bảo việc người dùng có được hệ thống mà họ yêu cầu. Việc kiểm nghiệm này hoàn thành bởi người dùng phụ thuộc vào các hiểu biết của họ vào các yêu cầu.

## CHƯƠNG 2: KIỂM CHỨNG VÀ XÁC NHẬN (V & V)

- 2.1. Lập kế hoạch V&V (Verification and validation planning)
- 2.2. Kiểm tra phần mềm (Software inspections)
- 2.3. Phân tích tĩnh tự động (Automated static analysis)
- 2.4. Phát triển phần mềm phòng sạch (Cleanroom software development)

### 2.1. Kiểm chứng và hợp lệ hoá

Kiểm thử phần mềm là một yếu tố trong chủ điểm rộng hơn thường được tham khảo tới như vấn đề kiểm chứng và hợp lệ hoá (V&V). Kiểm chứng nói tới một tập các hành động đảm bảo rằng phần mềm cài đặt đúng cho một chức năng đặc biệt. Hợp lệ hoá nói tới một tập các hoạt động khác đảm bảo rằng phần mềm đã được xây dựng lại theo yêu cầu của khách hàng. Boehm phát biểu điều này theo cách khác:

**Kiểm chứng:** “Chúng ta có làm ra sản phẩm đúng không? ”

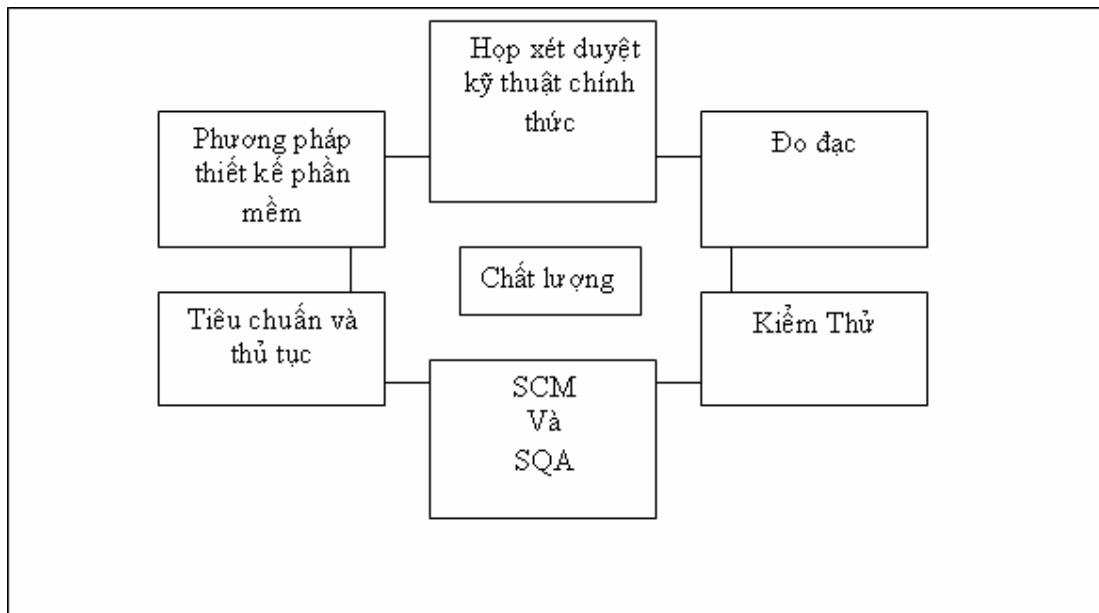
**Hợp lệ hoá:** “Chúng ta có làm ra đúng sản phẩm không? ”

Định nghĩa về V&V bao quát nhiều hoạt động ta đã tham khảo tới như việc đảm bảo chất lượng phần mềm (SQA).

Các phương pháp kỹ nghệ phần mềm cung cấp nền tảng để xây dựng nên chất lượng. Các phương pháp phân tích, thiết kế và thực hiện (mã hoá) làm nâng cao chất lượng bằng cách đưa ra những kỹ thuật thống nhất và kết quả dự kiến được. Các cuộc họp xét duyệt kỹ thuật chính thức (thảo trình) giúp đảm bảo chất lượng của sản phẩm được tạo ra như hệ quả của từng bước kỹ nghệ phần mềm. Qua toàn bộ tiến trình này, việc đo đạc và kiểm soát được áp dụng cho mọi phần tử của cấu hình phần mềm. Các chuẩn và thủ tục giúp đảm bảo tính thống nhất và tiến trình SQA chính thức buộc phải thi hành “ triết lý chất lượng toàn bộ ”.

Việc kiểm thử cung cấp một thành lũy cuối cùng để có thể thẩm định về chất lượng, lỗi có thể được phát hiện ra một cách thực tế hơn.

Nhưng không nên coi kiểm thử như một tấm lưới an toàn. Như người ta vẫn nói, “ Bạn không thể kiểm thử được chất lượng. Nếu nó không sẵn có trước khi bạn bắt đầu kiểm thử thì nó sẽ chẳng có khi bạn kết thúc kiểm thử.” Chất lượng được tổ hợp vào trong phần mềm trong toàn bộ tiến trình kỹ nghệ phần mềm. Việc áp dụng đúng các phương pháp và công cụ, các cuộc họp xét duyệt kỹ thuật chính thức và việc quản lý vững chắc cùng cách đo đạc tất cả dẫn tới chất lượng được xác nhận trong khi kiểm thử.



Hình 2.1 Đạt đến chất lượng phần mềm

Miller kể lại việc kiểm thử phần mềm về đảm bảo chất lượng bằng cách nói rằng “động cơ nền tảng của việc kiểm thử chương trình là để xác nhận chất lượng phần mềm bằng những phương pháp có thể được áp dụng một cách kinh tế và hiệu quả cho cả các hệ thống quy mô lớn và nhỏ.”

Điều quan trọng cần lưu ý rằng việc kiểm chứng và hợp lệ hoá bao gồm một phạm vi rộng các hoạt động SQA có chứa cả hợp xét duyệt chính thức, kiểm toán chất lượng và cấu hình, điều phối hiệu năng, mô phỏng, nghiên cứu khả thi, xét duyệt tài liệu, xét duyệt cơ sở dữ liệu, phân tích thuật toán, kiểm thử phát triển, kiểm thử chất lượng, kiểm thử cài đặt. Mặc dầu việc kiểm thử đóng một vai trò cực kỳ quan trọng trong V&V, nhiều hoạt động khác cũng còn cần tới.

### 2.1.1. Tổ chức việc kiểm thử phần mềm

Với mọi dự án phần mềm, có một mâu thuẫn cố hữu về lợi ích xuất hiện ngay khi việc kiểm thử bắt đầu. Người đã xây phần mềm bây giờ được yêu cầu kiểm thử phần mềm. Điều này bản thân nó dường như vô hại; sau rốt, ai biết được chương trình kỹ hơn là người làm ra nó? Nhưng không may, cũng những người phát triển này lại có mối quan tâm chứng minh rằng chương trình là không có lỗi, rằng nó làm việc đúng theo yêu cầu khách hàng, rằng nó sẽ được hoàn tất theo lịch biểu và trong phạm vi ngân sách. Một trong những mối quan tâm này lại làm giảm bớt việc tìm ra lỗi trong toàn bộ tiến trình kiểm thử.

Theo quan điểm tâm lý, việc phân tích và thiết kế phần mềm (cùng với mã hoá) là nhiệm vụ *xây dựng*. Người kỹ sư phần mềm tạo ra một chương trình máy tính, tài liệu về nó và các cấu trúc dữ liệu có liên quan. Giống như bất kỳ người xây dựng nào, người kỹ sư phần mềm tự hào về dinh thự đã được xây dựng và nhìn ngò vức vào bất

kỳ ai định làm sập đổ nó. Khi việc kiểm thử bắt đầu, có một nỗ lực tinh vi, dứt khoát để “đập vỡ” cái người kỹ sư phần mềm đã xây dựng. Theo quan điểm của người xây dựng, việc kiểm thử có thể được coi như (về tâm lý) có *tính phá hoại*. Cho nên người xây dựng dè dặt đề cập tới việc kiểm thử thiết kế và thực hiện sẽ chứng tỏ rằng chương trình làm việc, thay vì phát hiện lỗi. Điều không may lỗi sẽ hiện hữu. Và nếu người kỹ sư phần mềm không tìm ra chúng thì khách hàng sẽ tìm ra.

Thường có một số nhận thức sai có thể được suy diễn sai lạc từ thảo luận trên: (1) người phát triển phần mềm không nên tiến hành kiểm thử; (2) phần mềm nên được “tung qua tường” cho người lạ làm việc kiểm thử một cách tàn bạo; (3) người kiểm thử nên tham gia vào dự án chỉ khi bước kiểm thử sắp sửa bắt đầu. Từng phát biểu này đều không đúng.

Người phát biểu phần mềm bao giờ cũng có trách nhiệm với việc kiểm thử riêng các đơn vị (mô đun) chương trình, để đảm bảo rằng mỗi mô đun thực hiện đúng chức năng nó đã được thiết kế. Trong nhiều trường hợp, người phát triển cũng tiến hành cả kiểm thử tích hợp - bước kiểm thử dẫn đến việc xây dựng (và kiểm thử) toàn bộ cấu trúc chương trình. Chỉ sau khi kiến trúc phần mềm hoàn tất thì nhóm kiểm thử độc lập mới tham gia vào.

Vai trò của nhóm kiểm thử độc lập (ITG) là loại bỏ vấn đề cố hữu liên quan tới việc để người xây dựng kiểm thử những cái anh ta đã xây dựng ra. Việc kiểm thử độc lập loại bỏ xung khắc lợi ích nếu không có nhóm đó thì có thể hiện hữu. Cuối cùng nhân sự trong nhóm kiểm thử độc lập được trả tiền để tìm ra lỗi.

Tuy nhiên, người phát triển phần mềm không chuyển giao chương trình cho ITG rồi bỏ đi. Người phát triển và ITE làm việc chặt chẽ trong toàn bộ dự án phần mềm để đảm bảo rằng những kiểm thử kỹ lưỡng sẽ được tiến hành. Trong khi tiến hành kiểm thử, người phát triển phải có sẵn để sửa chữa lỗi đã phát hiện ra.

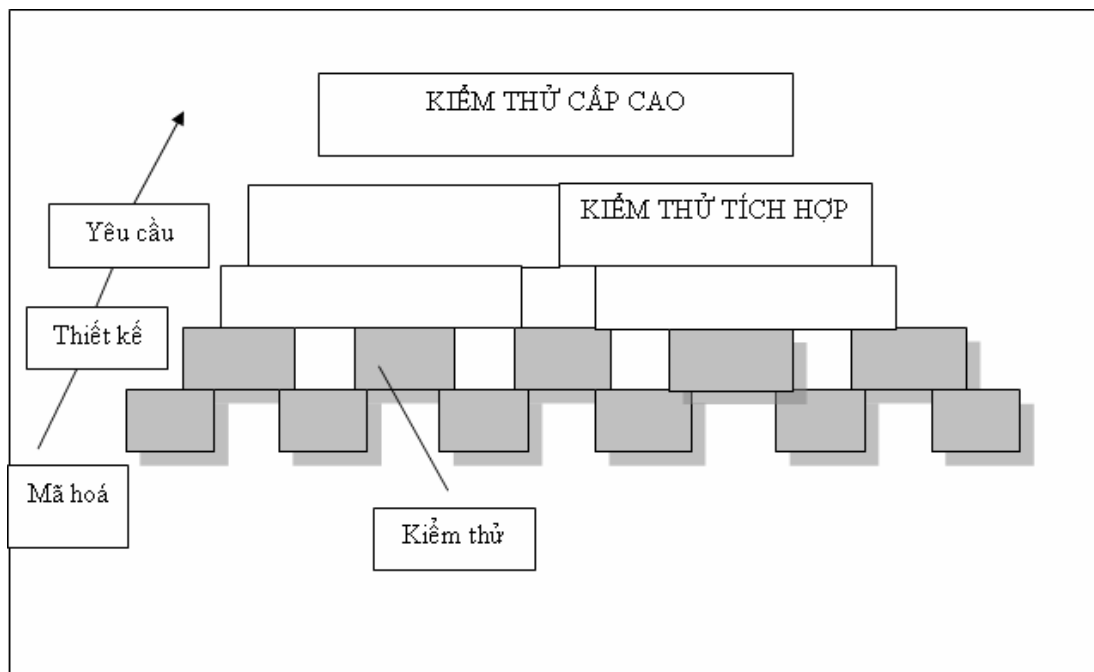
ITG là một phần của nhóm dự án phát triển phần mềm theo nghĩa là nó tham dự trong tiến trình đặc tả và vẫn còn tham dự (lập kế hoạch và xác định các thủ tục kiểm thử) trong toàn bộ dự án lớn. Tuy nhiên, trong nhiều trường hợp ITG báo cáo cho tổ chức đảm bảo chất lượng phần mềm, do đó đạt tới một mức độ độc lập có thể không có được nếu nó là một phần của tổ chức phát triển phần mềm.

### 2.1.2. Chiến lược kiểm thử phần mềm

Tiến trình kỹ nghệ phần mềm có thể được xét theo vòng xoắn ốc, như được minh họa trong Hình 2.2. Ban đầu, kỹ nghệ phần mềm xác định vai trò của phần mềm và đưa tới việc phân tích yêu cầu phần mềm, chỗ thiết lập nên lĩnh vực thông tin, chức năng, hành vi, hiệu năng, ràng buộc và tiêu chuẩn hợp lệ cho phần mềm. Đi vào trong vòng xoắn ốc, chúng ta tới thiết kế và cuối cùng tới mã hoá. Để xây dựng phần mềm máy tính, chúng ta đi dọc theo đường xoắn ốc, mỗi lần mức độ trừu tượng lại giảm dần.

Một chiến lược cho kiểm thử phần mềm cũng có thể xem xét bằng cách đi theo đường xoắn ốc của Hình 2.2 ra ngoài. Việc kiểm thử đơn vị bắt đầu tại tâm xoáy của xoắn ốc và tập chung vào các đơn vị của phần mềm khi được cài đặt trong chương trình gốc. Việc kiểm thử tiến triển bằng cách đi ra theo đường xoắn ốc tới kiểm thử tích hợp, nơi tập trung vào thiết kế và việc xây dựng kiến trúc phần mềm. Đi thêm một vòng xoáy nữa trên đường xoắn ốc chúng ta gặp kiểm thử hợp lệ, nơi các yêu cầu, được

thiết lập như một phần của việc phân tích yêu cầu phần mềm, được hợp lệ hoá theo phần mềm đã được xây dựng. Cuối cùng chúng ta tới kiểm thử hệ thống, nơi phần mềm và các phần tử hệ thống khác được kiểm thử như một toàn bộ. Để kiểm thử phần mềm máy tính, chúng ta theo đường xoáy mở rộng dần phạm vi kiểm thử một lần.



Hình 2.3 Các bước kiểm thử phần mềm

Xem xét tiến trình này theo quan điểm thủ tục vì việc kiểm thử bên trong hoàn cảnh kỹ nghệ phần mềm thực tại là một chuỗi gồm ba bước được thực hiện tuần tự nhau. Các bước này được vẽ trong Hình 2.3. Ban đầu, việc kiểm thử tập trung vào từng mô đun riêng biệt, đảm bảo rằng nó vận hành đúng đắn như một đơn vị. Do đó mới có tên kiểm thử đơn vị dùng rất nhiều các kỹ thuật kiểm thử hộp trắng, thử các đường đặc biệt trong cấu trúc điều khiển của một mô đun để đảm bảo bao quát đầy đủ và phát hiện ra lỗi tối đa. Tiếp đó các mô đun phải được lắp ghép hay tích hợp lại để tạo nên bộ trình phần mềm hoàn chỉnh. Việc kiểm thử tích hợp đề cập tới các vấn đề có liên quan tới các vấn đề kiểm chứng và xây dựng chương trình. Các kỹ thuật thiết kế kiểm thử hộp đen chiếm đại đa số trong việc tích hợp, mặc dầu một số giới hạn các kiểm thử hộp trắng cũng có thể được dùng để đảm bảo bao quát đa số các đường điều khiển.

Sau khi phần mềm đã được tích hợp (được xây dựng), một tập các phép kiểm thử cao cấp sẽ được tiến hành. Các tiêu chuẩn hợp lệ (được thiết lập trong phân tích yêu cầu) cũng phải được kiểm thử. Việc kiểm thử hợp lệ đưa ra sự đảm bảo cuối cùng rằng phần mềm đáp ứng cho tất cả các yêu cầu chức năng, hành vi và sự hoàn thiện. Các kỹ thuật kiểm thử hộp đen được dùng chủ yếu trong việc hợp lệ hoá này.

Bước kiểm thử cấp cao cuối cùng rơi ra ngoài phạm vi của kỹ nghệ phần mềm và rơi vào hoàn cảnh rộng hơn của kỹ nghệ hệ thống máy tính. Phần mềm, một khi được hợp lệ hoá, phải được tổ hợp với các phần tử hệ thống khác (như phần cứng, con người, cơ sở dữ liệu). Kiểm thử hệ thống kiểm chứng lại rằng tất cả các yếu tố có



khớp đúng với nhau không và rằng chức năng/ độ hoàn thiện hệ thống toàn bộ đã đạt được.

### 2.1.3. Tiêu chuẩn hoàn thành kiểm thử

Câu hỏi cổ điển nảy sinh mỗi khi có việc thảo luận về kiểm thử phần mềm là: Khi nào chúng ta thực hiện xong kiểm thử - làm sao ta biết rằng chúng ta đã kiểm thử đủ? Đáng buồn là không có câu trả lời xác định cho câu hỏi này, nhưng có một vài sự đáp ứng thực tế và những nỗ lực ban đầu theo hướng dẫn kinh nghiệm.

Một đáp ứng cho câu hỏi trên là: Bạn chẳng bao giờ hoàn thành việc kiểm thử, gánh nặng đơn giản chuyển từ bạn (người phát triển) sang khách hàng của bạn. Mỗi lúc khách hàng / người dùng thực hiện một chương trình máy tính thì chương trình này lại được kiểm thử trên một tập dữ liệu mới. Sự kiện đúng mức này nhấn mạnh tầm quan trọng của các hoạt động đảm bảo chất lượng phần mềm khác. Một đáp ứng khác (có điều gì đó nhạo báng nhưng dẫu sao cũng chính xác) là : Bạn hoàn thành việc kiểm thử khi bạn hết thời gian hay hết tiền.

Mặc dầu số ít người thực hành sẽ biện minh cho những đáp ứng trên, người kỹ sư phần mềm cần những tiêu chuẩn chặt chẽ hơn để xác định khi nào việc kiểm thử đủ được tiến hành. Musa và Ackerman gợi ý một đáp ứng dựa trên tiêu chuẩn thống kê: “ Không, chúng ta không thể tuyệt đối chắc chắn rằng phần mềm sẽ không bao giờ hỏng, nhưng theo mô hình thống kê đúng về lý thuyết và hợp lệ về thực nghiệm thì chúng ta đã hoàn thành kiểm thử đủ để nói với sự tin tưởng tới 95% rằng xác suất của 1000 giờ vận hành CPU không hỏng trong một môi trường được xác định về xác suất là ít nhất 0.995”

Dùng mô hình hoá thống kê và lý thuyết độ tin cậy phần mềm, các mô hình về hỏng hóc phần mềm (được phát hiện trong khi kiểm thử) xem như một hàm của thời gian thực hiện có thể được xây dựng ra. Một bản của mô hình sai hỏng, được gọi là *mô hình thực hiện- thời gian Poisson lô ga rit*, có dạng:

$$f(t) = \left(\frac{1}{p}\right)x \ln[ l_0(pt + 1) ] \quad (17.1)$$

với  $f(t)$  = số tích lũy những hỏng hóc dự kiến xuất hiện một khi phần mềm đã được kiểm thử trong một khoảng thời gian thực hiện  $t$ .

$l_0$  = mật độ hỏng phần mềm ban đầu (số hỏng trên đơn vị thời gian) vào lúc bắt đầu kiểm thử.

$P$  = việc giảm theo hàm mũ trong mật độ hỏng khi lỗi được phát hiện và sửa đổi được tiến hành.

Mật độ hỏng thể nghiệm.  $l(t)$ , có thể được suy ra bằng cách lấy đạo hàm của  $f(t)$ :

$$F(t) = \frac{l_0}{l_0 pt + 1} \quad (17.2)$$

Dùng mối quan hệ trong phương trình (2.2), người kiểm thử có thể tiên đoán việc loại bỏ lỗi khi việc kiểm thử tiến triển. Mật độ lỗi thực tại có thể được chấm lên trên đường cong dự kiến (hình 2.4). Nếu dữ liệu thực tại được thu thập trong khi

kiểm thử và mô hình thực hiện - thời gian theo logarit Poisson là xấp xỉ gần nhau với số điểm dữ liệu thì mô hình này có thể được dùng để dự đoán thời gian kiểm thử toàn bộ cần để đạt tới mật độ hỏng thấp chấp nhận được.

Bằng cách thu thập các độ đo trong khi kiểm thử phần mềm và dùng các mô hình về độ tin cậy phần mềm hiện có, có thể phát triển những hướng dẫn có nghĩa để trả lời câu hỏi: Khi nào thì chúng ta hoàn thành việc kiểm thử? Còn ít tranh luận về việc có phải làm công việc thêm nữa hay không trước khi các quy tắc định tính cho kiểm thử có thể xác định, nhưng cách tiếp cận kinh nghiệm hiện đang tồn tại được coi là tốt hơn đáng kể so với trực giác thô.

## 2.2. Phát triển phần mềm phòng sạch (cleanroom software development)

Cleanroom là một qui trình phát triển phần mềm hơn là một kỹ thuật kiểm thử. Cho đến bây giờ, kỹ thuật này vẫn được xem là một cách mới của việc suy nghĩ về kiểm thử và đảm bảo chất lượng phần mềm. Ý tưởng của cleanroom là nhằm tránh tiêu tốn chi phí cho hoạt động phát hiện và gỡ bỏ các lỗi bằng cách viết mã lệnh chương trình một cách chính xác ngay từ ban đầu với những phương pháp chính thống như kỹ thuật chứng minh tính đúng đắn trước khi kiểm thử.

### 2.2.1. Nghệ thuật của việc gỡ rối

Kiểm thử phần mềm là một tiến trình có thể được vạch kế hoạch và xác định một cách hệ thống. Việc thiết kế trường hợp kiểm thử có thể tiến hành một chiến lược xác định và có kết quả được tính toán theo thời gian.

Gỡ lỗi xuất hiện như hậu quả của việc kiểm thử thành công. Tức là, khi một trường hợp kiểm thử phát hiện ra lỗi thì việc gỡ lỗi là tiến trình sẽ nảy sinh để loại bỏ lỗi. Mặc dầu việc gỡ lỗi có thể nên là một tiến trình có trật tự, nó phần lớn còn là nghệ thuật. Người kỹ sư phần mềm khi tính các kết quả của phép thử, thường hay phải đương đầu với chỉ dẫn “triệu chứng” và vấn đề phần mềm. Tức là, cái biểu lộ ra bên ngoài của lỗi và nguyên nhân bên trong của lỗi có thể có mối quan hệ không hiển nhiên tới một lỗi khác. Tiến trình tâm trí ít hiểu biết gắn một triệu chứng với nguyên nhân chính việc gỡ lỗi.

### 2.2.2. Tiến trình gỡ lỗi

Gỡ lỗi không phải là kiểm thử, nhưng bao giờ cũng xuất hiện như một hệ quả kiểm thử. Tham khảo đến hình 2.12 thì tiến trình gỡ lỗi bắt đầu với việc thực hiện kiểm thử. Kết quả được thẩm định và gặp việc thiếu sự tương ứng giữa kết quả trông đợi và thực tế. Trong nhiều trường hợp, dữ liệu không tương ứng là triệu chứng của một nguyên nhân nền tảng còn bị che kín. Tiến trình gỡ lỗi cố gắng ghép triệu chứng với nguyên nhân, từ đó dẫn tới việc sửa lỗi.

Tiến trình gỡ lỗi bao giờ cũng sinh ra một trong hai kết quả logic: (1) Nguyên nhân sẽ được tìm ra, sửa chữa và loại bỏ hay (2) nguyên nhân sẽ không được tìm ra. Trong trường hợp sau, người thực hiện gỡ lỗi có thể hoài nghi một nguyên nhân, thiết kế ra một trường hợp kiểm thử giúp hợp lệ hoá hoài nghi của mình, và việc làm hướng tới việc sửa lỗi theo cách lặp lại.

**Tại sao gỡ lỗi lại khó? Rất có thể tâm lý con người (xem mục sau) có liên quan tới nhiều câu trả lời hơn là công nghệ phần mềm. Tuy nhiên một vài đặc trưng của lỗi đưa ra vài manh mối:**

- Triệu chứng và nguyên nhân có thể xa nhau về mặt địa lý. Tức là, những triệu chứng có thể xuất hiện trong một phần này của chương trình, trong khi nguyên nhân thực tế có thể định vị ở một vị trí xa. Các cấu trúc chương trình đi đôi với nhau làm trầm trọng thêm tình huống này.
- Triệu chứng có thể biến mất (tạm thời) khi một lỗi khác được sửa chữa.
- Triệu chứng thực tế có thể gây ra không lỗi (như do sự không chính xác của việc làm tròn số).
- Triệu chứng có thể được gây ra do lỗi con người không để lại dấu vết.
- Triệu chứng có thể là kết quả của vấn đề thời gian, thay vì vấn đề xử lý.
- Có thể khó tái tạo lại chính xác các điều kiện vào (như ứng dụng thời gian thực trong đó thứ tự vào không xác định)
- Triệu chứng có thể có lúc có lúc không. Điều này đặc biệt phổ biến trong các hệ thống nhúng việc đi đôi phần cứng và phần mềm không chặt chẽ.
- Triệu chứng có thể do nguyên nhân được phân bố qua một số các nhiệm vụ chạy trên các bộ xử lý khác nhau.
- Trong khi gỡ lỗi, chúng ta gặp không ít các lỗi chạy từ việc hơi khó chịu (như định dạng cái ra không đúng) tới các thảm họa (như hệ thống hỏng, gây ra các thiệt hại kinh tế hay vật lý trầm trọng). Xem như hậu quả của việc tăng lỗi, khối lượng sức ép để tìm ra lỗi cũng tăng thêm. Thông thường, sức ép buộc người phát triển phần mềm phải tìm ra lỗi và đồng thời đưa vào thêm hai lỗi nữa.

### **2.2.3. Xem xét tâm lý**

Không may, dường như có một số bằng chứng là sự tinh thông gỡ lỗi thuộc bẩm sinh con người. Một số người làm việc đó rất giỏi, số khác lại không. Mặc dù bằng chứng kinh nghiệm về gỡ lỗi vẫn còn để mở cho nhiều cách hiểu, nhưng biến thiên lớn nhất trong khả năng gỡ lỗi đã được báo cáo lại đối với các kỹ sư phần mềm có cùng nền tảng kinh nghiệm và giáo dục.

**Bình luận về khía cạnh gỡ lỗi của con người, Shneiderman phát biểu:**

Gỡ lỗi là một trong những phần chán nhất của lập trình. Nó có yếu tố của việc giải quyết vấn đề hay vấn đề hóc búa, đi đôi với việc thừa nhận khó chịu rằng bạn đã sai lầm. Hay âu lo và không sẵn lòng chấp nhận khả năng lỗi làm tăng khó khăn cho công việc. May mắn là có sự giảm nhẹ và bớt căng thẳng khi lỗi cuối cùng đã được sửa lỗi.

**Mặc dầu có thể khó học được việc gỡ lỗi, người ta vẫn đề nghị ra một số cách tiếp cận tới vấn đề. Chúng ta xem xét những vấn đề này trong mục tiếp.**

### **2.2.4. Cách tiếp cận gỡ lỗi**

**Bất kể tới cách tiếp cận nào được chọn gỡ lỗi có một mục tiêu quan trọng hơn cả: tìm ra và sửa chữa nguyên nhân lỗi phần mềm. Mục tiêu này được thực hiện bằng tổ hợp**

các đánh giá có hệ thống, trực giác và may mắn. Bradley mô tả cách tiếp cận gỡ lỗi theo cách này:

Gỡ lỗi là việc ứng dụng trực tiếp phương pháp khó học đã từng được phát triển hơn 2500 năm qua. Cơ sở của việc gỡ lỗi là định vị nguồn gốc của vấn đề [nguyên nhân] bằng việc phân hoạch nhị phân, thông qua các giả thiết làm việc để dự đoán các giá trị mới cần kiểm tra.

Ta hãy lấy một ví dụ không phải phần mềm: Đèn trong nhà tôi không làm việc. Nếu không có gì trong nhà làm việc thì nguyên nhân phải là cầu chì chính hay ở bên ngoài; tôi nhìn quanh để liệu xem hàng xóm có bị tắt đèn hay không. Tôi cắm chiếc đèn nghi ngờ vào ổ cắm khác và cắm một đồ điện khác vào mạch nghi ngờ. Cứ thế tiến hành các phương án giải quyết kiểm thử.

Nói chung, có thể đưa ra ba loại các tiếp cận gỡ lỗi:

- Bó buộc mạnh bạo
- Lật ngược
- Loại bỏ nguyên nhân

Loại bó buộc mạnh bạo có lẽ là phương pháp thông dụng nhất và kém hiệu quả nhất để cô lập nguyên nhân của lỗi phần mềm. Chúng ta áp dụng phương pháp gỡ lỗi bó buộc mạnh bạo khi tất cả các phương pháp khác đều thất bại. Dùng triết lý “cứ để máy tính tìm ra lỗi”, người ta cho xỏ ra nội dung bộ nhớ, gọi tới chương trình lưu dấu vết khi chạy và nạp chương trình với lệnh WRITE. Chúng ta hy vọng rằng đâu đó trong bãi lầy thông tin được tạo ra, chúng ta có thể tìm ra được một nguyên nhân của lỗi. Mặc dầu đồng thông tin được tạo ra cuối cùng có thể dẫn tới thành công, thường hơn cả là nó dẫn đến phí phạm công sức và thời gian. Phải dành suy nghĩ vào đó trước hết đã.

Lật ngược lại cách tiếp cận khá thông dụng có thể được dùng trong những chương trình nhỏ. Bắt đầu tại chỗ chúng được phát hiện ra, lật ngược theo những chương trình gốc (một cách thủ công) cho tới chỗ tìm ra nguyên nhân. Không may là khi số dòng chương trình gốc tăng lên, số con đường lật ngược tiềm năng có thể trở nên không quản lý nổi.

Cách tiếp cận thứ ba tới gỡ lỗi - loại bỏ nguyên nhân được biểu lộ bằng việc quy nạp hay diễn dịch và đưa vào khái niệm về phân hoạch nhị phân. Dữ liệu có liên quan tới việc xuất hiện lỗi được tổ chức để cô lập ra các nguyên nhân tiềm năng. Một “giả thiết nguyên nhân” được nêu ra và dữ liệu trên được dùng để chứng minh hay bác bỏ giả thiết đó. Một cách khác, ta có thể xây dựng ra một danh sách mọi nguyên nhân đặc biệt có nhiều hứa hẹn thì dữ liệu sẽ được làm mịn thêm để cố gắng cô lập ra lỗi.

Từng cách tiếp cận gỡ lỗi trên đây đều có thể được bổ sung thêm bởi công cụ gỡ lỗi. Chúng ta có thể áp dụng một phạm vi rộng các trình biên dịch gỡ lỗi, nhưng trợ giúp gỡ lỗi động (“Bộ dò dấu vết”), các bộ sinh trường hợp kiểm thử tự động, sổ bộ nhớ và bảng tham khảo chéo. Tuy nhiên các công cụ đều không phải là cách thay thế cho việc đánh giá dựa trên tài liệu thiết kế phần mềm đầy đủ và chương trình gốc rõ ràng.

Trong nhiều trường hợp, việc gỡ lỗi phần mềm máy tính tựa như việc giải quyết vấn đề trong thế giới kinh doanh. Brow và Sampson đã đưa ra một cách tiếp cận gỡ lỗi tên là “Phương pháp”, đó là việc thích nghi các kỹ thuật giải quyết vấn đề quản lý. Các tác giả này đề nghị phát triển một bản đặc tả về các độ lệch, mô tả cho vấn đề bằng cách phác họa “cái gì, khi nào, ở đâu và với phạm vi nào?”

Mỗi một trong những vấn đề nêu trên (cái gì, khi nào, ở đâu và với phạm vi nào) đều được chỉ ra thành những đáp ứng là hay không là để phân biệt rõ rệt giữa cái gì đã xảy ra và cái gì đã không xảy ra. Một khi thông tin về lỗi đã được ghi lại thì người ta xây dựng ra một giả thiết nguyên nhân dựa trên các phân biệt quan sát được từ những đáp ứng là hay không là. Việc gỡ lỗi tiếp tục dùng cách tiếp cận qui nạp hay diễn dịch được mô tả phần trên trong mục này.

Bất kỳ thảo luận nào về cách tiếp cận và công cụ gỡ lỗi cũng đều không đầy đủ nếu không nói tới một đồng minh mạnh mẽ: Người khác! Khái niệm về “lập trình vô ngã” của Weinberg (được thảo luận trước đây trong cuốn sách này) nên được mở rộng thành gỡ lỗi vô ngã. Mỗi người chúng ta đều có thể nhớ lại điều gì khó xử khi mất hàng giờ, hàng ngày vì một lỗi dai dẳng. Một đồng nghiệp vẫn vờ đi qua trong nỗi thất vọng rồi chúng tôi giải thích và tung ra bản tin chương trình ra. Lập tức (đường như) nguyên nhân lỗi bị phát hiện ra. Mỉm cười một cách ngạo nghễ, anh bạn đồng nghiệp chúng ta biến mất. Một quan điểm mới mẻ, không bị che phủ bởi hàng giờ thất vọng, có thể tạo ra những điều kỳ diệu. Câu châm ngôn cuối cùng về gỡ lỗi có thể là: Khi tất cả mọi thứ khác đều sai thì hãy nhờ sự giúp đỡ.

Một khi lỗi đã được tìm ra, thì nó phải được sửa chữa. Nhưng khi chúng ta đã lưu ý, việc sửa một lỗi đôi khi có thể lại đưa vào một lỗi khác và do đó lại gây hại hơn là tốt. Van Vleck gợi ý ba câu hỏi đơn giản người kỹ sư phần mềm nên hỏi trước khi tiến hành sửa chữa để loại bỏ nguyên nhân gây lỗi:

- Liệu nguyên nhân gây lỗi này có bị tái tạo ở phần khác của chương trình hay không? Trong nhiều tình huống, một khiếm khuyết chương trình bị gây ra bởi một mẫu hình logic sai sót có thể còn phát sinh ở đâu đó khác nữa. Việc xem xét tường minh về mẫu hình logic này có thể làm phát hiện ra thêm các lỗi khác
- “Lỗi tiếp” có thể bị đưa vào là gì khi tôi chữa lỗi này? Trước khi việc sửa lỗi được tiến hành, chương trình gốc (hay tốt hơn, thiết kế) nên được đánh giá lại để thẩm định việc dính nối các cấu trúc logic dữ liệu. Nếu việc sửa lỗi được tiến hành trong một phần có độ dính nối cao thì càng phải đề tâm nhiều khi tiến hành bất kỳ một sự thay đổi nào.

Ta có thể làm gì để ngăn cản lỗi này ngay từ đầu? Câu hỏi này là bước đầu tiên hướng tới việc thiết lập một cách tiếp cận đảm bảo chất lượng phần mềm thống kê. Nếu ta sửa chương trình cũng như sản phẩm thì lỗi sẽ loại bỏ chương trình hiện tại và có thể bị khử bỏ mọi chương trình tương lai

## CHƯƠNG 3: KIỂM THỬ PHẦN MỀM

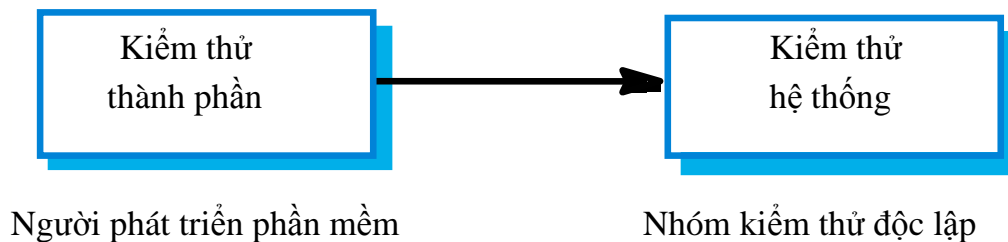
Mục tiêu của chương này là mô tả quá trình kiểm thử phần mềm và đưa ra các kỹ thuật kiểm thử. Khi đọc chương này, bạn sẽ:

- Hiểu được sự khác biệt giữa kiểm thử hợp lệ và kiểm thử khiếm khuyết.
- Hiểu được các nguyên lý của kiểm thử hệ thống và kiểm thử bộ phận.
- Hiểu được ba chiến lược có thể sử dụng để sinh các trường hợp kiểm thử hệ thống.
- Hiểu được các đặc điểm bản chất của công cụ phần mềm được sử dụng để kiểm thử tự động.

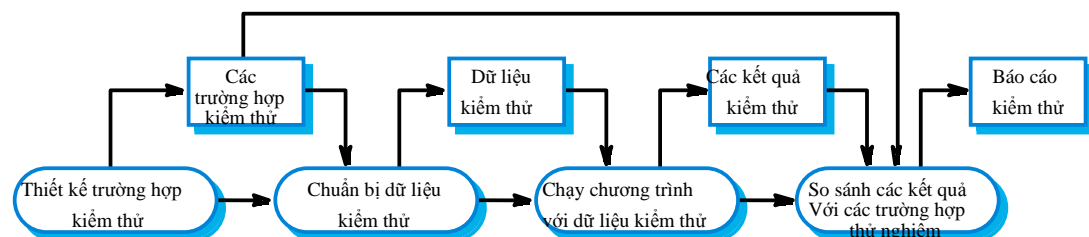
### 3.1. Quá trình kiểm thử

Quá trình kiểm thử phần mềm có hai mục tiêu riêng biệt:

1. Chứng minh cho người phát triển và khách hàng thấy các yêu cầu của phần mềm. Với phần mềm truyền thống, điều này có nghĩa là bạn có ít nhất một thử nghiệm cho mỗi yêu cầu của người dùng và tài liệu hệ thống yêu cầu. Với các sản phẩm phần mềm chung, điều đó có nghĩa là bạn nên thử nghiệm tất cả các đặc tính của hệ thống sẽ được kết hợp trong sản phẩm phát hành.
2. Phát hiện các lỗi và khiếm khuyết trong phần mềm: phần mềm thực hiện không đúng, không như mong đợi hoặc không làm theo như đặc tả. Kiểm tra khiếm khuyết tập trung vào việc tìm ra tất cả các kiểu thực hiện không như mong đợi của hệ thống, như sự đổ vỡ hệ thống, sự tương tác không mong muốn với hệ thống khác, tính toán sai và sai lệch dữ liệu.



Hình 3.1 Các giai đoạn kiểm thử



Hình 3.2 Một mô hình của quá trình kiểm thử phần mềm

Mục tiêu thứ nhất dẫn đến kiểm thử hợp lệ, sử dụng tập các thử nghiệm phản ánh mong muốn của người dùng để kiểm tra xem hệ thống có thực hiện đúng không. Mục tiêu thứ hai dẫn đến kiểm thử khiếm khuyết: các trường hợp kiểm thử được thiết kế để tìm ra các khiếm khuyết. Các trường hợp kiểm thử có thể được làm không rõ và không cần phản ánh cách hệ thống bình thường được sử dụng. Với kiểm thử hợp lệ, một thử nghiệm thành công là thử nghiệm mà hệ thống thực hiện đúng đắn. Với kiểm thử khiếm khuyết, một thử nghiệm thành công là một thử nghiệm tìm ra một khiếm khuyết, nguyên nhân làm cho hệ thống thực hiện không chính xác.

Kiểm thử có thể không chứng minh được phần mềm không có khiếm khuyết, hoặc nó sẽ thực hiện như đặc tả trong mọi trường hợp. Rất có thể một thử nghiệm bạn bỏ qua có thể phát hiện ra các vấn đề khác trong hệ thống. Như Dijkstra, một người đi đầu trong việc phát triển kỹ nghệ phần mềm, đã tuyên bố (1972): kiểm thử chỉ có thể phát hiện ra các lỗi hiện tại, chứ không thể đưa ra tất cả các lỗi.

Nói chung, vì vậy, mục tiêu của kiểm thử phần mềm là thuyết phục người phát triển phần mềm và khách hàng rằng phần mềm là đủ tốt cho các thao tác sử dụng. Kiểm thử là một quá trình được dùng để tạo nên sự tin tưởng trong phần mềm.

Mô hình tổng quát của quá trình kiểm thử được mô tả trong hình 3.2. Các trường hợp kiểm thử sự chỉ rõ của đầu vào để thử nghiệm và đầu ra mong đợi từ hệ thống cộng với một bản báo cáo sản phẩm đã được kiểm thử. Dữ liệu kiểm thử là đầu vào, được nghĩ ra để kiểm thử hệ thống. Dữ liệu kiểm thử thỉnh thoảng có thể được tự động sinh ra. Sinh các trường hợp kiểm thử tự động là điều không làm được. Đầu ra của thử nghiệm chỉ có thể được dự đoán bởi người hiêm biết về hoạt động của hệ thống.

Kiểm thử toàn diện: mọi chương trình có thể thực hiện tuần tự được kiểm tra, là điều không thể làm được. Vì vậy, kiểm thử, phải được thực hiện trên một tập con các trường hợp kiểm thử có thể xảy ra. Trong lý tưởng, các công ty phần mềm có những điều khoản để lựa chọn tập con này hơn là giao nó cho đội phát triển. Những điều khoản này có thể dựa trên những điều khoản kiểm thử chung, như một điều khoản là tất cả các câu lệnh trong chương trình nên được thực thi ít nhất một lần. Một sự lựa chọn là những điều khoản kiểm thử có thể sự trên kinh nghiệm sử dụng hệ thống, và có thể tập trung vào kiểm thử các đặc trưng hoạt động của hệ thống. Ví dụ:

1. Tất cả các đặc trưng của hệ thống được truy cập thông qua thực đơn nên được kiểm thử.
2. Kết hợp các chức năng (ví dụ định dạng văn bản) được truy cập thông qua cùng thực đơn phải được kiểm thử.
3. Khi đầu vào được đưa vào, tất cả các chức năng phải được kiểm thử với cùng một thử nghiệm đúng đắn và thử nghiệm không đúng đắn.

Điều đó rõ ràng từ kinh nghiệm với sản phẩm phần mềm lớn như phần mềm xử lý văn bản, hoặc bảng tính có thể so sánh các nguyên tắc thông thường được sử dụng trong lúc kiểm thử sản phẩm. Khi các đặc trưng của phần mềm được sử dụng cô lập, chúng làm việc bình thường. Các vấn đề phát sinh, như Whittaker giải thích (Whittaker, 2002), khi liên kết các đặc trưng không được kiểm thử cùng nhau. Ông đã đưa ra một ví dụ, khi sử dụng phần mềm xử lý văn bản sử dụng sử dụng lời chú thích ở cuối trang với cách sắp xếp nhiều cột làm cho văn bản trình bày không đúng.

**Khi một phần của quá trình lập kế hoạch V & V, người quản lý phải đưa ra các quyết định ai là người chịu trách nhiệm trong từng bước kiểm thử khác nhau. Với hầu hết các hệ thống, các lập trình viên chịu trách nhiệm kiểm thử các thành phần**

mà họ đã triển khai. Khi các lập trình viên đã hoàn thành các công việc đó, công việc được giao cho đội tổng hợp, họ sẽ tích hợp các môđun từ những người phát triển khác nhau để tạo nên phần mềm và kiểm thử toàn bộ hệ thống. Với hệ thống quan trọng, một quá trình theo nghi thức có thể được sử dụng, các người thử độc lập chịu trách nhiệm về tất cả các bước của quá trình kiểm thử. Trong kiểm thử hệ thống quan trọng, các thử nghiệm được kiểm thử riêng biệt và hồ sơ chi tiết của kết quả kiểm thử được duy trì.

Kiểm thử các thành phần được thực hiện bởi những người phát triển thường dựa trên hiểu biết trực giác về cách hoạt động của các thành phần. Tuy nhiên, kiểm thử hệ thống phải dựa trên văn bản đặc tả hệ thống. Đó có thể là một đặc tả chi tiết yêu cầu hệ thống, hoặc nó có thể là đặc tả hướng người sử dụng ở mức cao của các đặc tính được thực hiện trong hệ thống. Thường có một đội độc lập chịu trách nhiệm kiểm thử hệ thống, đội kiểm thử hệ thống làm việc từ người sử dụng và tài liệu yêu cầu hệ thống để lập kế hoạch kiểm thử hệ thống.

Hầu hết các thảo luận về kiểm thử bắt đầu với kiểm thử thành phần và sau đó chuyển đến kiểm thử hệ thống. Tôi đã đảo ngược thứ tự các thảo luận trong chương này bởi vì rất nhiều quá trình phát triển phần mềm bao gồm việc tích hợp các thành phần sử dụng lại và được lắp vào phần mềm để tạo nên các yêu cầu cụ thể. Tất cả các kiểm thử trong trường hợp này là kiểm thử hệ thống, và không có sự tách rời trong quá trình kiểm thử thành phần.

### 3.2. Kiểm thử hệ thống

Hệ thống gồm hai hoặc nhiều thành phần tích hợp nhằm thực hiện các chức năng hoặc đặc tính của hệ thống. Sau khi tích hợp các thành phần tạo nên hệ thống, quá trình kiểm thử hệ thống được tiến hành. Trong quá trình phát triển lặp đi lặp lại, kiểm thử hệ thống liên quan với kiểm thử một lượng công việc ngày càng tăng để phân phối cho khách hàng; trong quá trình thác nước, kiểm thử hệ thống liên quan với kiểm thử toàn bộ hệ thống.

Với hầu hết các hệ thống phức tạp, kiểm thử hệ thống gồm hai giai đoạn riêng biệt:

1. Kiểm thử tích hợp: đội kiểm thử nhận mã nguồn của hệ thống. Khi một vấn đề được phát hiện, đội tích hợp thử tìm nguồn gốc của vấn đề và nhận biết thành phần cần phải gỡ lỗi. Kiểm thử tích hợp hầu như liên quan với việc tìm các khiếm khuyết của hệ thống.
2. Kiểm thử phát hành: Một phiên bản của hệ thống có thể được phát hành tới người dùng được kiểm thử. Đội kiểm thử tập trung vào việc hợp lệ các yêu cầu của hệ thống và đảm bảo tính tin cậy của hệ thống. Kiểm thử phát hành thường là kiểm thử “hộp đen”, đội kiểm thử tập trung vào mô tả các đặc tính hệ thống có thể làm được hoặc không làm được. Các vấn đề được báo cáo cho đội phát triển để gỡ lỗi chương trình. Khách hàng được bao hàm trong kiểm thử phát hành, thường được gọi là kiểm thử chấp nhận. Nếu hệ thống phát hành đủ tốt, khách hàng có thể chấp nhận nó để sử dụng.

Về cơ bản, bạn có thể nghĩ kiểm thử tích hợp như là kiểm thử hệ thống chưa đầy đủ bao gồm một nhóm các thành phần. Kiểm thử phát hành liên quan đến kiểm thử hệ thống phát hành có ý định phân phối tới khách hàng. Tất nhiên, có sự gói chồng lên



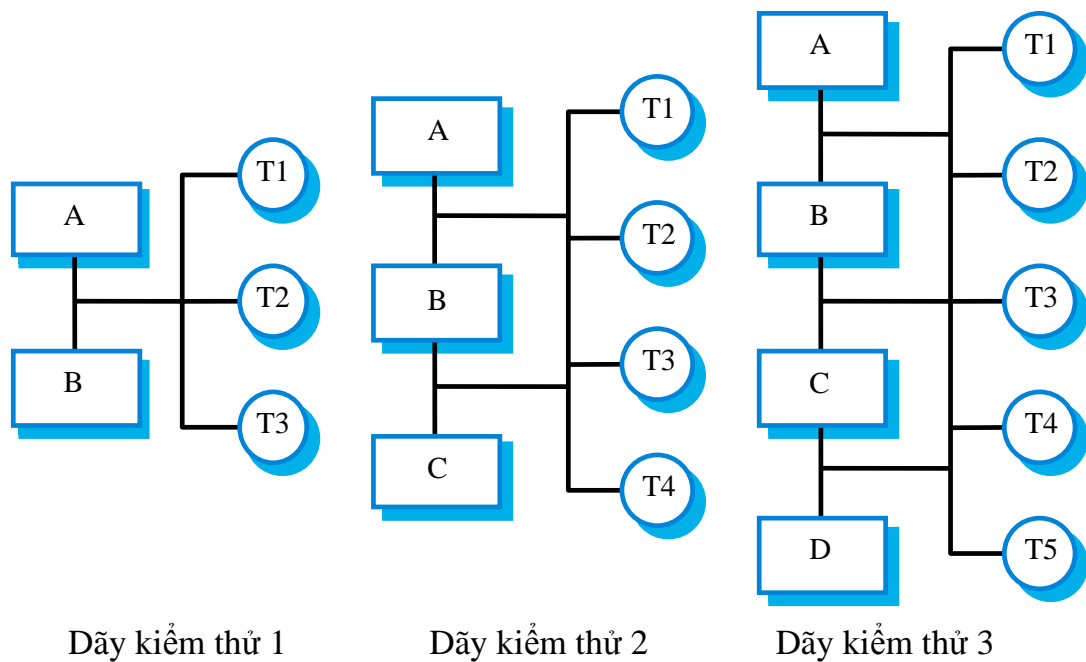
nhau, đặc biệt khi phát triển hệ thống và hệ thống được phát hành khi chưa hoàn thành. Thông thường, sự ưu tiên hàng đầu trong kiểm thử tích hợp là phát hiện ra khiếm khuyết trong hệ thống và sự ưu tiên hàng đầu trong kiểm thử hệ thống là làm hợp lệ các yêu cầu của hệ thống. Tuy nhiên trong thực tế, có vài kiểm thử hợp lệ và vài kiểm thử khiếm khuyết trong các quá trình.

### **3.3. Kiểm thử tích hợp**

Quá trình kiểm thử tích hợp bao gồm việc xây dựng hệ thống từ các thành phần và kiểm thử hệ thống tổng hợp với các vấn đề phát sinh từ sự tương tác giữa các thành phần. Các thành phần được tích hợp có thể trùng với chính nó, các thành phần có thể dùng lại được có thể thêm vào các hệ thống riêng biệt hoặc thành phần mới được phát triển. Với rất nhiều hệ thống lớn, có tất cả 3 loại thành phần được sử dụng. Kiểm thử tích hợp kiểm tra trên thực tế các thành phần làm việc với nhau, được gọi là chính xác và truyền dữ liệu đúng vào lúc thời gian đúng thông qua giao diện của chúng.

Hệ thống tích hợp bao gồm một nhóm các thành phần thực hiện vài chức năng của hệ thống và được tích hợp với nhau bằng cách gộp các mã để chúng làm việc cùng với nhau. Thỉnh thoảng, đầu tiên toàn bộ khung của hệ thống được phát triển, sau đó các thành phần được gộp lại để tạo nên hệ thống. Phương pháp này được gọi là tích hợp từ trên xuống (top-down). Một cách lựa chọn khác là đầu tiên bạn tích hợp các thành phần cơ sở cung cấp các dịch vụ chung, như mạng, truy cập cơ sở dữ liệu, sau đó các thành phần chức năng được thêm vào. Phương pháp này được gọi là tích hợp từ dưới lên (bottom-up). Trong thực tế, với rất nhiều hệ thống, chiến lược tích hợp là sự pha trộn các phương pháp trên. Trong cả hai phương pháp top-down và bottom-up, bạn thường phải thêm các mã để mô phỏng các thành phần khác và cho phép hệ thống thực hiện.

Một vấn đề chủ yếu nảy sinh trong lúc kiểm thử tích hợp là các lỗi cục bộ. Có nhiều sự tương tác phức tạp giữa các thành phần của hệ thống, và khi một đầu ra bất thường được phát hiện, bạn có thể khó nhận ra nơi mà lỗi xuất hiện. Để việc tìm lỗi cục bộ được dễ dàng, bạn nên thường xuyên tích hợp các thành phần của hệ thống và kiểm thử chúng. Ban đầu, bạn nên tích hợp một hệ thống cấu hình tối thiểu và kiểm thử hệ thống này. Sau đó bạn thêm dần các thành phần vào hệ thống đó và kiểm thử sau mỗi bước thêm vào.



Hình 3.3 Kiểm thử tích hợp lớn dần

Trong ví dụ trên hình 2.3, A,B,C,D là các thành phần và T1, T2, T3, T4, T5 là tập các thử nghiệm kết hợp các đặc trưng của hệ thống. Đầu tiên, các thành phần A và B được kết hợp để tạo nên hệ thống (hệ thống cấu hình tối thiểu), và các thử nghiệm T1, T2, T3 được thực hiện. Nếu phát hiện có khiếm khuyết, nó sẽ được hiệu chỉnh. Sau đó, thành phần C được tích hợp và các thử nghiệm T1, T2 và T3 được làm lặp lại để đảm bảo nó không tạo nên các kết quả không mong muốn khi tương tác với A và B. Nếu có vấn đề nảy sinh trong các kiểm thử này, nó hầu như chắc chắn do sự tương tác với các thành phần mới. Nguồn gốc của vấn đề đã được khoanh vùng, vì vậy làm đơn giản việc tìm và sửa lỗi. Tập thử nghiệm T4 cũng được thực hiện trên hệ thống. Cuối cùng, thành phần D được tích hợp vào hệ thống và kiểm thử được thực hiện trên các thử nghiệm đã có và các thử nghiệm mới.

Khi lập kế hoạch tích hợp, bạn phải quyết định thứ tự tích hợp các thành phần. Trong một quá trình như XP, khách hàng cũng tham gia trong quá phát triển, khách hàng quyết định các chức năng nên được thêm vào trong mỗi bước tích hợp hệ thống. Do đó, tích hợp hệ thống được điều khiển bởi sự ưu tiên của khách hàng. Trong cách tiếp cận khác để phát triển hệ thống, khi các thành phần và các thành phần riêng biệt được tích hợp, khách hàng có thể không tham gia vào quá trình tích hợp hệ thống và đội tích hợp quyết định thứ tự tích hợp các thành phần.

Trong trường hợp này, một quy tắc tốt là đầu tiên tích hợp các thành phần thực hiện hầu hết các chức năng thường sử dụng của hệ thống. Điều này có nghĩa là các thành phần thường được sử dụng hầu hết đã được kiểm thử. Ví dụ, trong hệ thống thư viện, LIBSYS, đầu tiên bạn nên tích hợp chức năng tìm kiếm trong hệ thống tối thiểu, để người dùng có thể tìm kiếm các tài mà họ cần. Sau đó, bạn nên tích hợp các chức năng cho phép người dùng tải tài liệu từ trên Internet và dần thêm các thành phần thực hiện các chức năng khác của hệ thống.

Tất nhiên, thực tế ít khi đơn giản như mô hình trên. Sự thực hiện các chức năng của hệ thống có thể liên quan đến nhiều thành phần. Để kiểm thử một đặc tính mới, bạn có thể phải tích hợp một vài thành phần khác nhau. Kiểm thử có thể phát hiện lỗi trong khi tương tác giữa các thành phần riêng biệt và các phần khác của hệ thống. Việc sửa lỗi có thể khó khăn bởi vì một nhóm các thành phần thực hiện chức năng đó có thể phải thay đổi. Hơn nữa, tích hợp và kiểm thử một thành phần mới có thể thay đổi tương tác giữa các thành phần đã được kiểm thử. Các lỗi có thể được phát hiện có thể đã không được phát hiện trong khi kiểm thử hệ thống cấu hình đơn giản.

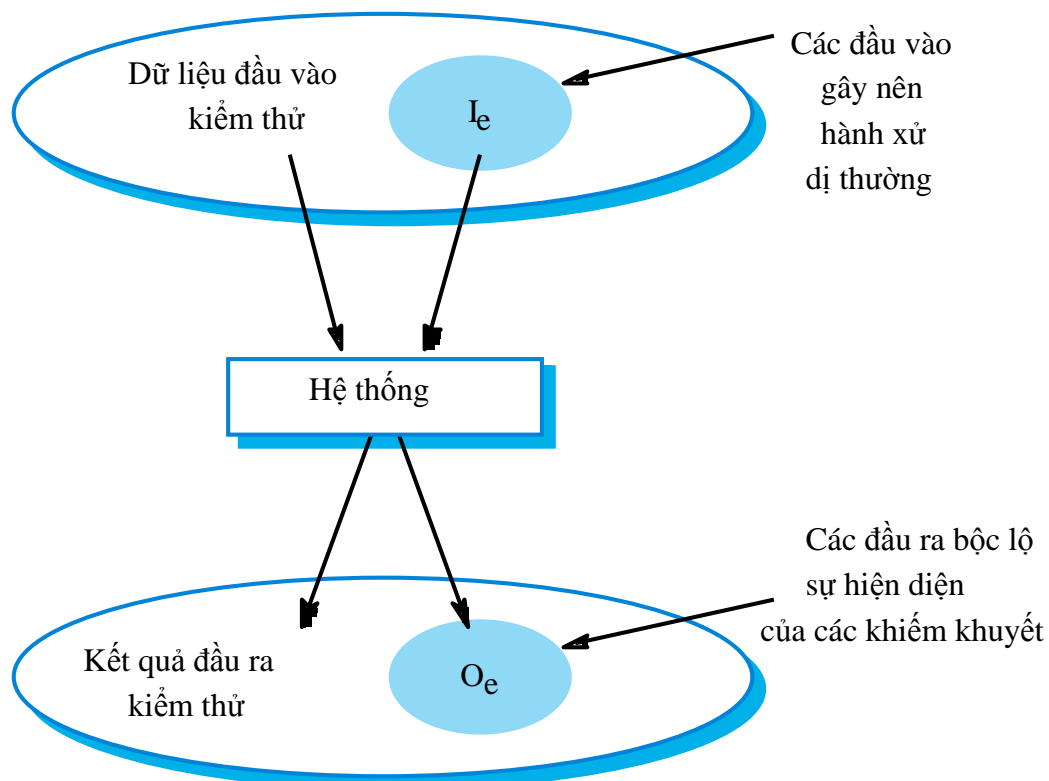
Những vấn đề này có nghĩa là khi một hệ thống tích hợp mới được tạo ra, cần phải chạy lại các thử nghiệm trong hệ thống tích hợp cũ để đảm bảo các yêu cầu các thử nghiệm đó vẫn thực hiện tốt, và các kiểm thử mới thực hiện tốt các chức năng mới của hệ thống. Việc thực hiện kiểm thử lại tập các thử nghiệm cũ gọi là kiểm thử hồi quy. Nếu kiểm thử hồi quy phát hiện có vấn đề, thì bạn phải kiểm tra có lỗi trong hệ thống cũ hay không mà hệ thống mới đã phát hiện ra, hoặc có lỗi do thêm các chức năng mới.

Rõ ràng, kiểm thử hồi quy là quá trình tốn kém, không khả thi nếu không có sự hỗ trợ tự động. Trong lập trình cực độ, tất cả các thử nghiệm được viết như mã có thể thực thi, các đầu vào thử nghiệm và kết quả mong đợi được xác định rõ và được tự động kiểm tra. Khi được sử dụng cùng với một khung kiểm thử tự động như Junit (Massol và Husted, 2003), điều này có nghĩa là các thử nghiệm có thể được tự động thực hiện lại. Đây là nguyên lý cơ bản của lập trình cực độ, khi tập các thử nghiệm toàn diện được thực hiện bất cứ lúc nào mã mới được tích hợp và các mã mới này không được chấp nhận cho đến khi tất cả các thử nghiệm được thực hiện thành công.

### **3.4. Kiểm thử phát hành**

Kiểm thử phát hành là quá trình kiểm thử một hệ thống sẽ được phân phối tới các khách hàng. Mục tiêu đầu tiên của quá trình này là làm tăng sự tin cậy của nhà cung cấp rằng sản phẩm họ cung cấp có đầy đủ các yêu cầu. Nếu thỏa mãn, hệ thống có thể được phát hành như một sản phẩm hoặc được phân phối đến các khách hàng. Để chứng tỏ hệ thống có đầy đủ các yêu cầu, bạn phải chỉ ra nó có các chức năng đặc tả, hiệu năng, và tính tin cậy cao, nó không gặp sai sót trong khi được sử dụng bình thường.

Kiểm thử phát hành thường là quá trình kiểm thử hộp đen, các thử nghiệm được lấy từ đặc tả hệ thống. Hệ thống được đối xử như chiếc hộp đen, các hoạt động của nó chỉ có thể được nhận biết qua việc nghiên cứu đầu vào và đầu ra của nó. Một tên khác của quá trình này là kiểm thử chức năng, bởi vì người kiểm tra chỉ tập trung xem xét các chức năng và không quan tâm sự thực thi của phần mềm.



Hình 3.4 Kiểm thử hộp đen

Hình 3.4 minh họa mô hình một hệ thống được kiểm thử bằng phương pháp kiểm thử hộp đen. Người kiểm tra đưa đầu vào vào thành phần hoặc hệ thống và kiểm tra đầu ra tương ứng. Nếu đầu ra không như dự báo trước (ví dụ, nếu đầu ra thuộc tập  $O_e$ ), kiểm thử phát hiện một lỗi trong phần mềm.

Khi hệ thống kiểm thử được thực hiện, bạn nên thử mở sê phần mềm bằng cách lựa chọn các trường hợp thử nghiệm trong tập  $I_e$  (trong hình 3.4). Bởi vì, mục đích của chúng ta là lựa chọn các đầu vào có xác suất sinh ra lỗi cao (đầu ra nằm trong tập  $O_e$ ). Bạn sử dụng các kinh nghiệm thành công trước đó và các nguyên tắc kiểm thử để đưa ra các lựa chọn.

Các tác giả như Whittaker (Whittaker, 2002) đã tóm lược những kinh nghiệm kiểm thử của họ trong một tập các nguyên tắc nhằm tăng khả năng tìm ra các thử nghiệm khiếm khuyết. Dưới đây là một vài nguyên tắc:

1. Lựa chọn những đầu vào làm cho hệ thống sinh ra tất cả các thông báo lỗi.
2. Thiết kế đầu vào làm cho bộ đệm đầu vào bị tràn.
3. Làm lặp lại với các đầu vào như nhau hoặc một dãy các đầu vào nhiều lần.
4. Làm sao để đầu ra không đúng được sinh ra.
5. Tính toán kết quả ra rất lớn hoặc rất nhỏ.

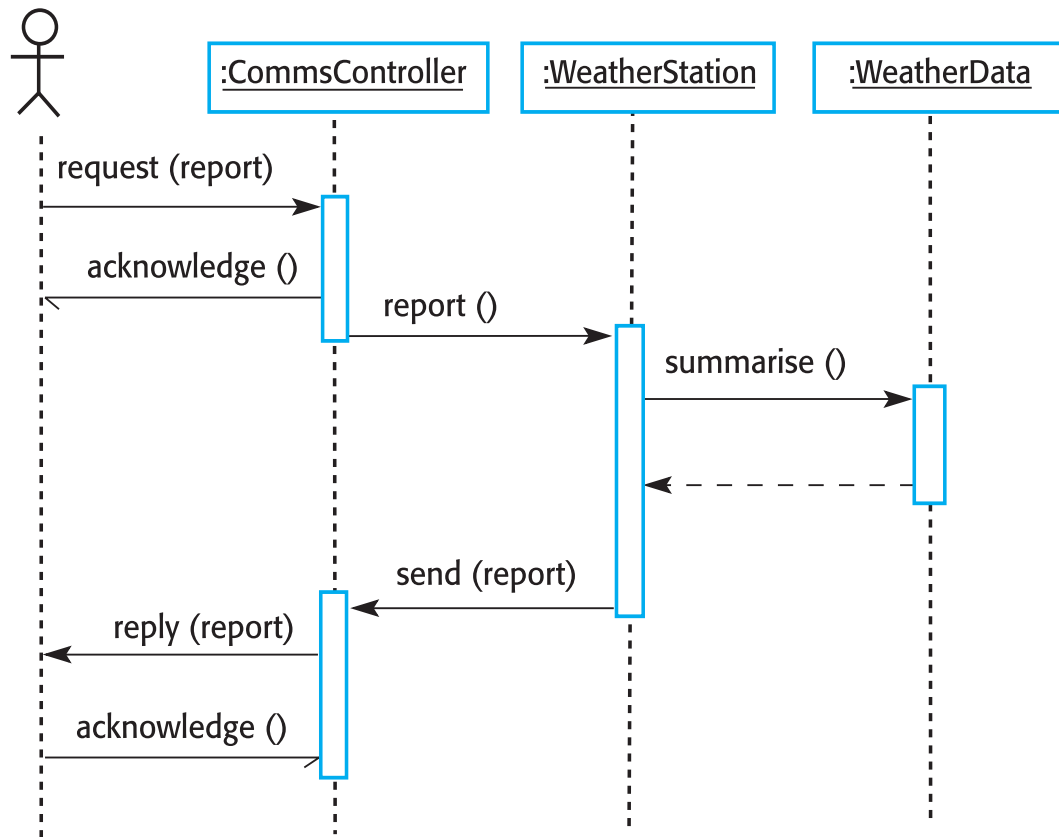
Để xác nhận hệ thống thực hiện chính xác các yêu cầu, cách tiếp cận tốt nhất vẫn đề này là kiểm thử dựa trên kịch bản, bạn đưa ra một số kịch bản và tạo nên các trường

hợp thử nghiệm từ các kịch bản đó. Ví dụ, kịch bản dưới đây có thể mô tả cách hệ thống thư viện LIBSYS, đã thảo luận trong chương trước, có thể được sử dụng:

Một sinh viên ở Scot-len nghiên cứu lịch sử nước Mỹ đã được yêu cầu viết một bài luận về “Tâm lý của người miền Tây nước Mỹ từ năm 1840 đến năm 1880”. Để làm việc đó, cô ấy cần tìm các tài liệu từ nhiều thư viện. Cô ấy đăng nhập vào hệ thống LIBSYS và sử dụng chức năng tìm kiếm để tìm xem cô ấy có được truy cập vào các tài liệu gốc trong khoảng thời gian ấy không. Cô ấy tìm được các nguồn tài liệu từ rất nhiều thư viện của các trường đại học của Mỹ, và cô ấy tải một vài bản sao các tài liệu đó. Tuy nhiên, với một vài tài liệu, cô ấy cần phải có sự xác nhận từ trường đại học của cô ấy rằng cô ấy thật sự là một sinh viên và các tài liệu được sử dụng cho những mục đích phi thương mại. Sau đó, sinh viên đó sử dụng các phương tiện của LIBSYS để yêu cầu sự cho phép và đăng ký các yêu cầu của họ. Nếu được xác nhận, các tài liệu đó sẽ được tải xuống từ máy chủ của thư viện và sau đó được in. Cô ấy nhận được một thông báo từ LIBSYS nói rằng cô ấy sẽ nhận được một e-mail khi các tài liệu đã in có giá trị để tập hợp.

Từ kịch bản trên, chúng ta có thể áp dụng một số thử nghiệm để tìm ra mục đích của LIBSYS:

1. Kiểm thử cơ chế đăng nhập bằng cách thực hiện các đăng nhập đúng và đăng nhập sai để kiểm tra người dùng hợp lệ được chấp nhận và người dùng không hợp lệ không được chấp nhận.
2. Kiểm thử cơ chế tìm kiếm bằng cách sử dụng các câu hỏi đã biết các tài liệu cần tìm để kiểm tra xem cơ chế tìm kiếm có thực sự tìm thấy các tài liệu đó.
3. Kiểm thử sự trình bày hệ thống để kiểm tra các thông tin về tài liệu có được hiển thị đúng không.
4. Kiểm thử cơ chế cho phép yêu cầu tải tài liệu xuống.
5. Kiểm thử e-mail trả lời cho biết tài liệu đã tải xuống là sẵn sàng sử dụng.



Hình 3.5 Biểu đồ dãy tập hợp dữ liệu về thời tiết

Với mỗi thử nghiệm, bạn nên thiết kế một tập các thử nghiệm bao gồm các đầu vào hợp lệ và đầu vào không hợp lệ để sinh ra các đầu ra hợp lệ và đầu ra không hợp lệ. Bạn cũng nên tổ chức kiểm thử dựa trên kịch bản, vì thế đầu tiên các kịch bản thích hợp được thử nghiệm, sau đó các kịch bản khác thường và ngoại lệ được xem xét, vì vậy sự cố gắng của bạn dành cho các phần mà hệ thống thường được sử dụng.

Nếu bạn đã sử dụng trường hợp người dùng để mô tả các yêu cầu của hệ thống, các trường hợp người dùng đó và biểu đồ liên kết nối tiếp có thể là cơ sở để kiểm thử hệ thống. Để minh họa điều này, tôi sử dụng một ví dụ từ hệ thống trạm dự báo thời tiết,

Hình 3.5 chỉ ra các thao tác lần lượt được thực hiện tại trạm dự báo thời tiết khi nó đáp ứng một yêu cầu để tập hợp dữ liệu cho hệ thống bản vẽ. Bạn có thể sử dụng biểu đồ này để nhận biết các thao tác sẽ được thử nghiệm và giúp cho việc thiết kế các trường hợp thử nghiệm để thực hiện các thử nghiệm. Vì vậy để đưa ra một yêu cầu cho một báo cáo sẽ dẫn đến sự thực hiện của một chuỗi các thao tác sau:

CommsController:request → WeatherStation:report → WeatherData:summarise

Biểu đồ đó có thể được sử dụng để nhận biết đầu vào và đầu ra cần tạo ra cho các thử nghiệm:

1. Một đầu vào của một yêu cầu báo cáo nên có một sự thừa nhận và cuối cùng báo cáo nên xuất phát từ yêu cầu. Trong lúc kiểm thử, bạn nên tạo ra dữ liệu tóm tắt, nó có thể được dùng để kiểm tra xem báo cáo được tổ chức chính xác.

2. Một yêu cầu đầu vào cho một báo cáo về kết quả của WeatherStation trong một báo cáo tóm tắt được sinh ra. Bạn có thể kiểm thử điều này một cách cô lập bằng cách tạo ra các dữ liệu thô tương ứng với bản tóm tắt, bạn đã chuẩn bị để kiểm tra CommosController và kiểm tra đối tượng WeatherStation đã được đưa ra chính xác trong bản tóm tắt.
3. Dữ liệu thô trên cũng được sử dụng để kiểm thử đối tượng WeatherData.

Tất nhiên, tôi đã làm đơn giản biểu đồ trong hình 3.5 vì nó không chỉ ra các ngoại lệ. Một kịch bản kiểm thử hoàn chỉnh cũng phải có trong bản kê khai và đảm bảo nắm bắt được đúng các ngoại lệ.

### 3.5. Kiểm thử hiệu năng

Ngay khi một hệ thống đã được tích hợp đầy đủ, hệ thống có thể được kiểm tra các thuộc tính nổi bật như hiệu năng và độ tin cậy. Kiểm thử hiệu năng phải được thiết kế để đảm bảo hệ thống có thể xử lý như mong muốn. Nó thường bao gồm việc lập một dãy các thử nghiệm, gánh nặng sẽ được tăng cho nên khi hệ thống không thể chấp nhận được nữa.

Cùng với các loại kiểm thử khác, kiểm thử hiệu năng liên quan đến cả việc kiểm chứng các yêu cầu của hệ thống và phát hiện các vấn đề và khiếm khuyết trong hệ thống. Để kiểm thử các yêu cầu hiệu năng đạt được, bạn phải xây dựng mô tả sơ lược thao tác. Mô tả sơ lược thao tác là tập các thử nghiệm phản ánh sự hòa trộn các công việc sẽ được thực hiện bởi hệ thống. Vì vậy, nếu 90% giao dịch trong hệ thống có kiểu A, 5% kiểu B và phần còn lại có kiểu C, D và E, thì chúng ta phải thiết kế mô tả sơ lược thao tác phần lớn tập trung vào kiểm thử kiểu A. Nếu không thì bạn sẽ không có được thử nghiệm chính xác về hiệu năng hoạt động của hệ thống.

Tất nhiên, cách tiếp cận này không nhất thiết là tốt để kiểm thử khiếm khuyết. Như tôi đã thảo luận, theo kinh nghiệm đã chỉ ra cách hiệu quả để phát hiện khiếm khuyết là thiết kế các thử nghiệm xung quanh giới hạn của hệ thống. Trong kiểm thử hiệu năng, điều này có nghĩa là nhấn mạnh hệ thống (vì thế nó có tên là kiểm thử nhấn mạnh) bằng cách tạo ra những đòi hỏi bên ngoài giới hạn thiết kế của phần mềm.

Ví dụ, một hệ thống xử lý các giao dịch có thể được thiết kế để xử lý đến 300 giao dịch mỗi giây; một hệ thống điều khiển có thể được thiết kế để điều khiển tới 1000 thiết bị đầu cuối khác nhau. Kiểm thử nhấn mạnh tiếp tục các thử nghiệm bên cạnh việc thiết kế lớn nhất được nạp vào hệ thống cho đến khi hệ thống gặp lỗi. Loại kiểm thử này có 2 chức năng:

1. Nó kiểm thử việc thực hiện lỗi của hệ thống. Trường hợp này có thể xuất hiện qua việc phối hợp các sự kiện không mong muốn bằng cách nạp vượt quá khả năng của hệ thống. Trong trường hợp này, sai sót của hệ thống làm cho dữ liệu bị hư hỏng hoặc không đáp ứng được yêu cầu của người dùng. Kiểm thử nhấn mạnh kiểm tra sự quá tải của hệ thống dẫn tới 'thất bại mềm' hơn là làm sụp đổ dưới lượng tải của nó.
2. Nó nhấn mạnh hệ thống và có thể gây nên khiếm khuyết trở nên rõ ràng mà bình thường không phát hiện ra. Mặc dù, nó chứng tỏ những khiếm khuyết không thể dẫn đến sự sai sót của hệ thống trong khi sử dụng bình thường, có thể hiếm gặp trong trường hợp bình thường mà kiểm thử gay gắt tái tạo.

Kiểm thử gay gắt có liên quan đặc biệt đến việc phân phối hệ thống dựa trên một mạng lưới máy xử lý. Các hệ thống thường đưa ra đòi hỏi cao khi chúng phải thực hiện nhiều công việc. Mạng trở thành bị làm mất tác dụng với dữ liệu kết hợp mà các quá trình khác nhau phải trao đổi, vì vậy các quá trình trở nên chậm hơn, như khi nó đợi dữ liệu yêu cầu từ quá trình khác.

### 3.6. Kiểm thử thành phần

Kiểm thử thành phần (thỉnh thoảng được gọi là kiểm thử đơn vị) là quá trình kiểm thử các thành phần riêng biệt của hệ thống. Đây là quá trình kiểm thử khiếm khuyết vì vậy mục tiêu của nó là tìm ra lỗi trong các thành phần. Khi thảo luận trong phần giới thiệu, với hầu hết các hệ thống, người phát triển các thành phần chịu trách nhiệm kiểm thử các thành phần. Có nhiều loại thành phần khác nhau, ta có thể kiểm thử chúng theo các bước sau:

1. Các chức năng và cách thức riêng biệt bên trong đối tượng.
2. Các lớp đối tượng có một vài thuộc tính và phương thức.
3. Kết hợp các thành phần để tạo nên các đối tượng và chức năng khác nhau. Các thành phần hỗn hợp có một giao diện rõ ràng được sử dụng để truy cập các chức năng của chúng.

Các chức năng và phương thức riêng lẻ là loại thành phần đơn giản nhất và các thử nghiệm của bạn là một tập các lời gọi tới các thủ tục với tham số đầu vào khác nhau. Bạn có thể sử dụng cách tiếp cận này để thiết kế trường hợp kiểm thử (được thảo luận trong phần sau), để thiết kế các thử nghiệm chức năng và phương thức.

Khi bạn kiểm thử các lớp đối tượng, bạn nên thiết kế các thử nghiệm để cung cấp tất cả các chức năng của đối tượng. Do đó, kiểm thử lớp đối tượng nên bao gồm:

1. Kiểm thử tất cả các thao tác cô lập liên kết tạo thành đối tượng.
2. Bố trí và kiểm tra tất cả các thuộc tính liên kết tạo thành đối tượng.
3. Kiểm tra tất cả các trạng thái của đối tượng. Điều này có nghĩa là tất cả các sự kiện gây ra các trạng thái khác nhau của đối tượng nên được mô phỏng.

Hình 3.6 Giao diện của đối tượng WeatherStation

WeatherStation
identifier
reportWeather () calibrate (instruments) test () startup (instruments) shutdown (instruments)

Ví dụ, trạm dự báo thời tiết có giao diện trình bày trên hình 3.6. Nó chỉ có một thuộc tính, là định danh của nó. Nó có một hằng số là tập thông số khi trạm dự báo thời tiết được thiết



đặt. Do đó, bạn chỉ cần một thử nghiệm để kiểm tra nó đã được thiết đặt hay chưa. Bạn cần xác định các trường hợp kiểm thử để kiểm tra reportWeather, calibrate, test, startup và shutdown. Trong trường hợp lý tưởng, bạn nên kiểm thử các phương thức riêng biệt, nhưng trong một vài trường hợp, cần có vài thử nghiệm liên tiếp. Ví dụ để kiểm thử phương thức shutdown bạn cần thực hiện phương thức startup.

Sử dụng mô hình này, bạn có thể nhận biết thứ tự của các trạng thái chuyển tiếp phải được kiểm thử và xác định thứ tự chuyển tiếp các sự kiện. Trong nguyên tắc này, bạn nên kiểm thử mọi trạng thái chuyển tiếp có thể xảy ra, mặc dù trong thực tế, điều này có thể rất tốn kém. Ví dụ dãy trạng thái nên kiểm thử trong trạm dự báo thời tiết bao gồm:

Shutdown → Waiting → Shutdown

Waiting → Calibrating → Testing → Transmitting → Waiting

Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

Nếu bạn sử dụng sự kế thừa sẽ làm cho việc thiết kế lớp đối tượng kiểm thử khó khăn hơn. Một lớp cha cung cấp các thao tác sẽ được kế thừa bởi một số lớp con, tất cả các lớp con nên được kiểm thử tất cả các thao tác kế thừa. Lý do là các thao tác kế thừa có thể đã thay đổi các thao tác và thuộc tính sau khi được kế thừa. Khi một thao tác của lớp cha được định nghĩa lại, thì nó phải được kiểm thử.

Khái niệm lớp tương đương, được thảo luận trong phần 23.3.2, có thể cũng được áp dụng cho các lớp đối tượng. Kiểm thử các lớp tương đương giống nhau có thể sử dụng các thuộc tính của đối tượng. Do đó, các lớp tương đương nên được nhận biết như sự khởi tạo, truy cập và cập nhật tất cả thuộc tính của lớp đối tượng.

### 3.7. Kiểm thử giao diện

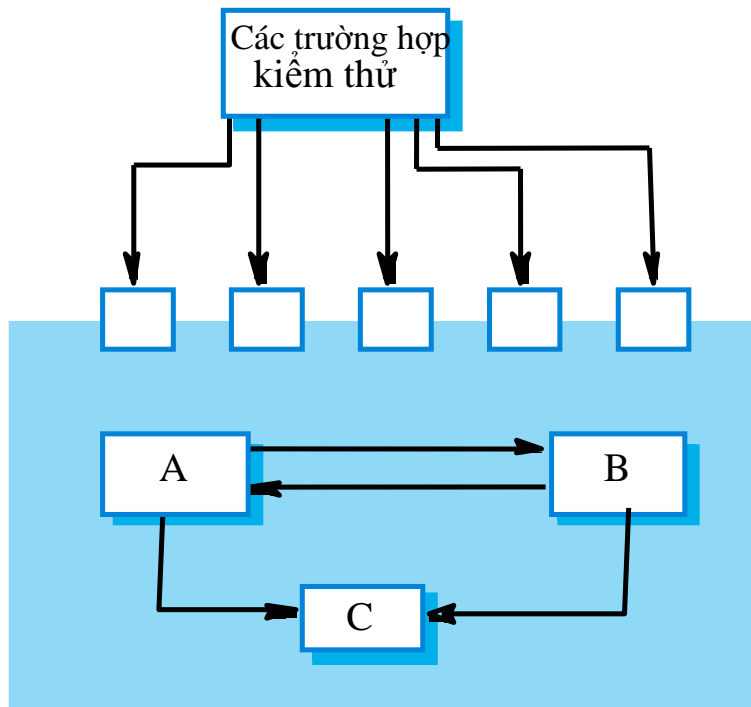
Nhiều thành phần trong một hệ thống là sự kết hợp các thành phần tạo nên bởi sự tương tác của một vài đối tượng. Kiểm thử các thành phần hỗn hợp chủ yếu liên quan đến kiểm thử hoạt động giao diện của chúng thông qua các đặc tả.

Hình 3.7 minh họa quá trình kiểm thử giao diện. Giả sử các thành phần A, B, và C đã được tích hợp để tạo nên một thành phần lớn hoặc một hệ thống con. Các thử nghiệm không chỉ áp dụng vào các thành phần riêng lẻ mà còn được áp dụng vào giao diện của các thành phần hỗn hợp được tạo nên bằng cách kết hợp các thành phần đó.

Kiểm thử giao diện đặc biệt quan trọng trong việc phát triển phần mềm hướng đối tượng và các thành phần cơ sở. Các đối tượng và các thành phần được xác định qua giao diện của chúng và có thể được sử dụng lại khi liên kết với các thành phần khác trong các hệ thống khác nhau. Các lỗi giao diện trong thành phần hỗn hợp không thể được phát hiện qua việc kiểm thử các đối tượng và các thành phần riêng lẻ. Sự tương tác giữa các thành phần trong thành phần hỗn hợp có thể phát sinh lỗi.

Có nhiều kiểu giao diện giữa các thành phần chương trình, do đó có thể xuất hiện các kiểu lỗi giao diện khác nhau:

Hình 3.7 Kiểm thử giao diện



1. Giao diện tham số: Khi dữ liệu hoặc tham chiếu chức năng được đưa từ thành phần này tới thành phần khác.
2. Giao diện chia sẻ bộ nhớ: Khi một khối bộ nhớ được chia sẻ giữa các thành phần. Dữ liệu được để trong bộ nhớ bởi một hệ thống con và được truy xuất bởi một hệ thống khác.
3. Giao diện thủ tục: Một thành phần bao gồm một tập các thủ tục có thể được gọi bởi các thành phần khác. Các đối tượng và các thành phần dùng lại có dạng giao diện này.
4. Giao diện truyền thông điệp: Một thành phần yêu cầu một dịch vụ từ một thành phần khác bằng cách gửi một thông điệp tới thành phần đó. Thông điệp trả lại bao gồm các kết quả thực hiện dịch vụ. Một vài hệ thống hướng đối tượng có dạng giao diện này như trong hệ thống chủ-khách (client-server).

Các lỗi giao diện là một dạng lỗi thường gặp trong các hệ thống phức tạp (Lutz, 1993). Các lỗi này được chia làm 3 loại:

1. Dùng sai giao diện: Một thành phần gọi tới thành phần khác và tạo nên một lỗi trong giao diện của chúng. Đây là loại lỗi rất thường gặp trong giao diện tham số: các tham số có thể được truyền sai kiểu, sai thứ tự hoặc sai số lượng tham số.
2. Hiểu sai giao diện: Một thành phần gọi tới thành phần khác nhưng hiểu sai các đặc tả giao diện của thành phần được gọi và làm sai hành vi của thành phần được gọi. Thành phần được gọi không hoạt động như mong đợi và làm cho thành phần gọi cũng hoạt động không như mong đợi. Ví dụ, một thủ tục tìm kiếm nhị phân có thể được gọi thực hiện trên một mảng chưa được xếp theo thứ tự, kết quả tìm kiếm sẽ không đúng.

3. Các lỗi trong bộ đếm thời gian: Các lỗi này xuất hiện trong các hệ thống thời gian thực sử dụng giao diện chia sẻ bộ nhớ hoặc giao diện truyền thông điệp. Dữ liệu của nhà sản xuất và dữ liệu của khách hàng có thể được điều khiển với các tốc độ khác nhau. Nếu không chú ý đến trong thiết kế giao diện, thì khách hàng có thể truy cập thông tin lỗi thời bởi vì thông tin của nhà sản xuất chưa được cập nhật trong giao diện chia sẻ.

Kiểm thử những khiếm khuyết trong giao diện rất khó khăn bởi vì một số lỗi giao diện chỉ biểu lộ trong những điều kiện đặc biệt. Ví dụ, một đối tượng có chứa một danh sách hàng đợi với cấu trúc dữ liệu có chiều dài cố định. Giả sử danh sách hàng đợi này được thực hiện với một cấu trúc dữ liệu vô hạn và không kiểm tra việc tràn hàng đợi khi một mục được thêm vào. Trường hợp này chỉ có thể phát hiện khi kiểm thử với những thử nghiệm làm cho tràn hàng đợi và làm sai hành vi của đối tượng theo những cách có thể nhận biết được.

Những lỗi khác có thể xuất hiện do sự tương tác giữa các lỗi trong các mô đun và đối tượng khác nhau. Những lỗi trong một đối tượng có thể chỉ được phát hiện khi một vài đối tượng khác hoạt động không như mong muốn. Ví dụ, một đối tượng có thể gọi một đối tượng khác để nhận được một vài dịch vụ và giả sử được đáp ứng chính xác. Nếu nó đã hiểu sai về giá trị được tính, thì giá trị trả về là hợp lệ nhưng không đúng. Điều này chỉ được phát hiện khi các tính toán sau đó có kết quả sai.

Sau đây là một vài nguyên tắc để kiểm thử giao diện:

1. Khảo sát những mã đã được kiểm thử và danh sách lời gọi tới các thành phần bên ngoài.
2. Với những tham số trong một giao diện, kiểm thử giao diện với tham số đưa vào rỗng.
3. Khi một thành phần được gọi thông qua một giao diện thủ tục, thiết kế thử nghiệm sao cho thành phần này bị sai. Các lỗi khác hầu như là do hiểu sai đặc tả chung.
4. Sử dụng kiểm thử gay gắt, như đã thảo luận ở phần trước, trong hệ thống truyền thông điệp. Thiết kế thử nghiệm sinh nhiều thông điệp hơn trong thực tế. Vấn đề bộ đếm thời gian có thể được phát hiện theo cách này.
5. Khi một vài thành phần tương tác thông qua chia sẻ bộ nhớ, thiết kế thử nghiệm với thứ tự các thành phần được kích hoạt thay đổi. Những thử nghiệm này có thể phát hiện những giả sử ngầm của các lập trình viên về thứ tự dữ liệu chia sẻ được sử dụng và được giải phóng.

Kỹ thuật hợp lệ tĩnh thường hiệu quả hơn kiểm thử để phát hiện lỗi giao diện. Một ngôn ngữ định kiểu chặt chẽ như JAVA cho phép ngăn chặn nhiều lỗi giao diện bởi trình biên dịch. Khi một ngôn ngữ không chặt chẽ như C được sử dụng, một phân tích tĩnh như LINT có thể phát hiện các lỗi giao diện. Sự kiểm tra chương trình có thể tập trung vào các giao diện giữa các thành phần và câu hỏi về hành vi giao diện xảy ra trong quá trình kiểm tra.

### **3.8. Thiết kế trường hợp thử (Test case design)**

Thiết kế trường hợp thử nghiệm là một phần của kiểm thử hệ thống và kiểm thử thành phần, bạn sẽ thiết kế các trường hợp thử nghiệm (đầu vào và đầu ra dự đoán) để kiểm thử hệ thống. Mục tiêu của quá trình thiết kế trường hợp kiểm thử là tạo ra một tập

các trường hợp thử nghiệm có hiệu quả để phát hiện khiếm khuyết của chương trình và chỉ ra các yêu cầu của hệ thống.

Để thiết kế một trường hợp thử nghiệm, bạn chọn một chức năng của hệ thống hoặc của thành phần mà bạn sẽ kiểm thử. Sau đó bạn chọn một tập các đầu thực hiện các chức năng đó, và cung cấp tài liệu về đầu ra mong muốn và giới hạn của đầu ra, và điểm mà có thể thiết kế tự động để kiểm tra thử nghiệm với đầu ra thực tế và đầu ra mong đợi vẫn như thế.

Có nhiều phương pháp khác nhau giúp bạn có thể thiết kế các trường hợp thử nghiệm:

1. Kiểm thử dựa trên các yêu cầu: Các trường hợp thử nghiệm được thiết kế để kiểm thử các yêu cầu hệ thống. Nó được sử dụng trong hầu hết các bước kiểm thử hệ thống bởi vì các yêu cầu hệ thống thường được thực hiện bởi một vài thành phần. Với mỗi yêu cầu, bạn xác định các trường hợp thử nghiệm để có thể chứng tỏ được hệ thống có yêu cầu đó.
2. Kiểm thử phân hoạch: bạn xác định các phân hoạch đầu vào và phân hoạch đầu ra và thiết kế thử nghiệm, vì vậy hệ thống thực hiện với đầu vào từ tất cả các phân hoạch và sinh ra đầu ra trong tất cả các phân hoạch. Các phân hoạch là các nhóm dữ liệu có chung đặc tính như tất cả các số đều âm, tất cả tên đều có độ dài nhỏ hơn 30 ký tự, tất cả các sự kiện phát sinh từ việc chọn các mục trên thực đơn...
3. Kiểm thử cấu trúc: Bạn sử dụng những hiểu biết về cấu trúc chương trình để thiết kế các thử nghiệm thực hiện tất cả các phần của chương trình. Về cơ bản, khi kiểm thử một chương trình, bạn nên kiểm tra thực thi mỗi câu lệnh ít nhất một lần. Kiểm thử cấu trúc giúp cho việc xác định các trường hợp thử nghiệm.

Thông thường, khi thiết kế các trường hợp thử nghiệm, bạn nên bắt đầu với các thử nghiệm mức cao nhất của các yêu cầu, sau đó thêm dần các thử nghiệm chi tiết bằng cách sử dụng kiểm thử phân hoạch và kiểm thử cấu trúc.

### **3.8.1. Kiểm thử dựa trên các yêu cầu**

Một nguyên lý chung của các yêu cầu kỹ nghệ là các yêu cầu phải có khả năng kiểm thử được. Các yêu cầu nên được viết theo cách mà một thử nghiệm có thể được thiết kế, do đó quan sát viên có thể kiểm tra xem yêu cầu đó đã thỏa mãn chưa. Vì vậy, kiểm thử dựa trên các yêu cầu là một tiếp cận có hệ thống để thiết kế trường hợp thử nghiệm giúp cho bạn xem xét mỗi yêu cầu và tìm ra các thử nghiệm. Kiểm thử dựa trên các yêu cầu có hiệu quả hơn kiểm thử khiếm khuyết – bạn đang chứng tỏ hệ thống thực hiện được đầy đủ các yêu cầu.

Ví dụ, hãy xem xét các yêu cầu cho hệ thống LIBSYS .

1. Người dùng có thể tìm kiếm hoặc tất cả các tập ban đầu của cơ sở dữ liệu hoặc lựa chọn một tập con từ đó.
2. Hệ thống sẽ cung cấp các khung nhìn hợp lý cho người dùng để đọc tài liệu trong kho tài liệu.
3. Mọi yêu cầu sẽ được cấp phát một định danh duy nhất (ORDER\_ID) để người dùng có thể được phép sao chép qua tài khoản của vùng lưu trữ thường trực.

Giả sử chức năng tìm kiếm đã được kiểm thử, thì các thử nghiệm có thể chấp nhận được cho yêu cầu thứ nhất là:

- Ban đầu, người dùng tìm kiếm các mục mà đã biết sự có mặt và đã biết không có trong tập cơ sở dữ liệu chỉ gồm có một cơ sở dữ liệu.

- Ban đầu, người dùng tìm kiếm các mục mà đã biết sự có mặt và đã biết không có trong tập cơ sở dữ liệu gồm có hai cơ sở dữ liệu.
- Ban đầu, người dùng tìm kiếm các mục mà đã biết sự có mặt và đã biết không có trong tập cơ sở dữ liệu gồm có nhiều hơn hai cơ sở dữ liệu.
- Lựa chọn một cơ sở dữ liệu từ tập cơ sở dữ liệu, người dùng tìm kiếm các mục mà đã biết sự có mặt và đã biết không có trong cơ sở dữ liệu đó.
- Lựa chọn nhiều hơn một cơ sở dữ liệu từ tập cơ sở dữ liệu, người dùng tìm kiếm các mục mà đã biết sự có mặt và đã biết không có trong cơ sở dữ liệu đó.

Từ đó, bạn có thể thấy kiểm thử một yêu cầu không có nghĩa là chỉ thực hiện kiểm thử trên một thử nghiệm. Thông thường, bạn phải thực kiểm thử nghiệm trên một vài thử nghiệm để đảm bảo bạn đã kiểm soát được yêu cầu đó.

Kiểm thử các yêu cầu khác trong hệ thống LIBSYS có thể được thực hiện theo giống như trên. Với yêu cầu thứ hai, bạn sẽ soạn ra các thử nghiệm để phân phối tất cả các kiểu tài liệu có thể được xử lý bởi hệ thống và kiểm tra sự hiển thị các tài liệu đó. Với yêu cầu thứ ba, bạn giả vờ đưa vào một vài yêu cầu, sau đó kiểm tra định danh yêu cầu được hiển thị trong giấy chứng nhận của người dùng, và kiểm tra định danh yêu cầu đó có là duy nhất hay không.

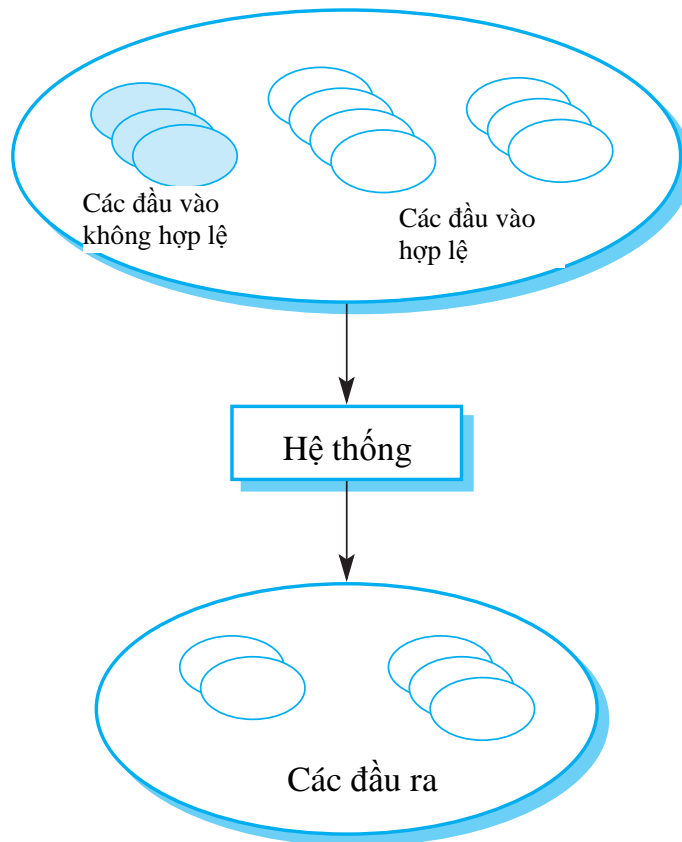
### 3.8.2. Kiểm thử phân hoạch

Dữ liệu đầu vào và kết quả đầu ra của chương trình thường được phân thành một số loại khác nhau, mỗi loại có những đặc trưng chung, như các số đều dương, các số đều âm, và các thực đơn lựa chọn. Thông thường, các chương trình thực hiện theo cách có thể so sánh được với tất cả thành viên của một lớp. Do đó, nếu chương trình được kiểm thử thực hiện những tính toán và yêu cầu hai số dương, thì bạn sẽ mong muốn chương trình thực hiện theo cách như nhau với tất cả các số dương.

Bởi vì cách thực hiện là tương đương, các loại này còn được gọi là phân hoạch tương đương hay miền tương đương (Bezier, 1990). Một cách tiếp cận có hệ thống để thiết kế các trường hợp kiểm thử là dựa trên sự định danh của tất cả các phân hoạch trong một hệ thống hoặc một thành phần. Các trường hợp thử nghiệm được thiết kế sao cho đầu vào và đầu ra nằm trong phân hoạch đó. Kiểm thử phân hoạch có thể được sử dụng để thiết kế các trường hợp thử nghiệm cho các hệ thống và các thành phần.

Trong hình 3.8, mỗi phân hoạch tương đương được biểu thị như một elíp. Đầu vào các phân hoạch tương đương là những tập dữ liệu, tất cả các tập thành viên nên được xử lý một cách tương đương. Đầu ra phân hoạch tương là đầu ra của chương trình và chúng có các đặc trưng chung, vì vậy chúng có thể được kiểm tra như một lớp riêng biệt. Bạn cũng xác định các phân hoạch có đầu vào ở bên ngoài các phân hoạch khác. Kiểm tra các thử nghiệm mà chương trình sử dụng đầu vào không hợp lệ có thực hiện đúng cách thức không. Các đầu vào hợp lệ và đầu vào không hợp lệ cũng được tổ chức thành các phân hoạch tương đương.

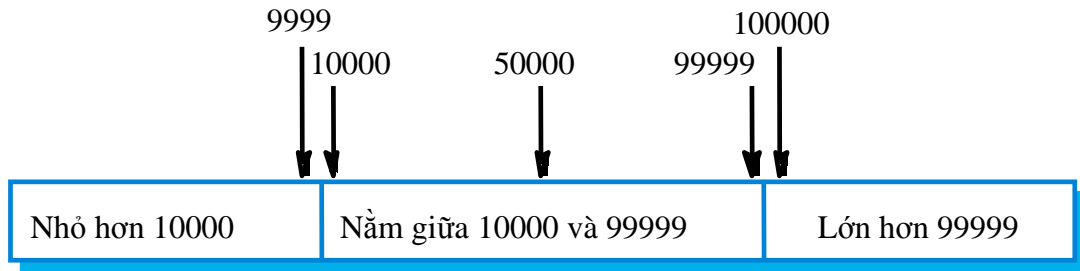
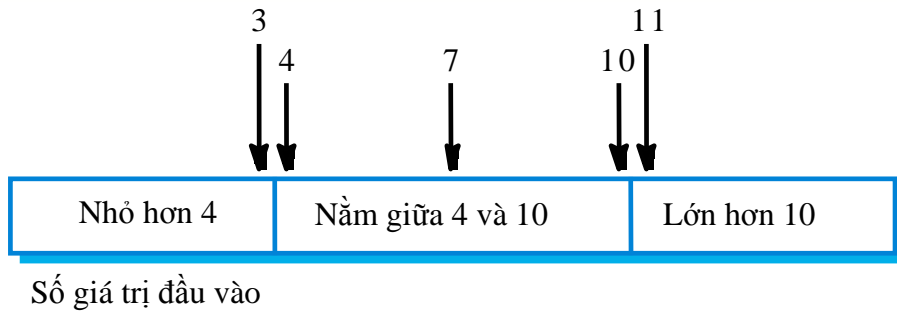
Hình 3.8  
Phân hoạch  
tương đương



Khi bạn đã xác định được tập các phân hoạch, bạn có thể lựa chọn các trường hợp thử nghiệm cho mỗi phân hoạch đó. Một quy tắc tốt để lựa chọn trường hợp thử nghiệm là lựa chọn các trường hợp thử nghiệm trên các giới hạn của phân hoạch cùng với các thử nghiệm gần với điểm giữa của phân hoạch. Lý do căn bản là người thiết kế và lập trình viên thường xem xét các giá trị đầu vào điển hình khi phát triển một hệ thống. Bạn kiểm thử điều đó bằng cách lựa chọn điểm giữa của hệ thống. Các giá trị giới hạn thường không điển hình (ví dụ, số 0 có thể được sử dụng khác nhau trong các tập các số không âm), vì vậy nó không được người phát triển chú ý tới. Các lỗi của chương trình thường xuất hiện khi nó xử lý các giá trị không điển hình.

Bạn xác định các phân hoạch bằng cách sử dụng đặc tả chương trình hoặc tài liệu hướng dẫn sử dụng, và từ kinh nghiệm của mình, bạn dự đoán các loại giá trị đầu vào thích hợp để phát hiện lỗi. Ví dụ, từ đặc trưng của chương trình: chương trình chấp nhận từ 4 đến 8 đầu vào là các số nguyên có 5 chữ số lớn hơn 10 000. Hình 3.9 chỉ ra các phân hoạch cho tình huống này và các giá trị đầu vào có thể xảy ra.

Để minh họa cho nguồn gốc của những trường hợp thử nghiệm này, sử dụng các đặc tả của thành phần tìm kiếm (trên hình 3.10). Thành phần này tìm kiếm trên một dãy các phần tử để đưa ra phần tử mong muốn (phần tử khóa). Nó trả lại vị trí của phần tử đó trong dãy. Tôi đã chỉ rõ đây là một cách trừu tượng để xác định các điều kiện tiên quyết phải đúng trước khi thành phần đó được gọi, và các hậu điều kiện phải đúng sau khi thực hiện.



**Hình 3.9 Các phân hoạch tương đương**

Điều kiện tiên quyết: Thử tục tìm kiếm sẽ chỉ làm việc với các dãy không rỗng. Hậu điều kiện: biến **Found** được thiết đặt nếu phần tử khóa thuộc dãy. Phần tử khóa có chỉ số L. Giá trị chỉ số không được xác định nếu phần tử đó không thuộc dãy.

Từ đặc trưng đó, bạn có thể nhận ra hai phân hoạch tương đương:

1. Các đầu vào có phần tử khóa là một phần tử của dãy (Found = true).
2. Các đầu vào có phần tử khóa không phải là một phần tử của dãy (Found = false).

**procedure** Search (Key : ELEM ; T: SEQ of ELEM;

Found : **in out** BOOLEAN; L: **in out** ELEM\_INDEX) ;

**Tiền điều kiện**

-- Dãy có ít nhất một phần tử

T'FIRST <= T'LAST

**Hậu điều kiện**

-- Phần tử được tìm thấy và được chỉ bởi L

( Found and T (L) = Key)

**hoặc**

-- Phần tử không thuộc dãy

( **not** Found and

**not** (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))

Hình 3.10 Đặc tả chương trình tìm kiếm

Dãy	Phần tử
Có một giá trị	Thuộc dãy
Có một giá trị	Không thuộc dãy
Nhiều hơn một giá trị	Là phần tử đầu tiên trong dãy
Nhiều hơn một giá trị	Là phần tử cuối cùng trong dãy
Nhiều hơn một giá trị	Là phần tử nằm giữa trong dãy
Nhiều hơn một giá trị	Không thuộc dãy

Dãy đầu vào	Khóa (Key)	Đầu ra (Found,L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Hình 3.11 Các phân hoạch tương đương cho chương trình tìm kiếm

Khi bạn thử nghiệm chương trình với các dãy, mảng hoặc danh sách, một số nguyên tắc thường được sử dụng để thiết kế các trường hợp kiểm thử:

1. Kiểm thử phần mềm với dãy chỉ có một giá trị. Lập trình viên thường nghĩ các dãy gồm vài giá trị, và thỉnh thoảng, họ cho rằng điều này luôn xảy ra trong các chương trình của họ. Vì vậy, chương trình có thể không làm việc chính xác khi dãy được đưa vào chỉ có một giá trị.
2. Sử dụng các dãy với các kích thước khác nhau trong các thử nghiệm khác nhau. Điều này làm giảm cơ hội một chương trình khiếm khuyết sẽ ngẫu nhiên đưa ra đầu ra chính xác bởi vì các đầu vào có các đặc tính ngẫu nhiên.
3. Xuất phát từ các thử nghiệm có phần tử đầu tiên, phần tử ở giữa, và phần tử cuối cùng được truy cập. Cách tiếp cận này bộc lộ các vấn đề tại các giới hạn phân hoạch.

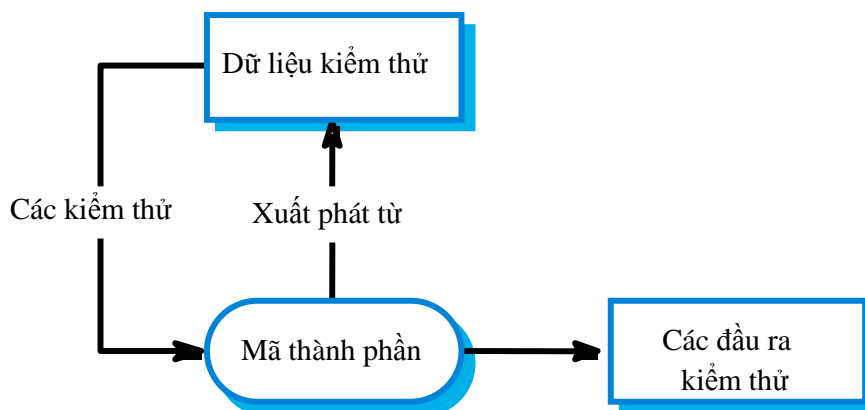
Từ các nguyên tắc trên, hai phân hoạch tương đương có thể được xác định:

1. Dãy đầu vào có một giá trị.
2. Số phần tử trong dãy đầu vào lớn hơn 1.

Sau khi, bạn xác định thêm các phân hoạch bằng cách kết hợp các phân hoạch đã có, ví dụ, kết hợp phân hoạch có số phần tử trong dãy lớn hơn 1 và phần tử khóa không thuộc



dãy. Hình 3.11 đưa ra các phân hoạch mà bạn đã xác định để kiểm thử thành phần tìm kiếm.

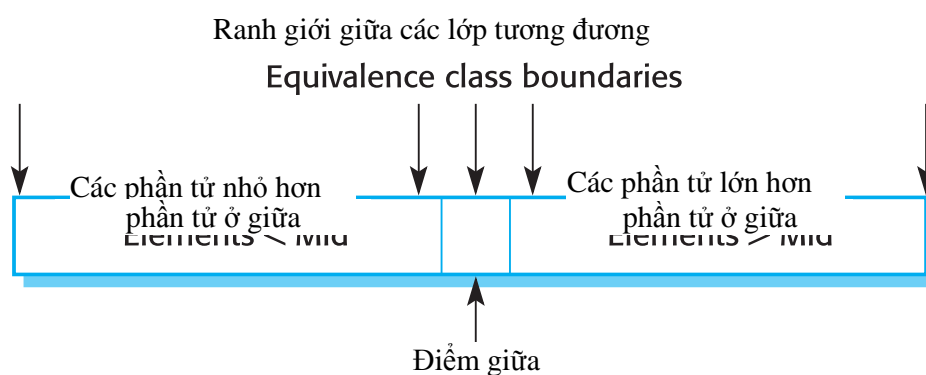


**Hình 3.12 Kiểm thử cấu trúc**

Một tập các trường hợp thử nghiệm có thể dựa trên các phân hoạch đó cũng được đưa ra trên hình 3.11. Nếu phần tử khóa không thuộc dãy, giá trị của  $L$  là không xác định (“?”). Nguyên tắc “các dãy với số kích thước khác nhau nên được sử dụng” đã được áp dụng trong các trường hợp thử nghiệm này.

Tập các giá trị đầu vào sử dụng để kiểm thử thủ tục tìm kiếm không bao giờ hết. Thủ tục này có thể gặp lỗi nếu dãy đầu vào tình cờ gồm các phần tử 1, 2, 3 và 4. Tuy nhiên, điều đó là hợp lý để giả sử: nếu thử nghiệm không phát hiện khiếm khuyết khi một thành viên của một loại được xử lý, không có thành viên khác của lớp sẽ xác định các khiếm khuyết. Tất nhiên, các khiếm khuyết sẽ vẫn tồn tại. Một vài phân hoạch tương đương có thể không được xác định, các lỗi có thể đã được tạo ra trong phân hoạch tương đương hoặc dữ liệu thử nghiệm có thể đã được chuẩn bị không đúng.

### 3.8.3. Kiểm thử cấu trúc



**Hình 3.13 Các lớp tương đương trong tìm kiếm nhị phân**

Kiểm thử cấu trúc (hình 3.12) là một cách tiếp cận để thiết kế các trường hợp kiểm thử, các thử nghiệm được xác định từ sự hiểu biết về cấu trúc và sự thực hiện của phần mềm. Cách tiếp cận này thỉnh thoảng còn được gọi là kiểm thử “hộp trắng”, “hộp kính”, hoặc kiểm thử “hộp trong” để phân biệt với kiểm thử hộp đen.

```

Class BinSearch {

// Đây là một hàm tìm kiếm nhị phân được thực hiện trên một dãy các
// đối tượng đã có thứ tự và một khóa, trả về một đối tượng với 2 thuộc
// tính là:
// index – giá trị chỉ số của khóa trong dãy
// found – có kiểu logic cho biết có hay không có khóa trong dãy
// Một đối tượng được trả về bởi vì trong Java không thể thông qua các
// kiểu cơ bản bằng tham chiếu tới một hàm và trả về hai giá trị
// Giá trị index = -1 nếu khóa không có trong dãy

    public static void search( int key, int[] elemArray, Result r)
    {
1.        int bottom = 0;
2.        int top = elemArray.length - 1;
           int mid;
3.        r.found = false;
4.        r.index = -1;
5.        while (bottom <= top)
           {
6.            mid = (top + bottom) / 2;
7.            if (elemArray[mid] = key)
           {
8.                r.index = mid;
9.                r.found = true;
10.           return;
           } // if part
           else
           {
11.             if (elemArray[mid] < key)
12.                 bottom = mid + 1;
           else
13.                 top = mid - 1;
           }
           } // while loop
14.    } // search
    } // BinSearch

```

**Hình 3.14 Chương trình tìm kiếm nhị phân được viết bằng Java**

Hiểu được cách sử dụng thuật toán trong một thành phần có thể giúp bạn xác định thêm các phân hoạch và các trường hợp thử nghiệm. Để minh họa điều này, tôi đã thực hiện cách đặc tả thủ tục tìm kiếm (hình 3.10) như một thủ tục tìm kiếm nhị phân (hình 3.14). Tất nhiên, điều kiện tiên quyết đã được bảo đảm nghiêm ngặt. Dãy được thực thi

Dãy đầu vào (T)	Khóa (Key)	Đầu ra (Found,L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30,31,41,45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

**Hình 3.15 Các trường hợp kiểm thử cho chương trình tìm kiếm**

như một mảng và mảng này phải được sắp xếp và giá trị giới hạn dưới phải nhỏ hơn giá trị giới hạn trên.

Để kiểm tra mã của thủ tục tìm kiếm, bạn có thể xem việc tìm kiếm nhị phân chia không gian tìm kiếm thành 3 phần. Mỗi phần được tạo bởi một phân hoạch tương đương (hình 3.13). Sau đó, bạn thiết kế các trường hợp thử nghiệm có phần tử khóa nằm tại các giới hạn của mỗi phân hoạch.

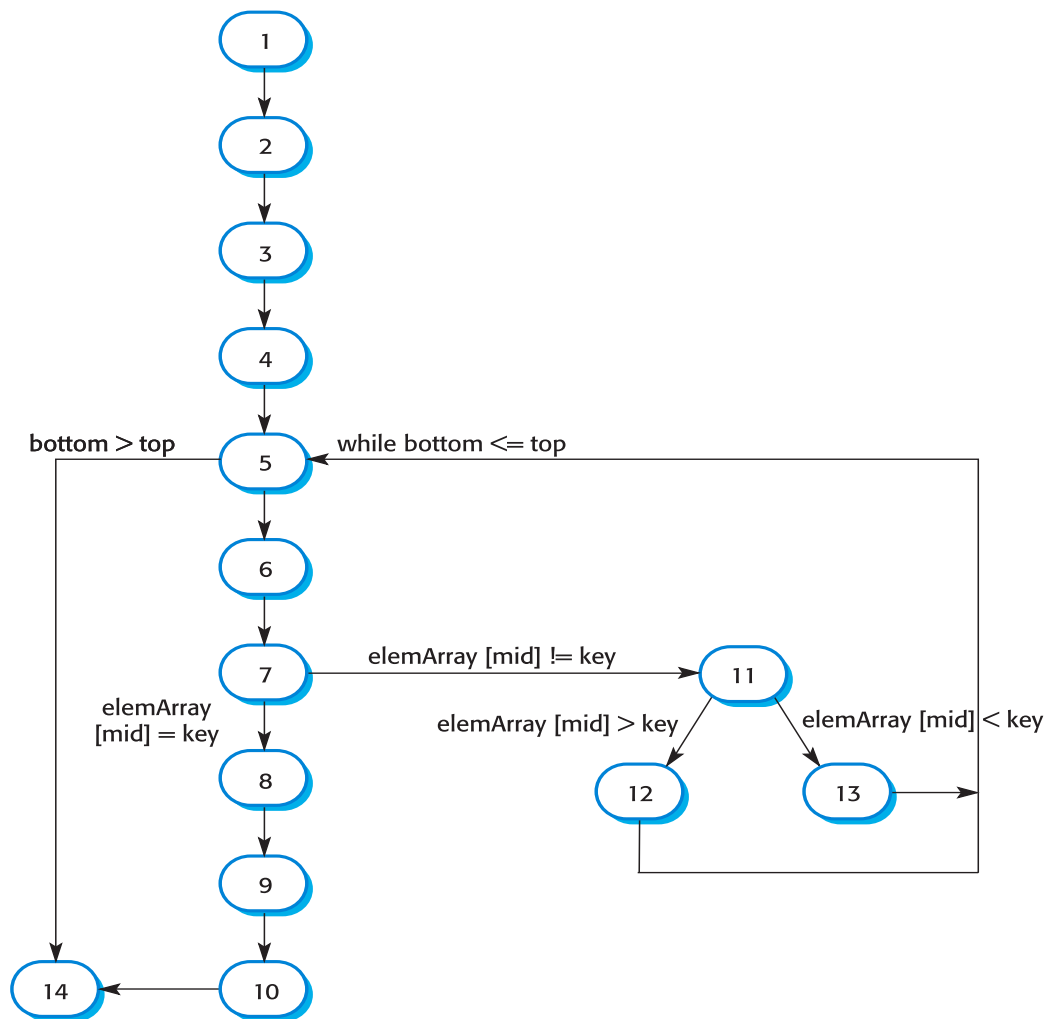
Điều này đưa đến một tập sửa lại của các trường hợp thử nghiệm cho thủ tục tìm kiếm, như trên hình 3.15. Chú ý, đã sửa đổi mảng đầu vào vì vậy nó đã được sắp xếp theo thứ tự tăng dần và đã thêm các thử nghiệm có phần tử khóa kề với phần tử giữa của mảng.

#### 3.8.4. Kiểm thử đường dẫn

Kiểm thử đường dẫn là một chiến lược kiểm thử cấu trúc. Mục tiêu của kiểm thử đường dẫn là thực hiện mọi đường dẫn thực hiện độc lập thông qua một thành phần hoặc chương trình. Nếu mọi đường dẫn thực hiện độc lập được thực hiện, thì tất cả các câu lệnh trong thành phần đó phải được thực hiện ít nhất một lần. Hơn nữa, tất cả câu lệnh điều kiện phải được kiểm thử với cả trường hợp đúng và sai. Trong quá trình phát triển hướng đối tượng, kiểm thử đường dẫn có thể được sử dụng khi kiểm thử các phương thức liên kết với các đối tượng.

Số lượng đường dẫn qua một chương trình thường tỷ lệ với kích thước của nó. Khi tất cả các môđun được tích hợp trong hệ thống, nó trở nên không khả thi để sử dụng kỹ thuật kiểm thử cấu trúc. Vì thế, kỹ thuật kiểm thử đường dẫn hầu như được sử dụng trong lúc kiểm thử thành phần.

Kiểm thử đường dẫn không kiểm tra tất cả các kết hợp có thể của các đường dẫn qua chương trình. Với bất kỳ thành phần nào ngoài các thành phần rất tầm thường không có vòng lặp, đây là mục tiêu không khả thi. Trong chương trình có các vòng lặp sẽ có một số vô hạn khả năng kết hợp đường dẫn. Thậm chí, khi tất cả các lệnh của chương trình đã được thực hiện ít nhất một lần, các khiếm khuyết của chương trình vẫn có thể được đưa ra khi các đường dẫn đặc biệt được kết hợp.



**Hình 2.16 Đồ thị luồng của chương trình tìm kiếm nhị phân**

Điểm xuất phát để kiểm thử đường dẫn là đồ thị luồng chương trình. Đây là mô hình khung của tất cả đường dẫn qua chương trình. Một đồ thị luồng chứa các nút miêu tả các quyết định và các cạnh trình bày luồng điều khiển. Đồ thị luồng được xây dựng bằng cách thay đổi các câu lệnh điều khiển chương trình sử dụng biểu đồ tương đương. Nếu không có các câu lệnh goto trong chương trình, đó là một quá trình đơn giản xuất phát từ đồ thị luồng. Mỗi nhánh trong câu lệnh điều kiện (if-then-else hoặc case) được miêu tả như một đường dẫn riêng biệt. Mỗi mũi tên trở lại nút điều kiện miêu tả một vòng lặp. Tôi đã vẽ đồ thị luồng cho phương thức tìm kiếm nhị phân trên hình 3.16. Để tạo nên sự tương ứng giữa đồ thị này và chương trình trên hình 3.14 được rõ ràng, tôi đã miêu tả mỗi câu lệnh như một nút riêng biệt, các số trong mỗi nút tương ứng với số dòng trong chương trình.

Mục đích của kiểm thử đường dẫn là đảm bảo mỗi đường dẫn độc lập qua chương trình được thực hiện ít nhất một lần. Một đường dẫn chương trình độc lập là một đường đi ngang qua ít nhất một cạnh mới trong đồ thị luồng. Cả nhánh đúng và nhánh sai của các điều kiện phải được thực hiện.

Đồ thị luồng cho thủ tục tìm kiếm nhị phân được miêu tả trên hình 3.16, mỗi nút biểu diễn một dòng trong chương trình với một câu lệnh có thể thực hiện được. Do đó, bằng

cách lần vết trên đồ thị luồng, bạn có thể nhận ra các đường dẫn qua đồ thị luồng tìm kiếm nhị phân:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14

1, 2, 3, 4, 5, 14

1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...

1, 2, 3, 4, 5, 6, 7, 11, 13, 5, ...

Nếu tất cả các đường dẫn được thực hiện, chúng ta có thể đảm bảo mọi câu lệnh trong phương thức đã được thực hiện ít nhất một lần và mỗi nhánh đã được thực hiện với các điều kiện đúng và sai.

Bạn có thể tìm được số lượng các đường dẫn độc lập trong một chương trình bằng tính toán vòng liên hợp (McCabe, 1976) trong đồ thị luồng chương trình. Với chương trình không có câu lệnh goto, giá trị vòng liên hợp là nhiều hơn số câu lệnh điều kiện trong chương trình. Một điều kiện đơn là một biểu thức logic không có các liên kết “and” hoặc “or”. Nếu chương trình bao gồm các điều kiện phức hợp, là các biểu thức logic bao gồm các liên kết “and” và “or”, thì bạn phải đếm số điều kiện đơn trong các điều kiện phức hợp khi tính số vòng liên hợp.

Vì vậy, nếu có 6 câu lệnh “if” và 1 vòng lặp “while” và các biểu thức điều kiện là đơn, thì số vòng liên hợp là 8. Nếu một biểu thức điều kiện là biểu thức phức hợp như “if A and B or C”, thì bạn tính nó như 3 điều kiện đơn. Do đó, số vòng liên hợp là 10. Số vòng liên hợp của thuật toán tìm kiếm nhị phân (hình 3.14) là 4 bởi vì nó có 3 điều kiện đơn tại các dòng 5, 7, 11.

Sau khi tính được số đường dẫn độc lập qua mã chương trình bằng tính toán số vòng liên hợp, bạn thiết kế các trường hợp thử nghiệm để thực hiện mỗi đường dẫn đó. Số lượng trường hợp thử nghiệm nhỏ nhất bạn cần để kiểm tra tất cả các đường dẫn tổng chương trình bằng số vòng liên hợp.

Thiết kế trường hợp thử nghiệm không khó khăn trong trường hợp chương trình là thủ tục tìm kiếm nhị phân. Tuy nhiên, khi chương trình có cấu trúc nhánh phức tạp, có thể rất khó khăn để dự đoán có bao nhiêu thử nghiệm đã được thực hiện. Trong trường hợp đó, một người phân tích chương trình năng động có thể được sử dụng để phát hiện sơ thảo sự thực thi của chương trình.

Những người phân tích chương trình năng động là các công cụ kiểm thử, cùng làm việc với trình biên dịch. Trong lúc biên dịch, những người phân tích này thêm các chỉ thị phụ để sinh ra mã. Chúng đếm số lần mỗi câu lệnh đã được thực hiện. Sau khi chương trình đã thực hiện, một bản sơ thảo thực thi có thể được in ra. Nó chỉ ra những phần chương trình đã thực thi và đã không thực thi bằng cách sử dụng các trường hợp thử nghiệm đặc biệt. Vì vậy, bản sơ thảo thực thi cho phép phát hiện các phần chương trình không được kiểm thử.

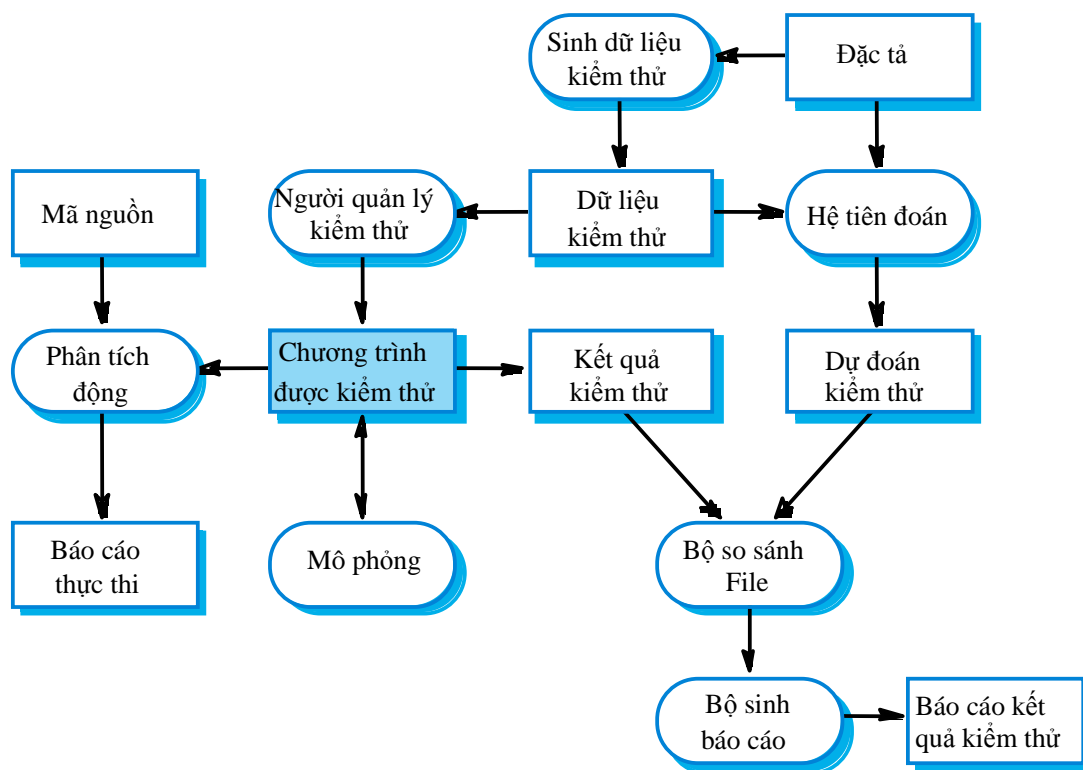
### **3.9. Tự động hóa kiểm thử (Test automation)**

Kiểm thử là một giai đoạn tốn kém và nặng nề trong quy trình phần mềm. Kết quả là những công cụ kiểm thử là một trong những công cụ phần mềm đầu tiên được phát triển. Hiện nay, các công cụ này đã bộc lộ nhiều sự thuận tiện và chúng làm giảm đáng kể chi phí kiểm thử.

Tôi đã thảo luận một cách tiếp cận để tự động hóa kiểm thử (Mosley và Posey, 2002) với một khung kiểm thử như JUnit (Massol và Husted, 2003) được sử dụng kiểm thử phức hồi. JUnit là một tập các lớp Java được người dùng mở rộng để tạo nên môi trường kiểm thử tự động. Mỗi thử nghiệm riêng lẻ được thực hiện như một đối tượng và một chương trình đang chạy thử nghiệm chạy tất cả các thử nghiệm đó. Các thử nghiệm đó nên được viết theo cách để chúng chỉ ra hệ thống kiểm thử có thực hiện như mong muốn không.

Một phần mềm kiểm thử workbench là một tập tích hợp các công cụ để phục vụ cho quá trình kiểm thử. Hơn nữa với các khung kiểm thử cho phép thực hiện kiểm thử tự động, một workbench có thể bao gồm các công cụ để mô phỏng các phần khác của hệ thống và để sinh ra dữ liệu thử nghiệm hệ thống. Hình 3.17 đưa ra một vài công cụ có thể bao gồm trong một workbench kiểm thử:

1. Người quản lý kiểm thử: quản lý quá trình chạy các thử nghiệm. Họ giữ vết của dữ liệu thử nghiệm, các kết quả mong đợi và chương trình để dàng kiểm thử. Các khung kiểm tự động hóa thử nghiệm như JUnit là ví dụ của các người quản lý thử nghiệm.
2. Máy sinh dữ liệu thử nghiệm: sinh các dữ liệu để thử nghiệm chương trình. Điều này có thể thực hiện bằng cách lựa chọn dữ liệu từ cơ sở dữ liệu hoặc sử dụng các mẫu để sinh ngẫu nhiên dữ liệu với khuôn dạng đúng đắn.
3. Hệ tiên đoán (Oracle): đưa ra các dự đoán về kết quả kiểm thử mong muốn. Các hệ tiên đoán có thể là phiên bản trước của chương trình hoặc hệ thống bản mẫu. Kiểm thử back-to-back , bao gồm việc thực hiện kiểm thử song song hệ tiên đoán và chương trình đó. Các khác biệt trong các đầu ra của chúng được làm nổi bật.
4. Hệ so sánh tập tin: so sánh các kết quả thử nghiệm chương trình với các kết quả thử nghiệm trước đó và báo cáo các khác biệt giữa chúng. Các hệ so sánh được sử dụng trong kiểm thử hồi quy (các kết quả thực hiện trong các phiên bản khác nhau được so sánh). Khi kiểm thử tự động được sử dụng, hệ so sánh có thể được gọi từ bên trong các kiểm thử đó.
5. Hệ sinh báo cáo: cung cấp các báo cáo để xác định và đưa ra các tiện lợi cho kết quả thử nghiệm.
6. Hệ phân tích động: thêm mã vào chương trình để đếm số lần mỗi câu lệnh đã được thực thi. Sau khi kiểm thử, một bản sơ thảo thực thi được sinh ra sẽ cho biết mỗi câu lệnh trong chương trình đã được thực hiện bao nhiêu lần.
7. Hệ mô phỏng (Simulator): Các loại hệ mô phỏng khác nhau có thể được cung cấp. Mục đích của các hệ mô phỏng là mô phỏng các máy khi chương trình được thực thi. Hệ mô phỏng giao diện người dùng là các chương trình điều khiển kịch bản mô phỏng nhiều tương tác đồng thời của người dùng. Sử dụng hệ mô phỏng cho I/O có nghĩa là bộ định thời gian của dãy giao dịch có thể được lặp đi lặp lại.



**Hình 3.17 Một workbench kiểm thử**

Khi sử dụng cho kiểm thử hệ thống lớn, các công cụ đó phải được định dạng và phù hợp với hệ thống cụ thể. Ví dụ:

1. Các công cụ mới có thể được thêm vào để kiểm thử các đặc trưng ứng dụng cụ thể, một vài công cụ hiện có có thể không cần đến.
2. Các kịch bản có thể được viết cho hệ mô phỏng giao diện người dùng và các mẫu đã xác định cho hệ sinh dữ liệu thử nghiệm. Các khuôn dạng báo cáo có thể cũng phải được xác định.
3. Các tập kết quả thử nghiệm mong muốn có thể phải chuẩn bị bằng tay nếu không một phiên bản chương trình nào trước đó có thể dùng được như một hệ tiên đoán.
4. Hệ so sánh tập tin mục đích đặc biệt có thể được viết bao gồm hiểu biết về cấu trúc của kết quả thử nghiệm trên tập tin.

Một lượng lớn thời gian và công sức thường cần để tạo nên một workbench thử nghiệm toàn diện. Do đó, các workbench hoàn chỉnh, như trên hình 3.17, chỉ được sử dụng khi phát triển các hệ thống lớn. Với các hệ thống đó, toàn bộ chi phí kiểm thử có thể lên tới 50% tổng giá trị phát triển, vì vậy, nó là hiệu quả để đầu tư cho công cụ chất lượng cao CASE hỗ trợ việc kiểm thử. Tuy nhiên, vì các loại hệ thống khác nhau yêu cầu sự hỗ trợ các loại kiểm thử khác nhau, các công cụ kiểm thử có thể không sẵn có để dùng. Rankin (Rankin, 2002) đã thảo luận một tình huống trong IBM và miêu tả thiết kế của hệ thống hỗ trợ kiểm thử, mà họ đã phát triển cho máy chủ kinh doanh điện tử.

Các điểm chính:

- Kiểm thử có thể chỉ ra sự hiện diện của các lỗi trong chương trình. Nó không thử chứng tỏ không còn lỗi trong chương trình.
- Kiểm thử thành phần là trách nhiệm của người phát triển thành phần. Một đội kiểm thử khác thường thực hiện kiểm thử hệ thống.
- Kiểm thử tích hợp là hoạt động kiểm thử hệ thống ban đầu khi bạn kiểm thử khiếm khuyết của các thành phần tích hợp. Kiểm thử phát hành liên quan đến kiểm thử của khách hàng và kiểm thử phát hành nên xác nhận hệ thống được phân phối có đầy đủ các yêu cầu.
- Khi kiểm thử hệ thống, bạn nên cố gắng “phá” hệ thống bằng cách sử dụng kinh nghiệm và các nguyên tắc để lựa chọn các kiểu thử nghiệm có hiệu quả để phát hiện khiếm khuyết trong hệ thống.
- Kiểm thử giao diện dùng để phát hiện các khiếm khuyết trong giao diện của các thành phần hỗn hợp. Các khiếm khuyết trong giao diện có thể nảy sinh bởi lỗi trong khi đọc các đặc tả chương trình, hiểu sai các đặc tả chương trình, các lỗi khác hoặc do thừa nhận bộ đếm thời gian không hợp lệ.
- Phân hoạch tương đương là một cách xác định các thử nghiệm. Nó phụ thuộc vào việc xác định các phân hoạch trong tập dữ liệu đầu vào và đầu ra, sự thực hiện chương trình với các giá trị từ các phân hoạch đó. Thông thường, các giá trị đó là giá trị tại giới hạn của phân hoạch.
- Kiểm thử cấu trúc dựa trên phân tích chương trình để phát hiện đường dẫn qua chương trình và sử dụng những phân tích để lựa chọn các thử nghiệm.
- Tự động hóa thử nghiệm làm giảm chi phí kiểm thử bằng cách hỗ trợ quá trình kiểm thử bằng cách công cụ phần mềm.



## CHƯƠNG 4: CÁC PHƯƠNG PHÁP KIỂM THỬ

*Chương này tập trung vào các kỹ thuật để tạo ra các trường hợp kiểm thử tốt và ít chi phí nhất, tất cả chúng phải thoả những mục tiêu kiểm thử ở chương trước. Nhắc lại các mục tiêu kiểm thử phần mềm là thiết kế các trường hợp kiểm thử có khả năng tìm kiếm nhiều lỗi nhất trong phần mềm và với ít thời gian và công sức nhất.*

Hiện tại phát triển rất nhiều phương thức thiết kế các trường hợp kiểm thử cho phần mềm. Những phương pháp này đều cung cấp một hướng kiểm thử có tính hệ thống. Qua trọng hơn nữa là chúng cung cấp một hệ thống có thể giúp đảm bảo sự hoàn chỉnh của các trường hợp kiểm thử phát hiện lỗi cho phần mềm.

Một sản phẩm đều có thể được kiểm thử theo 2 cách:

- Hiểu rõ một chức năng cụ thể của một hàm hay một module. Các trường hợp kiểm thử có thể xây dựng để kiểm thử tất cả các thao tác đó.
- Hiểu rõ cách hoạt động của một hàm/module hay sản phẩm. Các trường hợp kiểm thử có thể được xây dựng để đảm bảo tất cả các thành phần con khớp với nhau. Đó là tất cả các thao tác nội bộ của hàm dựa vào các mô tả và tất cả các thành phần nội bộ đã được kiểm thử một cách thoả đáng.

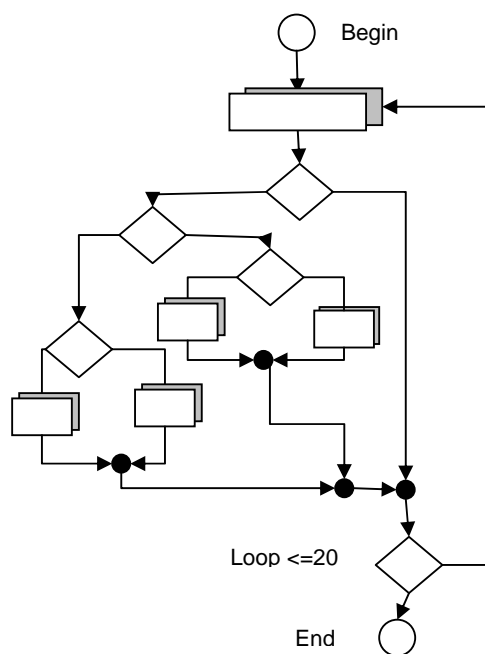
Cách tiếp cận đầu tiên được gọi là kiểm thử hộp đen ( black box testing ) và cách tiếp cận thứ hai là gọi là kiểm thử hộp trắng ( white box testing).

Khi đề cập đến kiểm thử phần mềm, black box testing còn được biết như là kiểm thử ở mức giao diện ( interface ). Mặc dù thật sự thì chúng được thiết kế để phát hiện lỗi. Black box testing còn được sử dụng để chứng minh khả năng hoạt động của hàm hay module chương trình và có thể cả một chương trình lớn: các thông số đầu vào được chấp nhận như mô tả của hàm, giá trị trả về cũng hoạt động tốt, đảm bảo các dữ liệu từ bên ngoài ví dụ như file dữ liệu được giữ/đảm bảo tính nguyên vẹn của dữ liệu khi thực thi hàm.

White box testing là kỹ thuật tập trung vào khảo sát chặt chẽ thủ tục một cách chi tiết. Tất cả những đường diễn tiến logic trong chương trình được kiểm tra bằng những trường hợp kiểm thử kiểm tra trên các tập điều kiện và cấu trúc lập cụ thể. kỹ thuật này sẽ kiểm tra trạng thái của chương trình tại rất nhiều điểm trong chương trình nhằm xác giá trị mong đợi tại các điểm này có khớp với giá trị thực tế hay không.

Với tất cả các mục tiêu kiểm định trên thì kỹ thuật white box testing có lẽ sẽ dẫn đến một chương trình chính xác tuyệt đối. Tất cả những gì chúng ta cần bây giờ là thiết kế tất cả các đường logic của chương trình và sau đó là cài đặt tất cả các trường hợp kiểm định có được. Tuy nhiên việc kiểm định một cách thấu đáo tất cả các trường hợp là một bài toán quá lớn và tốn rất nhiều chi phí. Chúng ta hãy xem xét ví dụ sau

### Hình 4.1. FlowChart



Bên trái là flowchart cho một chương trình đơn giản được viết bằng khoảng 100 dòng mã với một vòng lặp chính thực thi đoạn mã bên trong và lặp lại không quá 20 lần. Tuy nhiên khi tính toán cho thấy đối với chương trình này có đến khoảng  $10^{14}$  đường có thể được thực hiện.

Chúng ta làm tiếp một phép tính nhanh để thấy được chi phí dùng để kiểm thử đoạn chương trình nay một cách thấu đáo và chi tiết. Ta giả sử rằng để kiểm định một trường hợp cần chạy trung bình tốn một giây. Và chương trình kiểm thử sẽ được chạy 24 giờ một ngày và chạy suốt 365 ngày một năm. Vậy thì để chạy kiểm thử cho tất cả các trường hợp này cũng cần phải tốn khoản 3170 năm.

Do đó kiểm thử một cách thấu đáo là một việc bất khả thi cho những hệ thống lớn.

Mặc dù kỹ thuật này không thể hiện thực được trong thực tế với lượng tài nguyên có hạn, tuy nhiên với một số lượng có giới hạn các đường diễn tiến logic quan trọng có chọn lựa trước để kiểm thử. Phương pháp này có thể là rất khả thi

Ngoài ra các trường hợp kiểm thử còn có thể là sự kết hợp của cả hai kỹ thuật trên nhằm đạt được các mục tiêu của việc kiểm thử.

Và bây giờ chúng ta sẽ đi và chi tiết thảo luận về kỹ thuật kiểm thử hộp trắng

#### 4.1. Phương pháp white-box:

Là phương pháp kiểm nghiệm dựa vào cấu trúc/mã lệnh chương trình. Phương pháp white-box kiểm nghiệm một chương trình (một phần chương trình, hay một hệ thống, một phần của hệ thống) đáp ứng tốt tất cả các giá trị input bao gồm cả các giá trị không đúng hay không theo dự định của chương trình.

Phương pháp kiểm nghiệm white-box dựa trên:

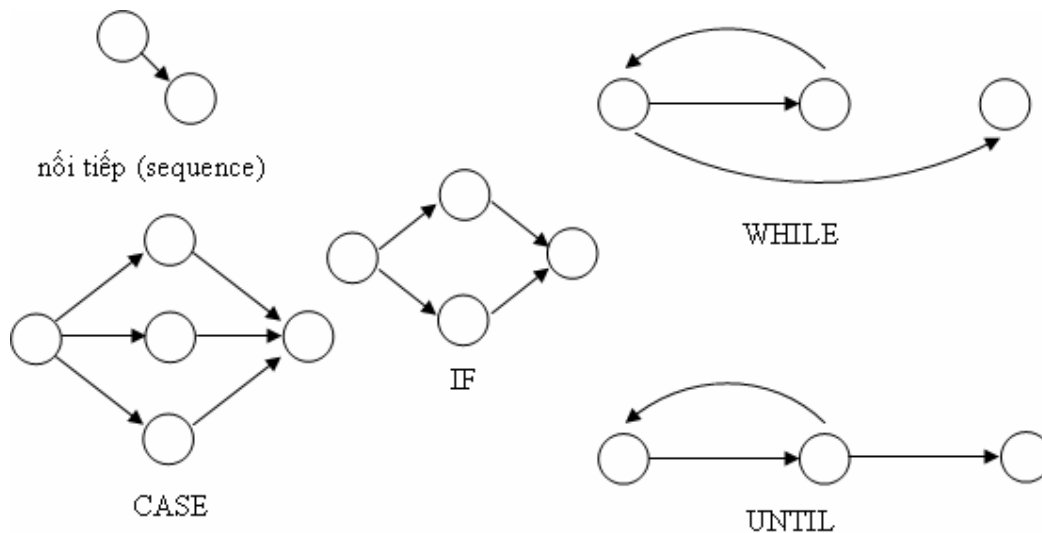
- Các câu lệnh (statement)
- Đường dẫn (path)
- Các điều kiện (condition)
- Vòng lặp (loop)
- Ngã rẽ (branch)

#### 4.1.1 Mô tả một số cấu trúc theo lược đồ:

Trong các phương pháp kiểm tra tính đúng đắn của chương trình, lược đồ được dùng để:

- Trừu tượng hóa cú pháp của mã lệnh.

- Làm khuôn mẫu cơ bản cho các nguyên tắc kiểm tra theo trường hợp.
- Kiểm tra tính đúng đắn trên toàn bộ lược đồ.
- 



#### 4.1.2 Kiểm tra theo câu lệnh: (Statement Testing)

Thiết kế quá trình kiểm tra sao cho mỗi câu lệnh của chương trình được thực hiện ít nhất một lần. Phương pháp kiểm tra này xuất phát từ ý tưởng:

- Từ phi một câu lệnh được thực hiện, nếu không ta không thể biết được có lỗi xảy ra trong câu lệnh đó hay không.
- Nhưng việc kiểm tra với một giá trị đầu vào không đảm bảo là sẽ đúng cho mọi trường hợp.

Ví dụ: Đoạn chương trình thực hiện tính:

$result = 0 + 1 + \dots + |value|,$

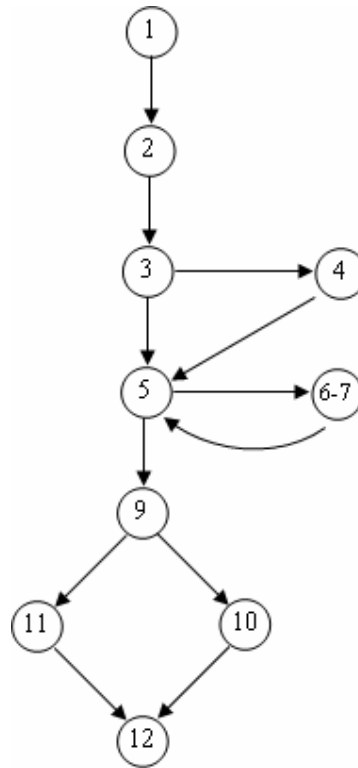
*nếu  $result \leq maxint$ , báo lỗi trong trường hợp ngược lại.*

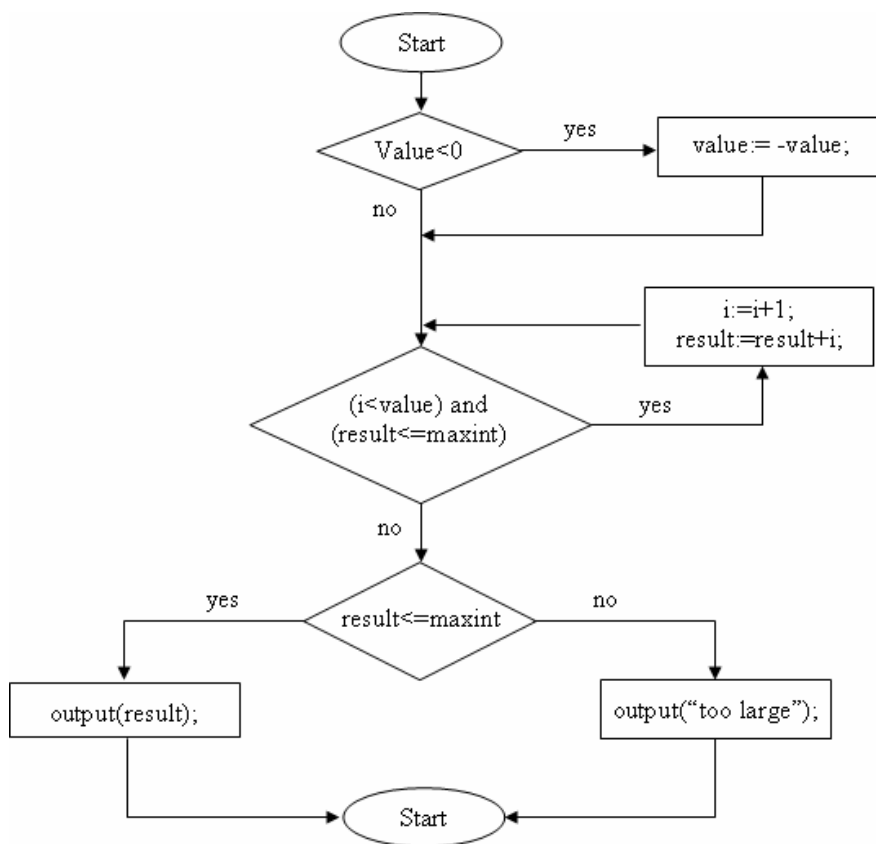
```

1    PROGRAM maxsum ( maxint, value : INT )
2        INT result := 0 ; i := 0 ;
3        IF value < 0
4            THEN value := - value ;
5        WHILE ( i < value ) AND ( result <= maxint )
6            DO    i := i + 1 ;
7                  result := result + i ;
8        OD;
9        IF result <= maxint

```

10            THEN OUTPUT ( result )  
11            ELSE OUTPUT ( “too large” )  
12    END.





Ví dụ với các bộ giá trị input:

maxint = 10, value = -1

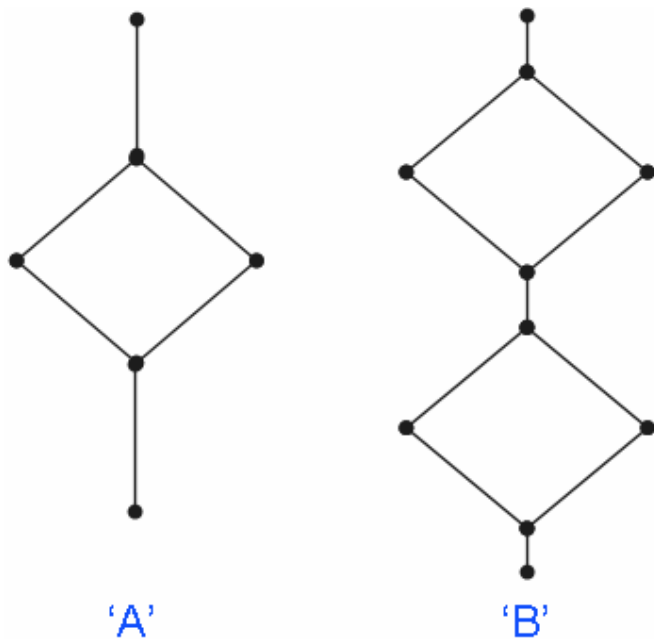
Hay

maxint = 0, value = -1

sẽ kiểm tra được toàn bộ các câu lệnh trong đoạn chương trình trên.

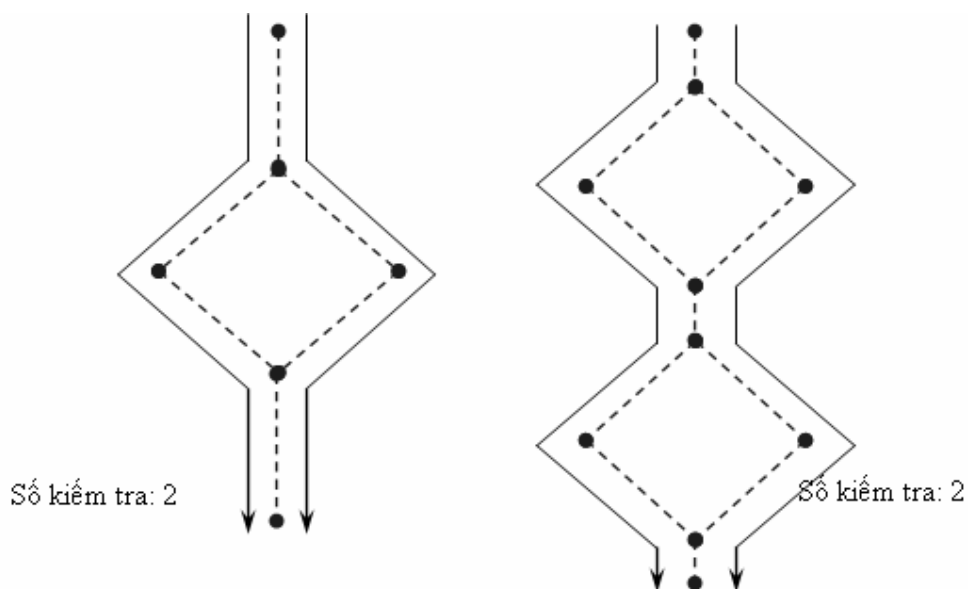
***Các vấn đề đối với phương pháp kiểm tra theo câu lệnh:***

Để đánh giá phương pháp này ta xem qua ví dụ sau:



*Hàm nào phức tạp hơn?*

Với câu hỏi đầu tiên “Lược đồ nào phức tạp hơn”, ta có câu trả lời là B. Và với câu hỏi tiếp theo “Lược đồ nào cần các bước kiểm tra nhiều hơn?” ta cũng trả lời là B.

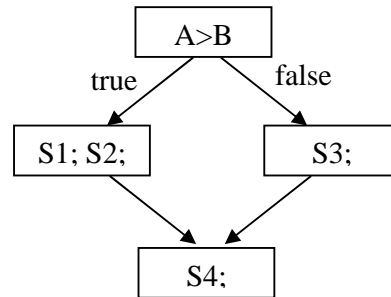


Tuy nhiên, ta thấy số lần kiểm tra tối thiểu để có thể kiểm tra toàn bộ các câu lệnh như trên cho cả 2 hàm đều là 2. Vì vậy, phương pháp này không tương ứng với sự phức tạp của mã lệnh.

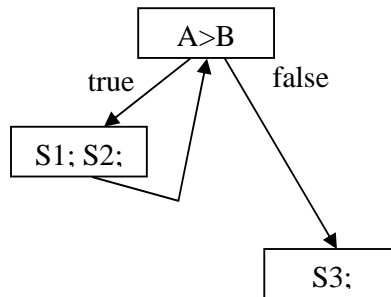
### 4.1.3 Kiểm tra theo đường dẫn: (Path Testing)

Là phương pháp kiểm tra bao trùm mọi đường dẫn của chương trình và cần kết hợp với lược đồ tiên trình.

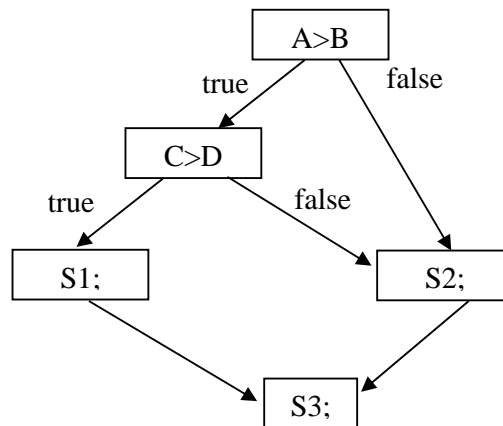
```
if ( A > B )
    S1;
    S2;
else
    S3;
S4;
```



```
while (A < B)
{
    S1;
    S2;
}
S3;
```



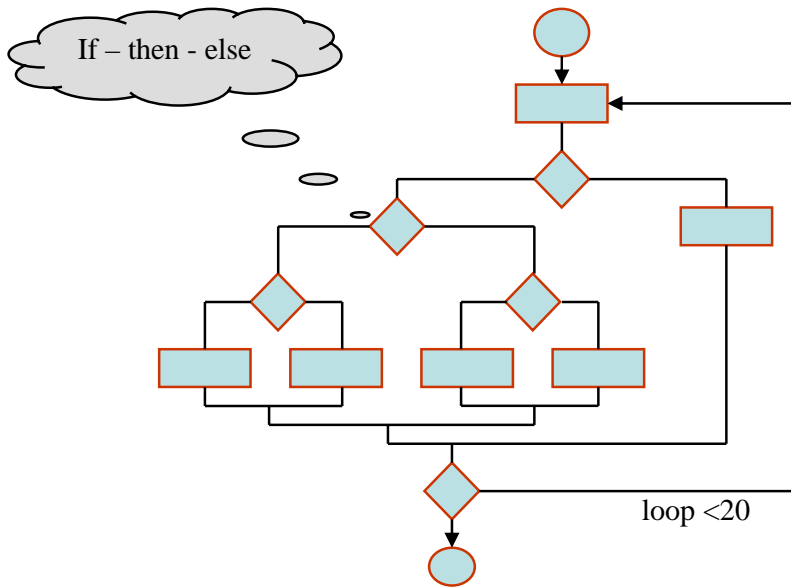
```
if (A < B && C < D)
    S1;
else
    S2;
S3;
```



#### Nhận xét:

Phương pháp kiểm tra theo đường dẫn phụ thuộc nhiều vào các biểu thức điều kiện. Tuy nhiên, có những trường hợp số lượng đường dẫn quá lớn (trường hợp vòng lặp). Vì vậy thường không phải là lựa chọn thực tế để tiến hành việc kiểm tra tính đúng đắn của chương trình.

Có khoảng  $5^{20} = 95.367.431.640.625$  đường dẫn



#### 4.1.4 Kiểm tra theo điều kiện: (Condition Testing)

Là phương pháp kiểm tra các biểu thức điều kiện trên 2 giá trị true và false.

Ta xét các ví dụ sau:

Ví dụ 1:

```
if (x > 0 && y > 0)
    x = 1;
else
    x = 2;
```

Các bộ kiểm tra  $\{ (x>0, y>0), (x \leq 0, y>0) \}$  sẽ kiểm tra toàn bộ các điều kiện.

Tuy nhiên: Không thỏa mãn với mọi giá trị input, cần kết hợp cả x và y để thực hiện bước kiểm tra.

Ví dụ 2:

```
while (x > 0 || y > 0)
{
    x--; y--;
    z += x*y;
}
```



Với bộ kiểm tra {  $(x > 0)$  } sẽ kiểm tra bao trùm được các điều kiện.

Tuy nhiên: Không kiểm tra được giá trị y.

Ví dụ 3:

```
if ( x != 0 )
    y = 5;
if ( z < 1 )
    z = z/x;
else
    z = 0;
```

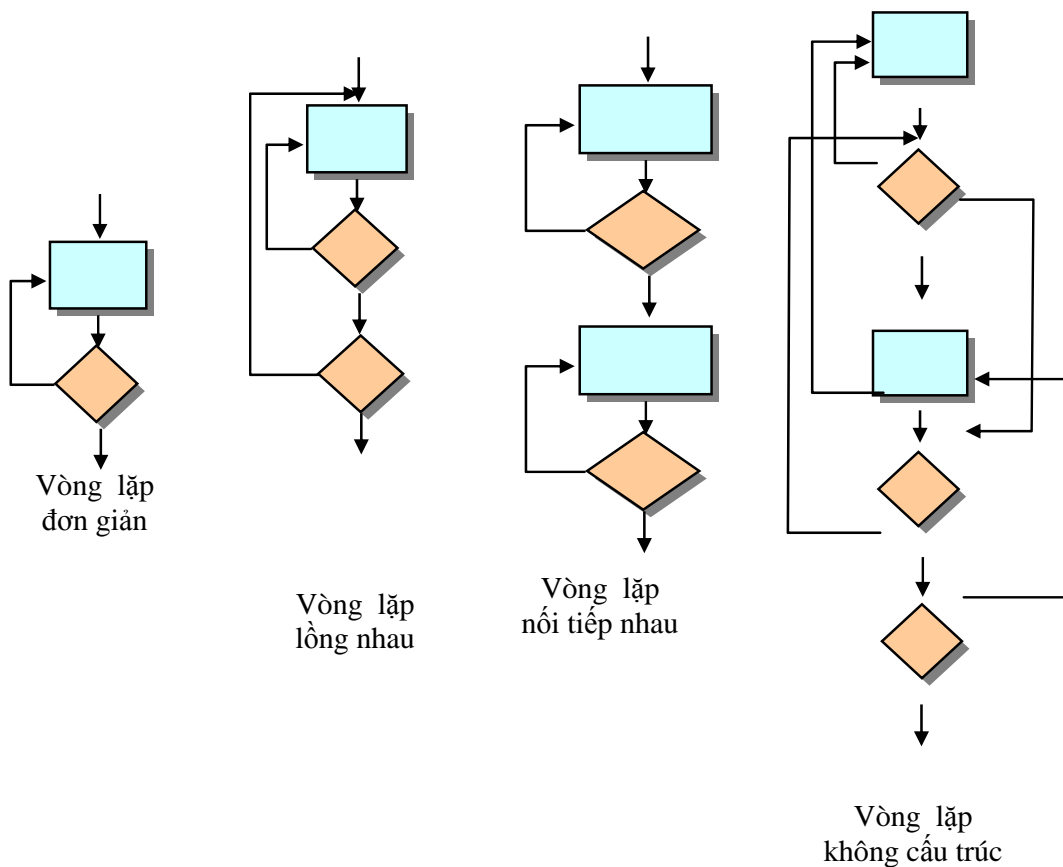
Với bộ kiểm tra {  $(x=0, z=1)$ ,  $(x=1, z=0)$  } sẽ kiểm tra bao trùm được các điều kiện.

Tuy nhiên: Không kiểm tra được trường hợp lỗi chia cho 0 (khi  $x=0$ ).

Nhận xét: Khi kiểm tra bằng phương pháp kiểm tra theo điều kiện cần xem xét kết hợp các điều kiện với nhau.

#### 4.1.5 Kiểm tra theo vòng lặp: (Loop Testing)

Là phương pháp tập trung vào tính hợp lệ của các cấu trúc vòng lặp.



**- Các bước cần kiểm tra cho vòng lặp đơn:**

- + Bỏ qua vòng lặp.
- + Lặp một lần.
- + Lặp hai lần.
- + Lặp m lần ( $m < n$ ).
- + Lặp  $(n-1)$ ,  $n$ ,  $(n+1)$  lần.

Trong đó  $n$  là số lần lặp tối đa của vòng lặp.

**- Các bước cần kiểm tra cho vòng lặp dạng lồng nhau:**

- + Khởi đầu với vòng lặp nằm bên trong nhất. Thiết lập các tham số lặp cho các vòng lặp bên ngoài về giá trị nhỏ nhất.
- + Kiểm tra với tham số  $\text{min}+1$ , 1 giá trị tiêu biểu,  $\text{max}-1$  và  $\text{max}$  cho vòng lặp bên trong nhất trong khi các tham số lặp của các vòng lặp bên ngoài là nhỏ nhất.
- + Tiếp tục tương tự với các vòng lặp liền ngoài tiếp theo cho đến khi tất cả vòng lặp bên ngoài được kiểm tra.

**- Các bước cần kiểm tra cho vòng lặp nối tiếp:**

- + Nếu các vòng lặp là độc lập với nhau thì kiểm tra như trường hợp các vòng lặp dạng đơn, nếu không thì kiểm tra như trường hợp các vòng lặp lồng nhau.

Ví dụ:

// LOOP TESTING EXAMPLE PROGRAM

import java.io.\*;

class LoopTestExampleApp {

// ----- FIELDS -----

public static BufferedReader keyboardInput =  
    new BufferedReader(new InputStreamReader(System.in));  
private static final int MINIMUM = 1;  
private static final int MAXIMUM = 10;

// ----- METHODS -----

```

/* Main method */
public static void main(String[] args) throws IOException {
    System.out.println("Input an integer value:");
    int input = new Integer(keyboardInput.readLine()).intValue();
    int numberOfIterations=0;
    for(int index=input;index >= MINIMUM && index <= MAXIMUM;index++) {
        numberOfIterations++;
    }
    // Output and end
    System.out.println("Number of iterations = " + numberOfIterations);
}
}

```

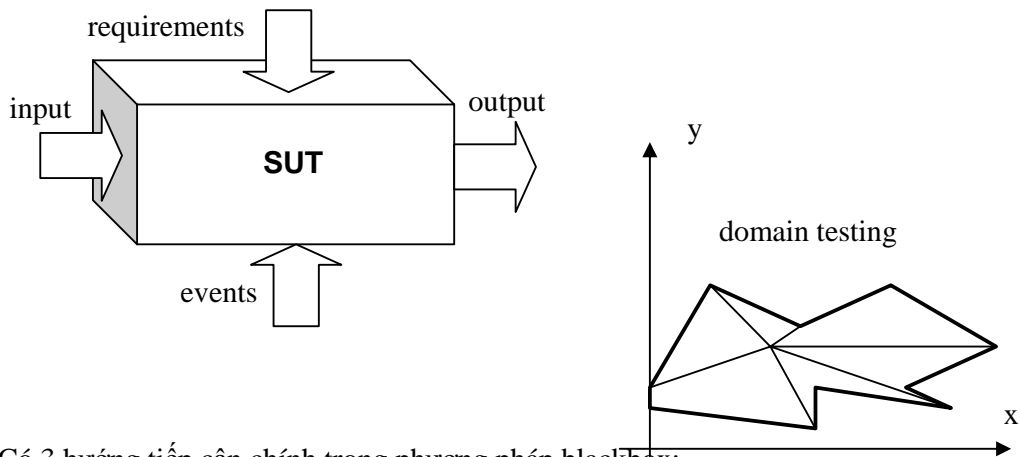
<b>Giá trị đầu vào</b>	<b>Kết quả (Số lần lặp)</b>
11	0 (bỏ qua vòng lặp)
10	1 (chạy 1 lần lặp)
9	2 (chạy 2 lần lặp)
5	6 (trường hợp chạy m lần lặp khi m<n)
2	9 (chạy N-1 lần lặp)
1	10 (chạy N lần lặp)
0	0 (bỏ qua vòng lặp)

#### 4.2. Phương pháp black-box:

Còn gọi là kiểm nghiệm chức năng. Việc kiểm nghiệm này được thực hiện mà không cần quan tâm đến các thiết kế và viết mã của chương trình. Kiểm nghiệm theo cách này chỉ quan tâm đến chức năng đã đề ra của chương trình. Vì vậy kiểm nghiệm loại này chỉ dựa vào bản mô tả chức năng của chương trình, xem chương trình có thực sự cung cấp đúng chức năng đã mô tả trong bản chức năng hay không mà thôi.

Kiểm nghiệm hộp đen dựa vào các định nghĩa về chức năng của chương trình. Các trường hợp thử nghiệm (test case) sẽ được tạo ra dựa nhiều vào bản mô tả chức năng chứ không phải dựa vào cấu trúc của chương trình. Gồm các phương pháp sau:

- Phân chia tương đương
- Phân tích giá trị biên
- Đồ thị Cause – Effect
- Kiểm tra hành vi (Behavioural testing)
- Kiểm thử ngẫu nhiên
- Ước lượng lỗi ....



Có 3 hướng tiếp cận chính trong phương pháp blackbox:

#### ***Phân tích miền vào/ra của chương trình:***

- Dẫn tới việc phân chia hợp lý miền Input/Output vào tập hợp con ‘interesting’.

#### ***Phân tích tính chất đáng chú ý của hộp đen:***

- Dẫn tới một loại ‘flow-graph-like’, có thể ứng dụng các kỹ thuật của hộp trắng (trên loại hộp đen này).

#### ***Heuristics:***

- Các kỹ thuật này giống với phân tích rủi ro, đầu vào ngẫu nhiên, kiểm thử ‘stress’.

#### **4.2.1. Phân chia tương đương:**

Phân chia (nếu có thể) tất cả các lớp đầu vào, như là:

- Có một số hạn chế về các lớp tương đương đầu vào.
- Chúng ta có thể chấp nhận một số lý do như:
  - Chương trình chạy để gom những tín hiệu đầu vào tương tự nhau vào trong cùng một lớp.
  - Test một giá trị đại diện của lớp.
  - Nếu giá trị đại diện bị lỗi thì các thành viên trong lớp đó cũng sẽ bị lỗi như thế.

#### 4.2.2. Lập kế hoạch:

Nhận dạng các lớp tương đương đầu vào:

- Dựa vào các điều kiện vào/ra trong đặc tính kỹ thuật/mô tả kỹ thuật.
- Cả hai lớp tương đương đầu vào: 'valid' và 'invalid'.
- Dựa vào heuristics và chuyên gia.
  - "input x in [1..10]" → classes:  $x < 1$ ,  $1 \leq x \leq 10$ ,  $x > 10$
  - "Loại liệt kê A, B, C" → classes: A, B, C, not{A,B,C}

Định nghĩa một/cặp của các trường hợp thử cho mỗi lớp.

- Kiểm thử các trường hợp thuộc lớp tương đương 'valid'
- Kiểm thử các trường hợp thuộc lớp tương đương 'invalid'

Ví dụ:

Kiểm một hàm tính giá trị tuyệt đối của một số integer. Các lớp tương đương:

Condition	Các lớp tương đương 'Valid'	Các lớp tương đương 'Invalid'
Số nhập vào	1	0, >1
Loại dữ liệu vào	integer	Non-integer
Abs	<0, >=0	

Kiểm các trường hợp:

$x = -10$ ,  $x = 100$   
 $x = \text{"XYZ"}$ ,  $x = -$   $x = 10 \ 20$

Ví dụ 2:

“ Một chương trình đọc 3 giá trị integer. Ba giá trị này được thể hiện như chiều dài của 3 cạnh một hình tam giác. Chương trình in một câu thông báo là tam giác thường (uligesidet), tam giác cân (ligebenet), hoặc tam giác đều (ligesidet).” [Myers]

+ Viết một tập các trường hợp để thử chương trình này.

*Các trường hợp test là:*

- Giá trị 3 cạnh có lệch nhau không?
- Giá trị 3 cạnh có bằng nhau không?
- Giá trị 3 cạnh tam giác cân?

- Ba hoán vị trước đó?
- Cạnh bằng 0?
- Cạnh có giá trị âm?
- Một cạnh bằng tổng của 2 cạnh kia?
- Ba hoán vị trước đó?
- Giá trị một cạnh lớn hơn tổng 2 cạnh kia?
- Ba hoán vị trước đó?
- Tất cả các cạnh bằng 0?
- Nhập vào giá trị không phải số nguyên (non-integer)?
- Số của các giá trị sai?
- Cho mỗi trường hợp thử: là giá trị đầu ra mong đợi?
- Kiểm tra cách chạy chương trình sau khi đầu ra hoàn chỉnh?

Ví dụ: Phân lớp tương đương

Kiểm tra một chương trình tính tổng giá trị đầu tiên của các số nguyên miễn là tổng này nhỏ hơn *maxint*. Mặt khác, khi có lỗi chương trình cần ghi lại, nếu giá trị âm, thì phải lấy giá trị tuyệt đối.

Dạng:

Nhập số nguyên **maxint** và **value**, giá trị **result** được tính là:

$$\text{Result} = \sum_{k=0}^{\text{Abs}(\text{value})} k \quad \text{nếu: } \leq \text{maxint}, \text{ ngoài ra thì sinh lỗi.}$$

Các lớp tương đương:

<i>Condition</i>	<i>Lớp tương đương ‘Valid’</i>	<i>Lớp tương đương ‘Invalid’</i>
Số nhập vào	2	<2, >2
Loại dữ liệu vào	Int int	Int no-int, no-int int
Abs(value)	Value<0, value≥0	
Maxint	Σk≤ maxint, Σk> maxint	

#### 4.2.3. Phân tích giá trị biên:

Dựa vào chuyên gia/Heuristics:

- Test điều kiện biên của các lớp thì có tác dụng nhiều hơn là đưa vào các giá trị trực tiếp như trên.
- Chọn các giá trị biên đầu vào để kiểm tra các lớp đầu vào thay vì thêm vào những giá trị tùy ý.
- Cũng chọn những giá trị đầu vào như thế để cho ra những giá trị biên đầu ra.
- Ví dụ về chiến lược mở rộng việc phân lớp:
  - Chọn một giá trị tùy ý cho mỗi lớp.
  - Chọn các giá trị chính xác ở biên trên và biên dưới của mỗi lớp.
  - Chọn các giá trị ngay lập tức ở dưới và trên mỗi biên (nếu có thể).

Ví dụ: Kiểm tra một hàm tính giá trị tuyệt đối của 1 số nguyên.

Các lớp tương đương 'valid' như sau:

Condition	Lớp tương đương 'Valid'	Lớp tương đương 'Invalid'
Abs	<0, >=0	

Các trường hợp thử:

Lớp $x < 0$ , Giá trị tùy ý:	$x = -10$
Lớp $x \geq 0$ , Giá trị tùy ý:	$x = 100$
Các lớp $x < 0, x \geq 0$ , Giá trị biên	$x = 0$
Các lớp $x < 0, x \geq 0$ , Giá trị dưới và trên	$x = -1, x = 1$

Ví dụ: Phân tích giá trị biên

Nhập vào số integer *maxint* và *value* tính toán giá trị *result* như sau:

$$\text{Result} = \sum_{k=0}^{\text{Value}} k \quad \text{nếu: } \leq \text{maxint}, \text{ ngoài ra thì sinh lỗi.}$$

Các lớp tương đương *valid*:

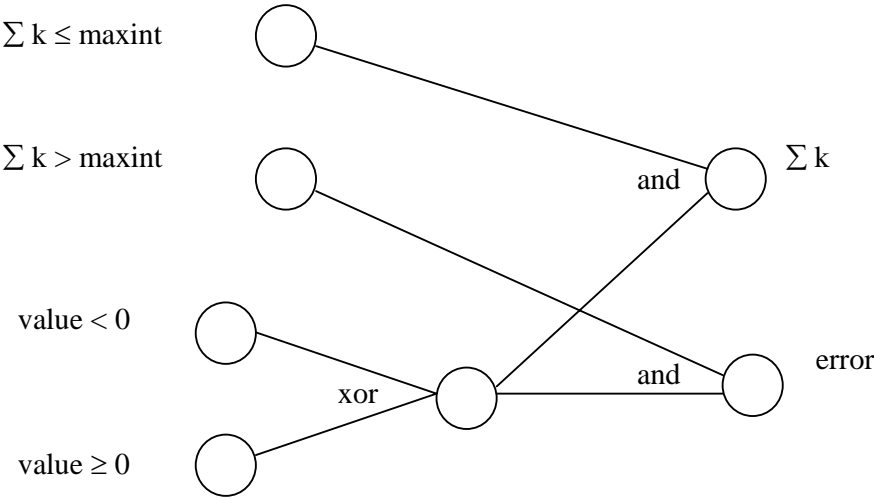
Condition	Lớp tương đương 'Valid'
Abs(value)	Value < 0, value ≥ 0
Maxint	Σk ≤ maxint, Σk > maxint

Chúng ta cần giá trị giữa  $\text{maxint} < 0$  và  $\text{maxint} \geq 0$ ?





	“XYZ”	10	Error
	100	9.1E4	error



<b>Causes</b>	$\sum k \leq maxint$	1	1	0	0
<b>inputs</b>	$\sum k > maxint$	0	0	1	1
	$value < 0$	1	0	1	0
	$value \geq 0$	0	1	0	1
<b>Effects</b>	$\sum k$	1	1	0	0
<b>outputs</b>	$error$	0	0	1	1

## CHƯƠNG 5: KIỂM THỬ TÍCH HỢP

*Người mới tập sự trong thế giới phần mềm có thể hỏi một câu hỏi có vẻ hợp lý khi mọi mô đun đã được kiểm thử đơn vị xong. Nếu tất cả chúng đã làm việc riêng biệt tốt thì sao các anh lại hoài nghi rằng chúng không làm việc khi ta gắn chúng lại với nhau? Vấn đề, dĩ nhiên, là ở chỗ “gắn chúng lại với nhau” – làm giao diện. Dữ liệu có thể bị mất qua giao diện; mô đun này có thể có sự bất cân, ảnh hưởng bất lợi sang mô đun khác; các chức năng con, khi tổ hợp, không thể tạo ra chức năng chính mong muốn; những sự không chính xác chấp nhận được ở từng mô đun có thể bị khuếch đại lên đến mức không chấp nhận nổi; các cấu trúc dữ liệu toàn cục có thể làm sinh ra vấn đề. Điều đáng buồn là danh sách này còn kéo dài mãi.*

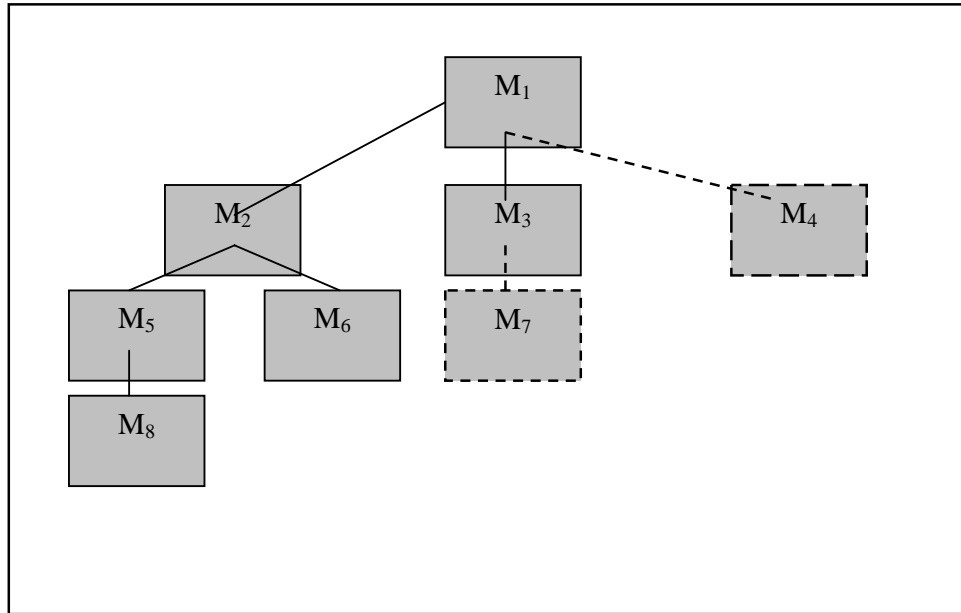
*Kiểm thử tích hợp là một kỹ thuật hệ thống để xây dựng cấu trúc chương trình trong khi đồng thời tiến hành các kiểm thử để phát hiện lỗi liên kết với việc giao tiếp. Mục đích lấy các mô đun đã kiểm thử đơn vị xong và xây dựng nên một cấu trúc chương trình được quy định bởi thiết kế.*

*Thường có một khuynh hướng cố gắng tích hợp không tăng dần; tức là xây dựng chương trình bằng cách dùng cách tiếp cận vụ nổ lớn “Big bang”. Tất cả các mô đun đều được tổ hợp trước. Toàn bộ chương trình được kiểm thử như một tổng thể. Và thường sẽ là một kết quả hỗn loạn! Gặp phải một tập hợp các lỗi. Việc sửa đổi thành khó khăn vì việc cô lập nguyên nhân bị phức tạp bởi việc trải rộng trên toàn chương trình. Khi những lỗi này đã được sửa chữa thì những lỗi mới lại xuất hiện và tiến trình này cứ tiếp diễn trong chu trình vô hạn.*

*Tích hợp tăng dần là ngược lại với cách tiếp cận vụ nổ lớn. Chương trình được xây dựng và kiểm thử trong từng đoạn nhỏ, nơi lỗi dễ được cô lập và sửa chữa hơn; giao diện có thể được kiểm thử đầy đủ hơn và cách tiếp cận hệ thống có thể được áp dụng. Trong những mục sau đây, một số chiến lược tích hợp tăng dần sẽ được thảo luận tới.*

### 5.1. Tích hợp trên xuống.

Tích hợp trên xuống là cách tiếp cận tăng dần tới việc xây dựng cấu trúc chương trình. Các mô đun được tích hợp bằng cách đi dần xuống qua cấp bậc điều khiển, bắt đầu với mô đun điều khiển chính (chương trình chính). Các mô đun phụ thuộc (và phụ thuộc cuối cùng) vào mô đun điều khiển chính sẽ được tổ hợp dần vào trong cấu trúc theo chiều sâu trước hoặc chiều rộng trước.



Hình 5.1. Tích hợp trên xuống

Tham khảo tới hình 5.1., việc tích hợp chiều sâu trước sẽ tích hợp tất cả các mô đun trên đường điều khiển chính của cấu trúc. Việc chọn một đường chính có tùy tiện và phụ thuộc vào các đặc trưng của ứng dụng. Chẳng hạn, chọn đường bên tay trái, các mô đun M1, M2, M5, sẽ được tích hợp trước. Tiếp đó M8 hay (nếu cần cho chức năng riêng của M2) M6 sẽ được tích hợp vào. Rồi, xây dựng tiếp các đường điều khiển ở giữa và bên phải. Việc tích hợp theo chiều rộng tổ hợp tất cả các mô đun trực tiếp phụ thuộc vào từng mức; đi xuyên qua cấu trúc ngang. Từ hình vẽ, các mô đun M2, M3, M4 (thay thế cho cuống S4) sẽ được tích hợp vào trước nhất. Các mức điều khiển tiếp M5, M6, .. theo sau.

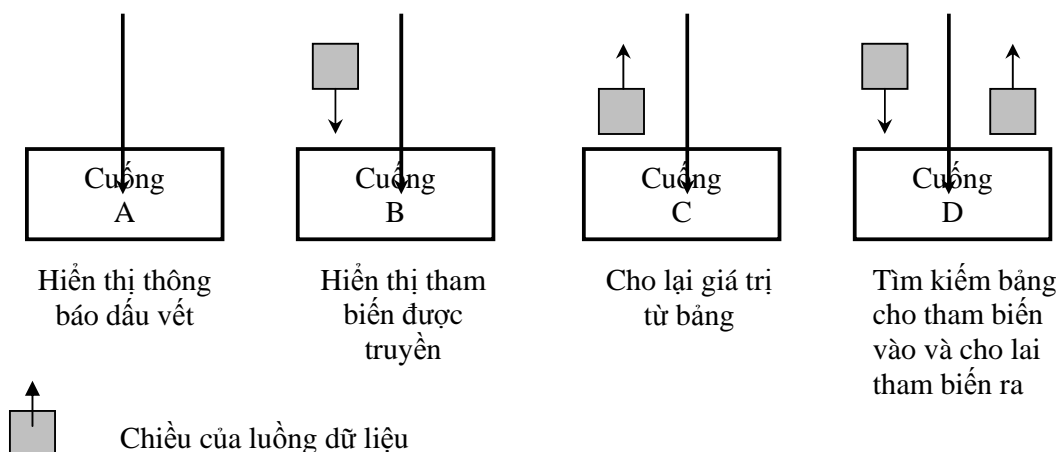
Tiến trình tích hợp được thực hiện trong một loạt năm bước:

1. Mô đun điều khiển chính được dùng như một khiên trình kiểm thử và các cuống được thế vào cho tất cả các mô đun phụ thuộc trực tiếp vào mô đun điều khiển chính.
2. Tùy theo các tiếp cận tích hợp được chọn lựa (nhưng theo chiều sâu hay theo chiều ngang trước) các cuống phụ thuộc được thay thế từng cái một mỗi lần bằng các mô đun thực tại.
3. Việc kiểm thử được tiến hành khi từng mô đun được tích hợp vào.
4. Khi hoàn thành từng tập các phép kiểm thử, cuống khác dễ được thay thế bằng các mô đun thực.
5. Kiểm thử hồi quy (tức là tiến hành tất cả hay một số các phép kiểm thử trước) có thể được tiến hành để đảm bảo từng những lỗi mới không bị đưa thêm vào.

Tiến trình này tiếp tục từ bước 2 tới khi toàn bộ cấu trúc chương trình đã được xây dựng xong. Hình 5.1 minh họa cho tiến trình này. Giả sử ta dùng cách tiếp cận độ sâu trước và một cấu trúc hoàn chỉnh bộ phận, cuống S7 là chuẩn bị được thay thế bởi mô đun M7. M7 bản thân nó có thể có các cuống sẽ bị thay thế bằng các mô đun tương ứng. Điều quan trọng phải chú ý là phải tiến hành các phép kiểm thử cho mỗi lần thay thế để kiểm chứng giao diện.

Chiến lược tích hợp trên xuống kiểm chứng việc điều khiển chính hay các điểm quyết định ngay từ đầu trong tiến trình kiểm thử. Trong một cấu trúc chương trình bố trí khéo, việc quyết định thường xuất hiện tại các mức trên trong cấp bậc và do đó được gặp phải trước. Nếu các vấn đề điều khiển chính quả là tồn tại thì việc nhận ra chúng là điều chủ chốt. Nếu việc tích hợp theo độ sâu được lựa chọn thì chức năng đầy đủ của phần mềm có thể được cài đặt và chứng minh. Chẳng hạn, ta hãy xét một cấu trúc giao tác cổ điển trong đó cần tới một loại các cái vào tương tác phức tạp rồi thu nhận và kiểm chứng chúng qua đường đi tới. Đường đi tới có thể được tích hợp theo cách trên xuống. Tất cả xử lý cái vào (đối với việc phát tán giao tác về sau) có thể được biểu diễn trước khi các phần tử khác của cấu trúc được tích hợp vào. Việc biểu diễn ban đầu về khả năng chức năng là cái xây dựng niềm tin cho cả người phát triển khách hàng.

Chiến lược phát triển trên xuống có vẻ như không phức tạp nhưng trong thực tế, các vấn đề logic có thể nảy sinh. Điều thông thường nhất của những vấn đề này xuất hiện khi việc xử lý tại mức thấp trong cấp bậc đòi hỏi việc kiểm thử tích hợp ở mức trên. Cuồng thay thế cho các mô đun cấp thấp vào lúc bắt đầu của kiểm thử trên xuống; do đó không có dữ liệu nào có nghĩa có thể chảy ngược lên trong cấu trúc chương trình. Người kiểm thử đứng trước hai chọn lựa: (1) để trễ nhiều việc kiểm thử tới khi cuồng được thay thế bằng mô đun thực thể, (2) xây dựng các cuồng thực hiện những chức năng giới hạn mô phỏng cho mô đun thực tại, và (3) tích hợp phần mềm từ đáy cấp bậc lên. Hình 5.2 minh hoạ cho các lớp cuồng điển hình, bố trí từ đơn giản nhất (cuồng A) tới phức tạp nhất (cuồng D).



Hình 5.2 Cuồng

Cách tiếp cận thứ nhất (để trễ kiểm thử cho đến khi cuồng được thay thế bởi mô đun thực tại) gây cho chúng ta bị mất điều khiển đối với tương ứng giữa kiểm thử đặc biệt và việc tổ hợp các mô đun đặc biệt. Điều này có thể dẫn tới những khó khăn trong việc xác định nguyên nhân lỗi và có khuynh hướng vi phạm bản chất bị ràng buộc cao độ của cách tiếp cận trên xuống. Các tiếp cận thứ hai thì được, nhưng có thể dẫn tới tổng phí khá lớn, vì cuồng trở thành ngày càng phức tạp hơn. Cách tiếp cận thứ ba, được gọi là kiểm thử dưới lên, được thảo luận trong mục sau:

## 5.2. Tích hợp dưới lên.

Kiểm thử tích hợp dưới lên, như tên nó đã hàm ý, bắt đầu xây dựng và kiểm thử với các mô đun nguyên tử (tức là các mô đun ở mức thấp nhất trong cấu trúc chương trình). Vì

các mô đun này được tích hợp từ dưới lên trên nên việc xử lý yêu cầu đối với các mô đun phụ thuộc của một mức nào đó bao giờ cũng có sẵn và nhu cầu về cuống bị dẹp bỏ.

Chiến lược tích hợp dưới lên có thể được thực hiện qua những bước sau:

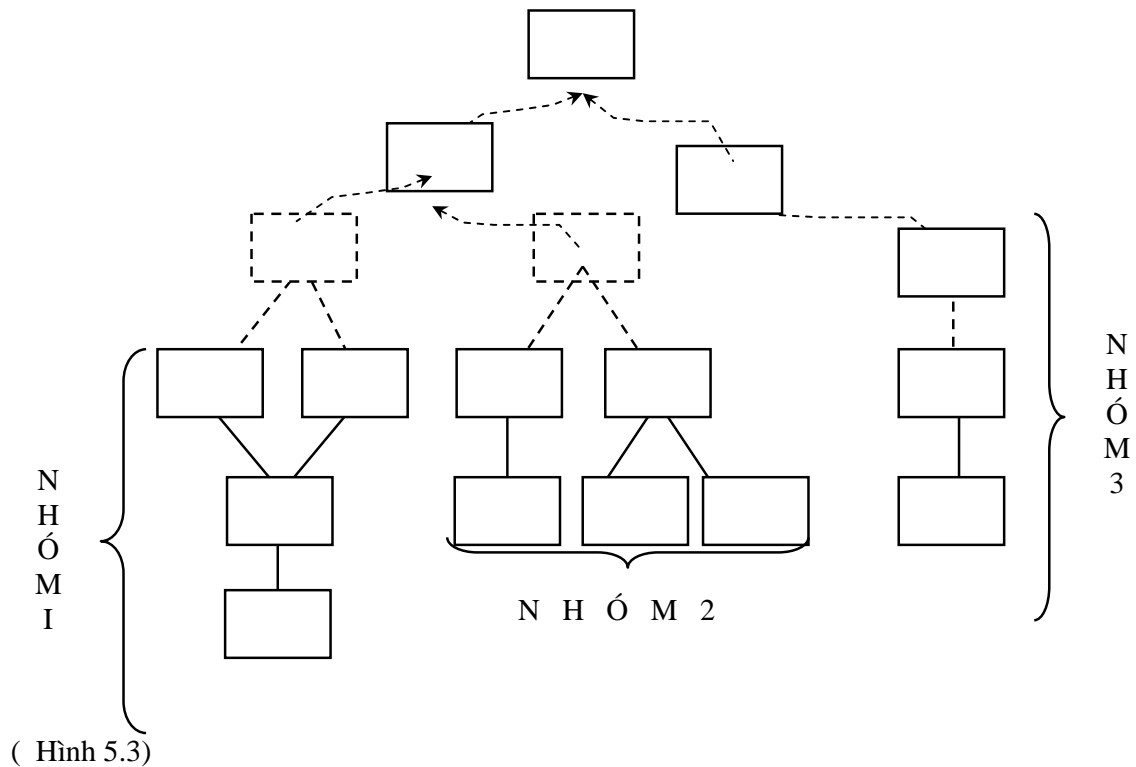
1. Các mô đun mức thấp được tổ hợp vào các chùm (đôi khi cũng còn được gọi là kiểu kiến trúc) thực hiện cho một chức năng con phần mềm đặc biệt.
2. Khiến trình (một chương trình điều khiển cho kiểm thử) được viết ra để phối hợp việc vào và trường hợp kiểm thử.
3. Kiểm thử chùm
4. Loại bỏ khiến trình và chùm được tổ hợp chuyển lên trong cấu trúc chương trình.

Việc tích hợp đi theo mẫu hình được minh hoạ trong hình 5.3. Các mô đun được tổ hợp để tạo nên các chùm 1, 2, 3. Từng chùm đều được kiểm thử bằng cách dùng một khiến trình (được vẽ trong khối sẫm màu). Các mô đun trong các chùm 1 và 2 phụ thuộc vào  $M_a$ . Các khiến trình D1 và D2 được loại bỏ và chùm được giao tiếp trực tiếp với  $M_a$ . Tương tự, khiến trình D3 cho chùm 3 loại bỏ trước khi tích hợp với mô đun  $M_b$ . Cả  $M_a$  và  $M_b$  cuối cùng sẽ được tích hợp với mô đun M, và cứ như thế. Các loại khiến trình khác nhau được minh hoạ trong hình 5.4

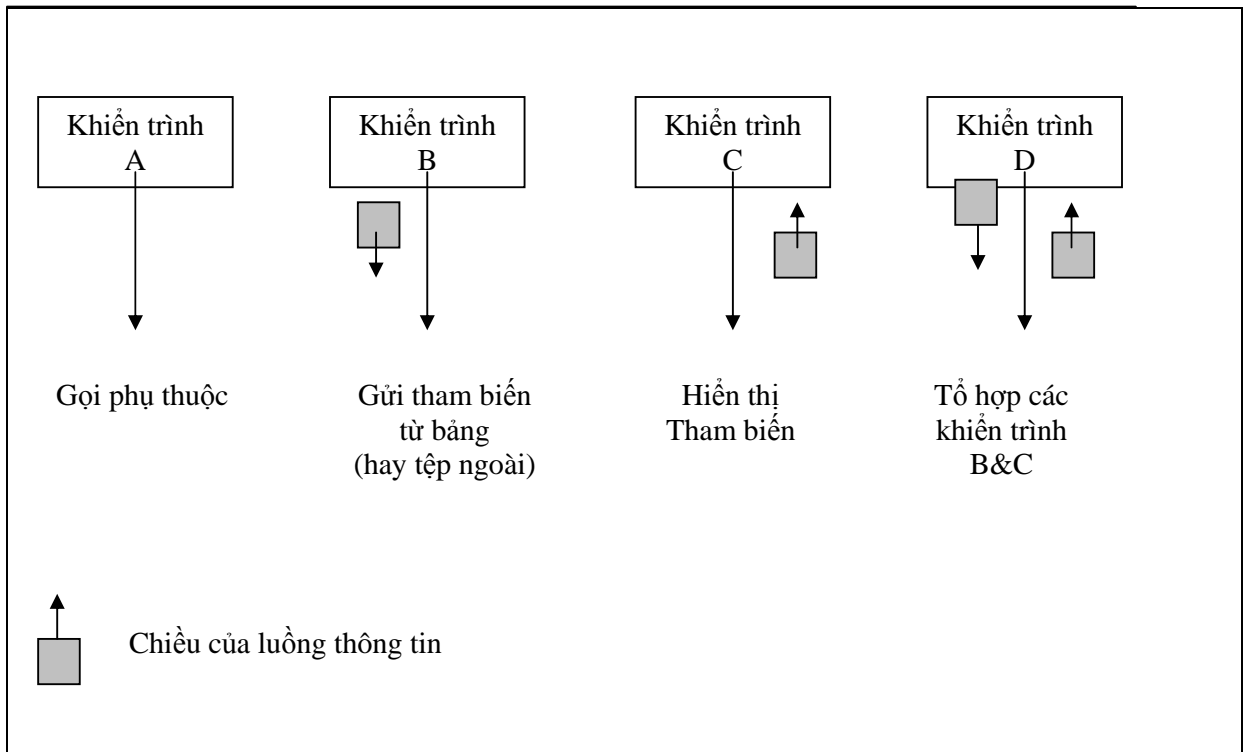
Khi việc tích hợp đi lên, nhu cầu về các khiến trình kiểm thử tách biệt ít dần. Trong thực tế, nếu hai mức đỉnh của cấu trúc chương trình được tích hợp theo kiểu trên xuống thì số các khiến trình có thể được giảm bớt khá nhiều và việc tích hợp các chùm được đơn giản hơn rất nhiều.

### **5.3. Kiểm thử nội quy.**

Mỗi một lần một module mới được thêm vào như là một phần của kiểm thử tích hợp, phần mềm thay đổi. Các đường dẫn dữ liệu mới được thiết lập, vào ra mới cũng xảy ra, và logic điều khiển mới được gọi ra. Những thay đổi này có thể sinh ra vấn đề đối với các hàm đã làm việc hoàn hảo trước đó. Trong một ngữ cảnh của một chiến lược kiểm thử phần mềm, kiểm thử hồi quy là việc thực hiện lại một vài tập hợp con các kiểm thử mà đã được quản lý để đảm bảo những thay đổi đó không sinh ra các hiệu ứng phụ không kiểm soát được.



Trong một ngữ cảnh rộng như vậy, kiểm thử thành công (bất cứ loại nào) kết quả là các tìm ra các lỗi cả các lỗi phải được làm đúng. Một khi phần mềm đã được làm đúng, một vài bộ phận của cấu hình phần mềm (chương trình, các tài liệu của nó, dữ liệu mà hỗ trợ nó) bị thay đổi. Kiểm thử hồi quy là hoạt động mà giúp đảm bảo rằng những thay đổi (trong toàn bộ quá trình kiểm thử hoặc là các lý do khác) không dẫn tới các cách thức không lường trước hoặc các lỗi phát sinh thêm.



Hình 5.4 Kiến trình

Kiểm thử hồi quy có thể thực hiện bằng thao tác bằng tay bằng cách thực hiện lại một tập hợp con của tất cả các trường hợp kiểm tra hoặc sử dụng các công cụ bắt lỗi tự động. Các công cụ bắt lỗi trở lại (capture - playback tools) cho phép người kỹ sư phần mềm có thể bắt được các trường hợp kiểm tra và kết quả cho một chuỗi lỗi và so sánh trở lại. Bộ kiểm thử hồi quy bao gồm ba lớp các trường hợp kiểm thử khác nhau:

- Một là biểu diễn ví dụ của các trường hợp kiểm thử mà nó sẽ thực hiện với tất cả các chức năng của phần mềm.
- Thêm vào các kiểm thử ứng với các chức năng của phần mềm giống như là giả định khi thay đổi.
- Các trường hợp kiểm thử ứng với các thành phần của phần mềm đã bị thay đổi.

Giống như tiến trình kiểm thử tích hợp, số các kiểm thử hồi quy có thể tăng lên khá lớn. Vì thế bộ kiểm thử hồi quy nên được thiết kế để bao gồm chỉ các trường hợp kiểm thử mà nhằm vào một hay nhiều các lớp của lỗi trong mỗi chức năng chương trình chính. Rõ ràng là không thực tế và không hiệu quả để thực hiện lại tất cả các kiểm thử cho mọi chức năng chương trình mỗi lần khi có sự thay đổi xảy ra.

#### 5.4. Gợi ý về việc kiểm thử tích hợp

Đã có nhiều thảo luận về ưu và nhược điểm tương đối của việc kiểm thử tích hợp trên xuống và dưới lên. Nói chung, ưu điểm của chiến lược này có khuynh hướng thành nhược điểm của chiến lược kia. Nhược điểm chính của cách tiếp cận trên xuống là cần tới các cuống và có thể có những khó khăn kiểm thử kèm theo liên kết với chúng. Vấn đề liên

quan tới các cuống có thể bù đắp lại bằng ưu điểm của việc kiểm thử các chức năng điều khiển chính sớm sửa. Nhược điểm chính của việc tích hợp dưới lên là ở chỗ “chương trình như một thực thể thì chưa tồn tại chừng nào mô đun cuối cùng chưa được thêm vào”. Nhược điểm này được làm dịu đi bằng thiết kế trường hợp kiểm thử sớm hơn và việc thiếu cuống.

Sự lựa chọn một chiến lược tích hợp phụ thuộc vào các đặc trưng phần mềm và đôi khi cả lịch biểu dự án. Nói chung, một cách tiếp cận tổ hợp (đôi khi còn được gọi là kiểm thử bánh kẹp thịt) dùng chiến lược trên xuống cho các mức trên của cấu trúc chương trình, đi đôi với chiến lược dưới lên cho các mức phụ thuộc, có thể là sự thỏa hiệp tốt nhất.

Khi việc kiểm thử tích hợp được tiến hành, người kiểm thử phải xác định mô đun gắng. Một mô đun gắng có duy nhất một hay nhiều đặc trưng sau: (1) đề cập tới nhiều yêu cầu phần mềm, (2) có mức điều khiển cao (nằm ở vị trí tương đối cao trong cấu trúc chương trình), (3) là phức tạp hay dễ sinh lỗi (độ phức tạp xoay vòng có thể được dùng làm một chỉ báo) hay (4) có yêu cầu độ hoàn thiện xác định. Các mô đun gắng nên được kiểm thử sớm nhất có thể được. Bên cạnh đó, kiểm thử hồi quy nên tập trung vào chức năng mô đun gắng.

### 5.5. Lập tài liệu về kiểm thử tích hợp

Kế hoạch tổng thể cho việc tích hợp phần mềm và một mô tả về các kiểm thử đặc biệt được ghi trong bản *Đặc tả kiểm thử*. Bản đặc tả kiểm thử này có thể bàn giao được trong tiến trình kỹ nghệ phần mềm và trở thành một phần của cấu hình phần mềm. Hình 17.11 trình bày dàn bài cho bản đặc tả kiểm thử có thể được dùng như một khuôn khổ cho tài liệu này

Hình 5.5 Dàn bài đặc tả kiểm thử

- I. Phạm vi kiểm thử
- II. Kế hoạch kiểm thử
  - A. Các giai đoạn và khối kiểm thử.
  - B. Lịch biểu
  - C. Tổng phí phần mềm
  - D. Môi trường và tài nguyên
- III. Thủ tục kiểm thử n (mô tả việc kiểm thử cho khối n)
  - A. Thứ tự tích hợp
    - 1. Mục đích
    - 2. Mô đun cần kiểm thử
  - B. Kiểm thử đơn vị cho các mô đun trong khối
    - 1. Mô tả kiểm thử cho mô đun
    - 2. Mô tả tổng phí phần mềm
    - 3. Kết quả dự kiến
  - C. Môi trường kiểm thử
    - 1. Công cụ hay kỹ thuật đặc biệt
    - 2. Mô tả tổng phí phần mềm



D. Dữ liệu trường hợp kiểm thử

E. Kết quả dự kiến thực tế

IV. Kết quả kiểm thử thực tế

V. Tham khảo

VI. Phụ lục

Phạm vi kiểm thử tóm tắt các đặc trưng chức năng, sự hoàn thiện và thiết kế bên trong riêng, cần phải được kiểm thử. Nỗ lực kiểm thử và cần gắn kèm, tiêu chuẩn để hoàn tất từng giai đoạn kiểm thử cần được mô tả, và các ràng buộc lịch biểu cần được làm tư liệu.

Phân kế hoạch kiểm thử mô tả chiến lược chung cho việc tích hợp. Việc kiểm thử được chia thành các giai đoạn và khối, đề cập tới các đặc trưng hành vi và chức năng riêng của phần mềm. Chẳng hạn, kiểm thử tích hợp cho một hệ thống CAD hướng đồ thị có thể được chia thành các giai đoạn kiểm thử sau:

- Giao diện người dùng (chọn chỉ lệnh; tạo ra việc vẽ; biểu diễn hiển thị; xử lý lỗi và biểu diễn lỗi)
- Thao tác và phân tích dữ liệu (tạo ra ký hiệu; tầm hướng; quay; tính các tính chất vật lý)
- Xử lý và sinh hiển thị (hiển thị hai chiều, ...)
- Quản trị cơ sở dữ liệu (thêm nhập; cập nhật; toàn vẹn; hiệu năng)

Mỗi một trong các giai đoạn và giai đoạn con (được ghi trong dấu ngoặc tròn) đều nêu ra phạm trù chức năng rộng bên trong phần mềm và nói chung có thể có liên quan tới một lĩnh vực riêng của cấu trúc chương trình. Do đó, các khối chương trình (nhóm các mô đun) được tạo ra để tương ứng với từng giai đoạn.

Các tiêu chuẩn sau đây và phép kiểm thử tương ứng được áp dụng cho tất cả các giai đoạn kiểm thử:

Tính thống nhất giao diện. Các giao diện bên trong và bên ngoài được kiểm thử khi từng mô đun (hay chùm) được tổ hợp vào trong cấu trúc.

Hợp lệ chức năng. Tiến hành các kiểm thử đã được thiết kế để phát hiện ra lỗi chức năng.

Nội dung thông tin. Tiến hành các kiểm thử đã được thiết kế để phát hiện ra lỗi liên kết với các cấu trúc dữ liệu cục bộ hay toàn cục được sử dụng.

Sự hoàn thiện. Tiến hành kiểm thử đã được thiết kế để kiểm chứng các cận hoàn thiện đã được thiết lập trong thiết kế phần mềm.

Những tiêu chuẩn này và các kiểm thử liên kết với chúng được thảo luận trong mục bản Đặc tả kiểm thử.

Lịch biểu để tích hợp, tổng phí phần mềm, và các chủ thể có liên quan cũng được thảo luận như một phần của mục kế hoạch kiểm thử. Ngày tháng bắt đầu và kết thúc cho từng giai đoạn được thiết lập và “cửa sổ có sẵn” cho các mô đun đã xong kiểm thử đơn vị cũng phải được xác định. Một mô tả tóm tắt về tổng phí phần mềm (cuồng và khiển trình) tập trung vào các đặc trưng có thể yêu cầu nỗ lực đặc biệt. Cuối cùng, cũng phải mô tả môi trường và tài nguyên kiểm thử. Các cấu hình phần cứng bất thường, các bộ mô phỏng ngoại lai, các công cụ kỹ thuật kiểm thử đặc biệt là một số trong nhiều chủ đề có thể được thảo luận trong mục này.

Một thủ tục kiểm thử chi tiết cần tới để hoàn thành bản kế hoạch kiểm thử sẽ được mô tả trong mục thủ tục kiểm thử. Tham khảo lại dàn bài bản Đặc tả kiểm thử mục II, trật tự của việc tích hợp và các phép kiểm tử tương ứng tại mỗi bước tích hợp phải được mô tả. Một danh sách tất cả các trường hợp kiểm thử (có thể để tham khảo về sau) và kết quả dự kiến cũng nên được đưa vào.

Lịch sử các kết quả kiểm thử thực tại, các vấn đề, hay các đặc thù được ghi lại trong bốn mục của bản Đặc tả kiểm thử. Thông tin có trong mục này có thể rất quan trọng cho việc bảo trì phần mềm. Các tham khảo thích hợp các phụ lục cũng được trình bày trong hai mục cuối.

Giống như tất cả các phần tử khác của cấu hình phần mềm định dạng căn bản Đặc tả kiểm thử có thể được tổ chức theo nhu cầu cục bộ của tổ chức phát triển phần mềm. Tuy nhiên điều quan trọng cần lưu ý là một chiến lược tích hợp có trong bản Kế hoạch kiểm thử, và các chi tiết kiểm thử được mô tả trong bản Đặc tả kiểm thử là thành phần bản chất và phải có.

## CHƯƠNG 6: KỸ NGHỆ ĐỘ TIN CẬY PHẦN MỀM

Mục tiêu của chương này là thảo luận kỹ thuật thẩm định và xác nhận tính hợp lệ được dùng trong việc phát triển hệ thống quan trọng. Khi bạn đọc chương này, bạn sẽ:

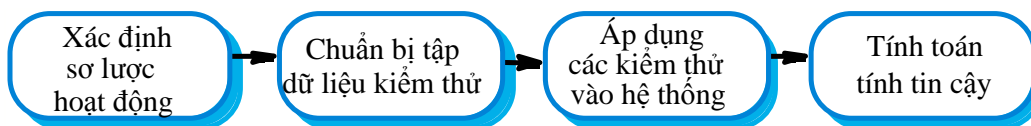
- Hiểu được phương pháp đo độ tin cậy của hệ thống và các mô hình phát triển độ tin cậy có thể được sử dụng để dự báo khi một mức độ yêu cầu độ tin cậy đạt được.
- Hiểu được các nguyên lý của các luận chứng tính an toàn và cách các nguyên lý này có thể được sử dụng cùng với phương pháp V & V trong việc đảm bảo tính an toàn của hệ thống.
- Hiểu được các vấn đề trong việc đảm bảo tính bảo mật của hệ thống.
- Làm quen với các trường hợp an toàn mà đưa ra các luận chứng và dấu hiệu của an toàn hệ thống.

### 6.1. Giới thiệu

Rõ ràng, việc thẩm định và xác nhận tính hợp lệ của hệ thống quan trọng đã rất phổ biến trong việc xác nhận tính hợp lệ của bất kỳ hệ thống nào. Quá trình V & V mô tả các đặc tả của hệ thống, và dịch vụ hệ thống và cách cư xử với các yêu cầu của người dùng. Tuy nhiên, với hệ thống quan trọng, tính tin cậy được đòi hỏi cao, hơn nữa kiểm thử và phân tích được yêu cầu để đưa ra bằng chứng chứng tỏ hệ thống đáng tin cậy. Có hai lý do tại sao bạn nên làm việc đó:

**1. Chi phí thất bại:** Chi phí và hậu quả của việc thất bại trong hệ thống quan trọng có khả năng cao hơn trong hệ thống không quan trọng. Bạn giảm nguy cơ thất bại của hệ thống bằng cách sử dụng việc thẩm định và xác nhận tính hợp lệ của hệ thống. Việc tìm và loại bỏ lỗi trước khi hệ thống được phân phối luôn luôn rẻ hơn chi phí do các sự cố của hệ thống.

**2. Xác nhận tính hợp lệ của các thuộc tính tin cậy:** Bạn có thể phải tạo ra một trường hợp bình thường để cho các khách hàng thấy hệ thống có chứa các yêu cầu về tính tin cậy (tính sẵn sàng, tính tin cậy, tính an toàn và tính bảo mật). Để đánh giá đặc tính tin cậy đòi hỏi sự hoạt động cụ thể V & V (được thảo luận ở phần sau trong chương này). Trong một vài trường hợp, người kiểm soát bên ngoài như người có thẩm quyền trong ngành hàng không quốc gia có thể phải chứng nhận rằng hệ thống là an toàn trước khi nó có thể cất cánh. Để đạt được chứng nhận này, bạn phải thiết kế và thực thi thủ tục đặc biệt V & V nhằm tập hợp chứng cứ về tính an toàn của hệ thống.



Hình 6.1 Quá trình đo tính tin cậy

Vì rất nhiều lý do, giá trị của V & V với hệ thống quan trọng luôn luôn cao hơn so với các loại hệ thống khác. Thông thường V & V chiếm hơn 50% tổng giá trị phát

triển của hệ thống phần mềm quan trọng. Tất nhiên, đây là giá đã được điều chỉnh, khi thất bại của hệ thống đã được ngăn ngừa. Ví dụ, năm 1996, một hệ thống phần mềm quan trọng trên tên lửa Ariane 5 bị hỏng và một vài vệ tinh đã bị phá hủy, gây thiệt hại hàng trăm triệu đôla. Sau đó, những người có trách nhiệm đã khám phá ra rằng sự thiếu hụt trong hệ thống V & V có phần nào trách nhiệm trong việc này.

Mặc dù, quá trình xác nhận tính hợp lệ của hệ thống quan trọng phần lớn tập trung vào việc xác nhận tính hợp lệ của hệ thống, các hoạt động liên quan nên kiểm tra để xác nhận quá trình phát triển hệ thống đã được thực hiện. Khi thảo luận trong chương 27 và 28, chất lượng hệ thống bị ảnh hưởng bởi chất lượng của quá trình được sử dụng để phát triển hệ thống. Tóm lại, quá trình tốt dẫn đến hệ thống tốt. Vì vậy, để cung cấp hệ thống tin cậy, bạn cần phải tin tưởng rằng quá trình phát triển hợp lý đã được tiến hành.

Đảm bảo quá trình là một phần của tiêu chuẩn ISO 9000 về quản lý chất lượng, được trình bày tóm tắt trong chương sau. Tài liệu yêu cầu chuẩn của các quá trình được sử dụng và liên kết các hoạt động để đảm bảo các quá trình đã được thực hiện. Thông thường yêu cầu có các hồ sơ quá trình để chứng nhận việc hoàn thành các hoạt động xử lý và kiểm tra chất lượng sản phẩm. Tiêu chuẩn ISO 9000 chỉ rõ đầu ra của quá trình cần đưa ra và người chịu trách nhiệm đưa ra nó. .

## **6.2. Xác nhận tính tin cậy**

Số lượng độ đo đã được phát triển để chỉ ra yêu cầu tin cậy của một hệ thống. Để xác nhận hệ thống có các yêu cầu nào, bạn phải đo độ tin cậy của hệ thống như bởi một người dùng hệ thống thông thường.

**Quá trình đo độ tin cậy của hệ thống được minh họa trong hình 6.1. Quá trình này gồm 4 bước:**

1. Đầu tiên, bạn nghiên cứu các hệ thống tồn tại của các kiểu như nhau để đưa ra mô tả sơ lược hoạt động. Mô tả sơ lược hoạt động nhận biết loại của đầu vào hệ thống và xác suất xuất hiện các đầu vào này trong trường hợp bình thường.
2. Sau đó, bạn thiết đặt tập các dữ liệu kiểm thử để phản ánh mô tả sơ lược hoạt động. Có nghĩa là bạn tạo ra dữ liệu kiểm thử với phân bố xác suất như nhau (như dữ liệu cho hệ thống mà bạn đã nghiên cứu). Thông thường, bạn sử dụng máy sinh dữ liệu kiểm thử để kiểm tra quá trình này.
3. Bạn kiểm thử hệ thống sử dụng các dữ liệu đã được sinh ở trên và đếm số lượng và các loại lỗi xảy ra. Số lần lỗi cũng được ghi nhận. Như thảo luận trong chương 9, các đơn vị thời gian bạn nên chọn phù hợp với độ đo tính tin cậy bạn dùng.
4. Cuối cùng, bạn tiến hành thống kê các lỗi quan trọng, bạn có thể tính toán độ tin cậy của phần mềm và đưa ra giá trị độ đo độ tin cậy.

Cách tiếp cận này được gọi là kiểm thử thống kê. Mục đích của kiểm thử thống kê là đánh giá độ tin cậy của hệ thống. Việc đánh giá cùng với kiểm thử sai sót, được

thảo luận trong chương 2, có cùng mục đích là tìm ra lỗi của hệ thống. Prowell et al.(1999) đưa ra một mô tả tốt của kiểm thử thống kê trong sách của ông (Kỹ nghệ phần mềm phòng sạch).

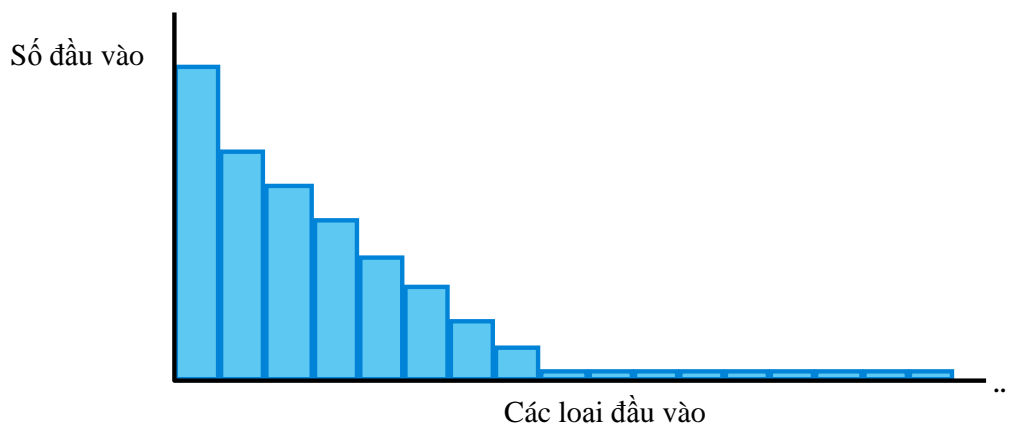
**Cách tiếp cận dựa trên độ đo tính tin cậy không dễ dàng áp dụng trong thực tế. Những khó khăn chủ yếu xuất hiện do:**

1. Không chắc chắn mô tả sơ lược hoạt động: Mô tả sơ lược hoạt động dựa trên kinh nghiệm, với các hệ thống khác có thể không phản ánh chính xác thực tế sử dụng của hệ thống.
2. Giá trị cao của sự sinh ra dữ liệu kiểm tra: có thể rất đắt để sinh một lượng lớn dữ liệu yêu cầu trong mô tả sơ lược hoạt động trừ khi quá trình có thể hoàn toàn tự động.
3. Thống kê không chắc chắn khi yêu cầu tính tin cậy cao được chỉ ra: Bạn phải sinh một số lượng thống kê quan trọng các sai sót để cho phép đo độ tin cậy chính xác. Khi phần mềm đã được xác thực tính tin cậy, một vài sai sót liên quan xuất hiện và nó khó khăn để sinh sai sót mới.

Phát triển mô tả sơ lược thao tác chính xác chắc chắn có thể với vài kiểu hệ thống, như hệ thống truyền thông có một mẫu tiêu chuẩn hóa được sử dụng. Tuy nhiên, với các loại hệ thống khác có rất nhiều người sử dụng khác nhau, mỗi người có một cách riêng khi sử dụng hệ thống. Như tôi đã thảo luận trong chương 3, những người dùng khác nhau có các ấn tượng khác nhau về độ tin cậy vì họ sử dụng hệ thống theo những cách khác nhau.

Từ đó, cách tốt nhất để sinh lượng lớn dữ liệu để đáp ứng yêu cầu đo độ tin cậy là sử dụng một hệ sinh dữ liệu kiểm thử mà có thể thiết đặt tự động sinh đầu vào phù hợp với mô tả sơ lược hoạt động. Tuy nhiên, nó thường không thể tự động sinh ra tất cả dữ liệu thử nghiệm cho các hệ thống tương tác bởi vì các đầu vào thường là câu trả lời tới đầu ra hệ thống. Tập dữ liệu cho các hệ thống đó phải được sinh ra bằng tay, do đó chi phí cao hơn. Ngay cả khi điều đó có thể hoàn toàn tự động, viết lệnh cho hệ sinh dữ liệu thử nghiệm có thể tiết kiệm nhiều thời gian.

Thống kê không chắc chắn là một vấn đề chung trong việc đo độ tin cậy của hệ thống. Để tạo nên một dự đoán chính xác độ tin cậy, bạn cần phải làm nhiều hơn là chỉ phát hiện ra một lỗi hệ thống đơn lẻ. Bạn phải sinh ra một lượng lớn dữ liệu phù hợp, thống kê số các lỗi để chắc chắn rằng độ tin cậy của bạn là chính xác. Điều này tốt nhất khi bạn tìm ra rất ít lỗi trong hệ thống, khó khăn là nó trở thành đo sự hiệu quả của kỹ thuật ít lỗi. Nếu tính tin cậy được xác định ở mức rất cao, nó thường không thực tế để sinh đủ lỗi hệ thống để kiểm tra các đặc tả đó.



Hình 6.2 Một sơ thảo hoạt động

### 6.2.1. Sơ thảo hoạt động

Sơ thảo hoạt động của phần mềm phản ánh cách phần mềm sẽ được sử dụng trong thực tế. Sơ thảo hoạt động gồm đặc tả các loại đầu vào và khả năng xuất hiện của chúng. Khi một hệ thống phần mềm mới thay thế một hệ thống bằng tay hoặc một hệ thống tự động hóa, điều đó là dễ thuyết phục để đánh giá các mẫu cách dùng có thể có của phần mềm mới. Nó nên phù hợp với cách sử dụng hiện có, với một vài sự thừa nhận được tạo bởi chức năng mới có thể có trong phần mềm mới. Ví dụ, một sơ thảo hoạt động có thể được xác định cho các hệ thống chuyển mạch viễn thông bởi vì các công ty viễn thông biết tất cả các mẫu cuộc gọi mà các hệ thống đó phải điều khiển.

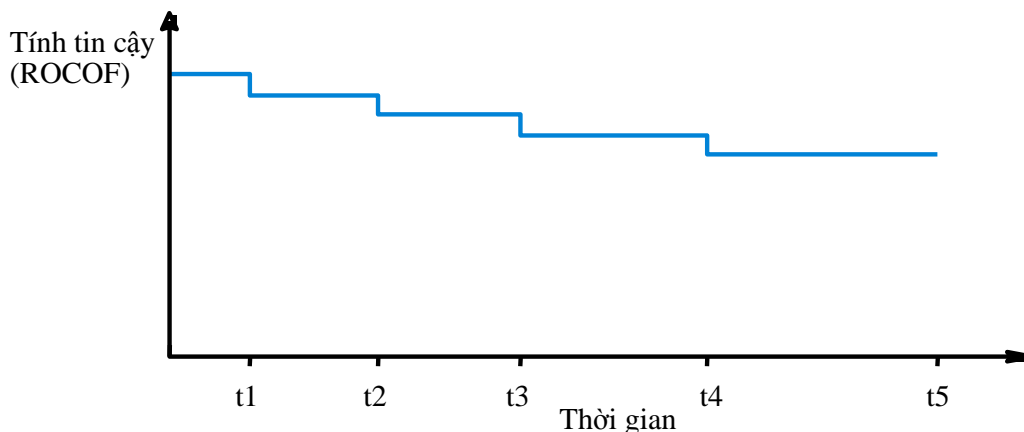
Thông thường, sơ thảo hoạt động như là các đầu vào có khả năng cao nhất được sinh ra và được phân vào một lượng nhỏ các lớp, như chỉ ra ở bên trái hình 6.2. Có một lượng lớn các lớp có các đầu vào có khả năng không xảy ra cao nhưng không phải là không thể xảy ra. Nó được chỉ ra ở bên phải hình 6.2. Dấu (...) có nghĩa là còn có rất nhiều đầu vào khác.

Musa (Musa, 1993; Musa, 1998) đề xuất các nguyên tắc để phát triển các sơ thảo hoạt động. Ông là một kỹ sư hệ thống viễn thông, và ông đã có thời gian dài làm việc tập hợp dữ liệu người dùng trong lĩnh vực này. Do đó, ông đã rút ra kết luận: Quá trình phát triển sơ thảo hoạt động tương đối dễ làm. Với một hệ thống yêu cầu công sức phát triển của khoảng 15 người làm việc trong một năm, sơ thảo hoạt động đã được phát triển trong khoảng 1 người/tháng. Trong các trường hợp khác, hệ sinh sơ thảo hoạt động cần nhiều thời gian hơn (2-3 người/năm), nhưng chi phí đã trải rộng ra hệ thống phát hành. Musa tính rằng công ty của ông (một công ty viễn thông) có ít nhất 10 nhóm để đầu tư vào việc phát triển sơ thảo hoạt động.

Tuy nhiên, khi một hệ thống phần mềm mới và tiên tiến được phát hành, ta rất khó đoán trước được nó sẽ được sử dụng như thế nào để đưa ra sơ thảo hoạt động chính xác. Rất nhiều người dùng với trình độ, kinh nghiệm và sự mong muốn khác nhau có thể sử dụng hệ thống mới. Nó không có cơ sở dữ liệu lịch sử cách dùng. Người dùng

có thể sử dụng hệ thống theo nhiều cách mà người phát triển hệ thống đã không dự đoán trước.

Vấn đề trở nên phức tạp hơn bởi vì các sơ thảo hoạt động có thể thay đổi lúc hệ thống đã được sử dụng. Khi người dùng nghiên cứu một hệ thống mới và trở nên tin tưởng hơn về nó, họ thường sử dụng nó theo những cách phức tạp. Do những khó khăn đó, Hamlet (Hamlet, 1992) cho rằng nó không có khả năng để phát triển một sơ thảo hoạt động tin cậy. Nếu bạn không chắc chắn rằng sơ thảo hoạt động của bạn là chính xác, thì bạn có thể không tin tưởng về sự chính xác của độ đo tính tin cậy của bạn.



Hình 23.3 Mô hình chức năng của quá trình gia tăng tính tin cậy với bước tăng bằng nhau

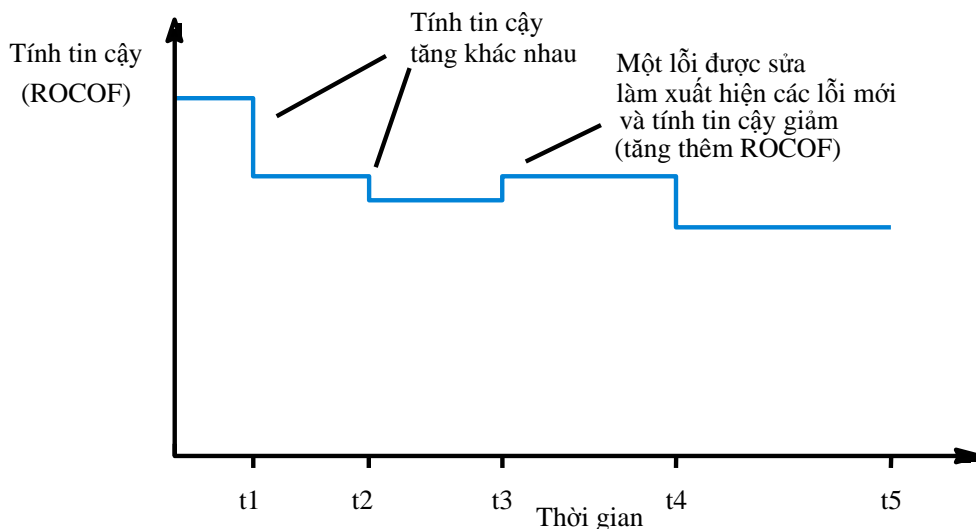
### 6.2.2. Dự đoán tính tin cậy

Trong lúc thẩm định phần mềm, người quản lý phải phân công quá trình kiểm thử hệ thống. Vì quá trình kiểm thử phần mềm rất tốn kém, nên nó sẽ được dừng ngay khi có thể và không “kiểm thử quá” hệ thống. Kiểm thử có thể dừng khi mức độ yêu cầu tính tin cậy của hệ thống đã được thực hiện. Tất nhiên, thỉnh thoảng, các dự đoán tính tin cậy có thể cho thấy mức độ yêu cầu tính tin cậy của hệ thống sẽ không bao giờ được thực hiện. Trong trường hợp đó, người quản lý phải đưa ra quyết định khó khăn: viết lại các phần của phần mềm hoặc sửa lại mô tả hệ thống.

Mô hình quá trình gia tăng tính tin cậy là một mô hình mà tính tin cậy của hệ thống thay đổi quá giờ trong thời gian quá trình kiểm thử. Khi các lỗi hệ thống được phát hiện, các khiếm khuyết cơ sở dẫn đến các lỗi đó đã được sửa chữa, vì vậy tính tin cậy của hệ thống có thể được cải thiện trong lúc kiểm thử và gỡ lỗi hệ thống. Để dự đoán tính tin cậy, mô hình quá trình gia tăng tính tin cậy nhận thức phải hiểu là mô hình toán học. Tôi không đi vào các mức cụ thể của vấn đề này, đơn giản chỉ thảo luận các nguyên tắc của quá trình gia tăng tính tin cậy.

Có nhiều mô hình quá trình gia tăng tính tin cậy đã được bắt nguồn từ kinh nghiệm trong các lĩnh vực ứng dụng khác nhau. Như Kan (Kan, 2003) đã phát biểu rằng: hầu hết các mô hình đó theo luật số mũ, với tính tin cậy tăng nhanh khi các khiếm khuyết

được phát hiện và loại bỏ (hình 6.5). Sau đó, sự tăng thêm nhỏ dần đi và tiến tới trạng thái ổn định khi rất ít khiếm khuyết được phát hiện và loại bỏ trong lần kiểm thử cuối cùng.



Hình 6.4 Mô hình chức năng quá trình gia tăng tích tin cậy với bước tăng ngẫu nhiên

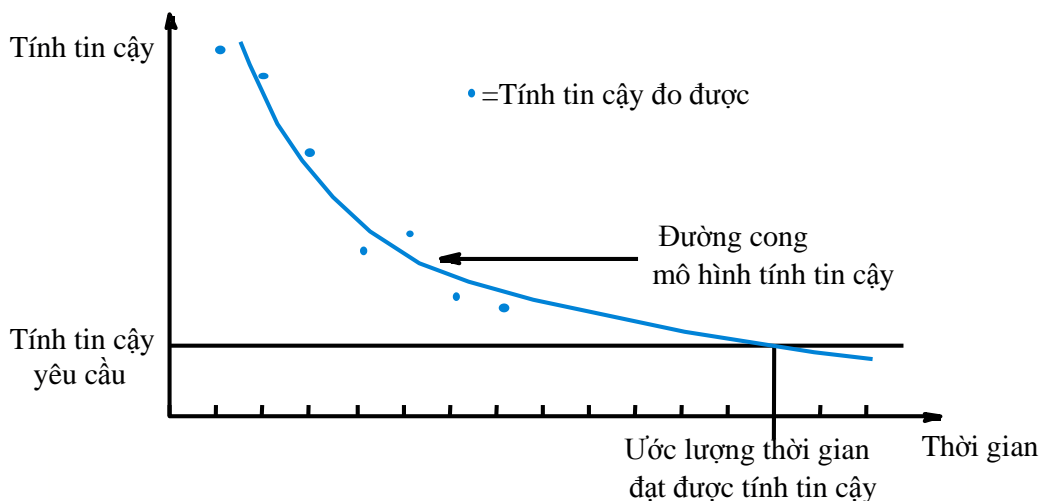
Mô hình đơn giản nhất minh họa khái niệm gia tăng tích tin cậy là mô hình bước chức năng (Jelinski và Moranda, 1972). Tích tin cậy tăng liên tiếp mỗi khi một lỗi (hoặc một tập lỗi) được phát hiện và sửa chữa (hình 6.3) và một phiên bản mới của phần mềm được tạo ra. Mô hình này giả sử rằng sự sửa chữa phần mềm luôn được thực hiện chính xác vì vậy số lỗi và khiếm khuyết liên hợp của phần mềm giảm trong mỗi phiên bản mới của hệ thống. Khi sự sửa chữa được tạo ra, tỷ lệ xuất hiện lỗi của phần mềm (ROCOF) có thể giảm, như trên hình 6.3. Chú ý các chu kỳ thời gian trên trục hoành phản ánh thời gian giữa các lần phát hành hệ thống để kiểm thử, vì vậy nó thường có chiều dài không bằng nhau.

Tuy nhiên, trong thực tế, các lỗi phần mềm không lúc nào cũng được sửa trong lúc gỡ lỗi, và khi bạn thay đổi một chương trình, thỉnh thoảng bạn đưa các lỗi mới vào chương trình đó. Khả năng xuất hiện các lỗi đó có thể cao hơn khả năng xuất hiện các lỗi đã được sửa chữa. Do đó, thỉnh thoảng tích tin cậy của hệ thống có thể trở nên tồi hơn trong phiên bản mới.

Mô hình quá trình gia tăng tích tin cậy đơn giản bước bằng nhau cũng giả sử tất cả các lỗi đóng góp như nhau vào tích tin cậy của hệ thống, và mỗi lỗi được sửa chữa đóng góp một lượng như nhau vào việc gia tăng tích tin cậy. Tuy nhiên, không phải tất cả các lỗi có khả năng xảy ra như nhau. Sửa chữa các lỗi phổ biến đóng góp vào việc gia tăng tích tin cậy nhiều hơn là sửa chữa các lỗi chỉ thỉnh thoảng xảy ra. Có lẽ bạn cũng cho rằng dễ dàng tìm kiếm các lỗi có khả năng xảy ra trong quá trình kiểm thử, vì thế tích tin cậy có thể tăng nhiều hơn khi các lỗi ít có khả năng xảy ra được phát hiện.



Cuối cùng, các mô hình như đề xuất của Littlewood và Verrall đưa ra các vấn đề bằng cách đưa một thành phần ngẫu nhiên vào quá trình gia tăng tính tin cậy nhằm cải thiện tác động của một sửa chữa trong phần mềm. Do đó, mỗi sửa chữa không dẫn đến cùng một lượng tăng tính tin cậy bằng nhau trong phần mềm, các biến đổi phụ thuộc vào tính ngẫu nhiên (hình 6.4).



Hình 6.5 Dự đoán tính tin cậy

Mô hình của Littlewood và Verrall cho phép phủ nhận quá trình gia tăng tính tin cậy khi sự sửa chữa đưa vào nhiều lỗi hơn. Đó cũng là mô hình khi một lỗi được sửa chữa, tính tin cậy được cải thiện trung bình bởi mỗi sửa chữa giảm. Lí do là hầu hết các lỗi có khả năng xảy ra có thể đã được phát hiện sớm trong quá trình kiểm thử. Sửa chữa các lỗi đó đóng góp phần lớn vào quá trình gia tăng tính tin cậy.

Các mô hình ở trên là các mô hình rời rạc phản ánh quá trình gia tăng tính tin cậy. Khi một phiên bản mới của phần mềm đã được sửa lỗi được đưa đi kiểm thử, nó nên có tỷ lệ lỗi xuất hiện thấp hơn phiên bản trước. Tuy nhiên, để dự đoán tính tin cậy sẽ đạt được sau khi thực hiện kiểm thử, chúng ta cần mô hình toán học liên tục. Nhiều mô hình này nhận được từ các lĩnh vực ứng dụng khác nhau, đã được đề xuất và so sánh.

Đơn giản, bạn có thể dự đoán tính tin cậy bằng cách kết hợp dữ liệu đo tính tin cậy và mô hình nhận biết tính tin cậy. Sau đó, bạn ngoại suy mô hình đó với các mức yêu cầu tính tin cậy và quan sát khi một mức yêu cầu tính tin cậy sẽ đạt được (hình 6.5). Do đó, kiểm thử và gỡ lỗi phải được thực hiện liên tục cho đến khi thỏa mãn các yêu cầu.

Dự đoán tính tin cậy của hệ thống từ mô hình quá trình gia tăng tính tin cậy có hai lợi ích chính:

1. *Lập kế hoạch kiểm thử*: đưa ra lịch kiểm thử hiện tại, bạn có thể dự đoán khi nào quá trình kiểm thử được hoàn thành. Nếu điều đó kết thúc sau ngày dự kiến phát

hành hệ thống, thì bạn phải triển khai bổ sung tài nguyên cho việc kiểm thử và gỡ lỗi để tăng nhanh tỷ lệ phát triển tính tin cậy.

2. *Sự đàm phán khách hàng*: Đôi khi mô hình tính tin cậy cho thấy sự tăng lên của tính tin cậy rất chậm và sự thiếu cân xứng của các cố gắng kiểm thử được yêu cầu với lợi ích đạt được tương đối ít. Nó có thể đáng giá để đàm phán lại các yêu cầu về tính tin cậy với khách hàng. Một sự lựa chọn khác, mô hình đó dự đoán tính các yêu cầu về tính tin cậy có thể sẽ không bao giờ đạt được. Trong trường hợp đó, bạn sẽ phải đàm phán lại với khách hàng về các yêu cầu về tính tin cậy của hệ thống.

Ở đây, tôi đã đơn giản hóa mô hình quá trình gia tăng tính tin cậy nhằm đem lại cho bạn những hiểu biết cơ bản về khái niệm này. Nếu bạn muốn sử dụng những mô hình này, bạn phải hiểu biết các kiến thức toán học bên dưới những mô hình này và các vấn đề thực tế của chúng. Littlewood và Musa đã viết các vấn đề bao quát của các mô hình gia tăng tính tin cậy và Kan viết một cuốn sách tóm tắt rất hay về vấn đề này. Nhiều tác giả khác đã mô tả các kinh nghiệm thực tế của họ khi sử dụng các mô hình quá trình gia tăng tính tin cậy.

### 6.3. Đảm bảo tính an toàn

Các quá trình đảm bảo tính an toàn và thẩm định tính tin cậy có mục tiêu khác nhau. Bạn có thể xác định số lượng tính tin cậy bằng cách sử dụng một vài độ đo, do đó đo được tính tin cậy của hệ thống. Với các giới hạn của quá trình đo, bạn biết các mức yêu cầu của tính tin cậy có thể đạt được hay không. Tuy nhiên, tính an toàn không thể xác định đầy đủ ý nghĩa theo số lượng, và do đó không thể được đo khi kiểm thử hệ thống.

Vì vậy, đảm bảo tính an toàn liên quan tới việc chứng minh mức độ tin cậy của hệ thống, nó có thể thay đổi từ rất thấp đến rất cao. Đây là một chủ đề với các chuyên gia phán đoán dựa trên các dấu hiệu của hệ thống, môi trường và các quá trình phát triển hệ thống. Trong nhiều trường hợp, sự tin cậy này phần nào dựa trên kinh nghiệm tổ chức phát triển hệ thống. Nếu trước đó một công ty đã phát triển nhiều hệ thống điều khiển mà đã hoạt động an toàn, thì đó là điều hợp lý để cho rằng họ sẽ tiếp tục phát triển các hệ thống an toàn.

Tuy nhiên, sự đánh giá phải đảm bảo bởi những chứng cứ rõ ràng từ thiết kế hệ thống, các kết quả của hệ thống V & V, và các quá trình phát triển hệ thống đã được sử dụng. Với một số hệ thống, các chứng cứ rõ ràng được thu thập trong một hộp an toàn (xem phần 6.4), nó cho phép một người điều chỉnh bên ngoài đi đến kết luận sự tin tưởng của người phát triển về tính an toàn của hệ thống được chứng minh là đúng.

Các quá trình V & V với các hệ thống quan trọng an toàn là phổ biến với các quá trình có thể so sánh được của bất kỳ hệ thống nào với các yêu cầu tính tin cậy cao. Đó phải là quá trình kiểm thử bao quát để phát hiện các khiếm khuyết có thể xảy ra, và tại những chỗ thích hợp, kiểm thử thống kê có thể được sử dụng để đánh giá tính tin cậy của hệ thống. Tuy nhiên, bởi vì tỷ lệ lỗi cực thấp được yêu cầu trong nhiều hệ thống quan trọng an toàn, kiểm thử thống kê không thể luôn cung cấp sự đánh giá về

số lượng tính an toàn của hệ thống. Các thử nghiệm cung cấp một vài bằng chứng, mà đã được sử dụng cùng với các bằng chứng khác như các kết quả của sự xem xét lại và kiểm tra tính , để đưa ra kết luận về tính an toàn của hệ thống.

Sự xem xét lại bao quát là cần thiết trong lúc quá trình phát triển hướng tính an toàn để trưng bày phần mềm tới những người mà sẽ xem xét nó từ nhiều khung nhìn khác nhau. Pernas đề xuất 5 loại xem xét lại mà nên được ủy thác với hệ thống quan trọng an toàn.

1. xem xét lại chính xác chức năng mong đợi.
2. xem xét lại cấu trúc có thể duy trì được, và có thể hiểu được.
3. xem xét lại để kiểm tra lại thiết kế thuật toán và cấu trúc dữ liệu là thích hợp với hành vi xác định.
4. xem xét lại tính chắc chắn của thiết kế mã, thuật toán và cấu trúc dữ liệu.
5. xem xét lại sự đầy đủ của các trường hợp kiểm thử.

Một sự thừa nhận làm cơ sở của hoạt động về tính an toàn hệ thống là nhiều thiếu sót của hệ thống có thể dẫn tới những rủi ro về tính an toàn quan trọng là ít hơn đáng kể so với tổng số thiếu sót có thể tồn tại trong hệ thống đó. Đảm bảo tính an toàn có thể tập trung vào những lỗi có tiềm năng gây rủi ro. Nếu nó có thể được chứng minh rằng những lỗi đó không thể xuất hiện, hoặc nếu những lỗi đó xuất hiện, sự rủi ro kết hợp sẽ không đưa đến một tai nạn, thì hệ thống là an toàn. Đây là những luận chứng cơ bản về tính an toàn mà tôi thảo luận trong phần tới.

### 6.3.1. Những luận chứng về tính an toàn

Việc chứng minh một chương đúng đắn, như thảo luận trong chương trước, đã được đề ra như một kỹ thuật thẩm định phần mềm khoảng hơn 30 năm trước. Việc chứng minh một chương trình bình thường chắc chắn có thể được xây dựng cho các hệ thống nhỏ. Tuy nhiên, những khó khăn thực tế của việc chứng minh rằng một hệ thống đáp ứng đầy đủ các đặc tả nó là quá lớn vài tổ chức xem xét việc chứng minh đúng đắn trở thành một chi phí. Tuy nhiên, với một số ứng dụng quan trọng, nó có thể kinh tế để phát triển việc chứng minh tính đúng đắn nhằm tăng sự tin tưởng rằng hệ thống đáp ứng các yêu cầu về tính an toàn và tính bảo mật. Đây là trường hợp đặc biệt khi chức năng tính an toàn quan trọng có thể cô lập trong một hệ thống con nhỏ mà có thể được xác định chính xác.

Mặc dù, nó có thể không mang lại lợi nhuận để phát triển việc chứng minh tính đúng đắn cho hầu hết các hệ thống, thỉnh thoảng nó có thể thực hiện được để phát triển những luận chứng đơn giản về tính an toàn để chứng minh chương trình đáp ứng các yêu cầu về tính an toàn. Với một luận chứng về tính an toàn, nó có thể không cần thiết để chứng minh các chức năng của chương trình được xác định. Nó chỉ cần thiết để chứng minh rằng sự thực thi của chương trình không thể dẫn tới một trạng thái không an toàn.

Hầu hết các kỹ thuật hiệu quả để chứng minh tính an toàn của hệ thống là chứng minh bằng phản chứng. Bạn bắt đầu với giả thiết rằng một trạng thái không an toàn đã được xác định bằng phân tích rủi ro hệ thống, có thể được đi đến khi chương trình thực thi. Bạn viết một thuộc tính để xác định đó là trạng thái không an toàn. Sau đó, một cách có hệ thống, bạn phân tích mã chương trình và chỉ ra, với tất cả các đường dẫn chương trình dẫn tới trạng thái đó, điều kiện kết thúc của các đường dẫn đó mâu thuẫn với thuộc tính trạng thái không an toàn. Nếu có trường hợp đó, giả thiết ban đầu của trạng thái không an toàn là không đúng. Nếu bạn lặp lại điều đó với tất cả định danh rủi ro, thì phần mềm là an toàn.

- Liều lượng Insulin được phân phối là một hàm của mức độ đường trong máu, liều lượng Insulin phân phối lần trước và thời gian phân phối liều thuốc trước.

```
currentDose = computeInsulin();

// Tính an toàn kiểm tra và điều chỉnh currentDose nếu cần thiết

// Câu lệnh if-1

if (previousDose == 0)
{
    if (currentDose > 16)
        currentDose = 16;
}
else
    if (currentDose > (previousDose * 2))
        currentDose = previousDose * 2;

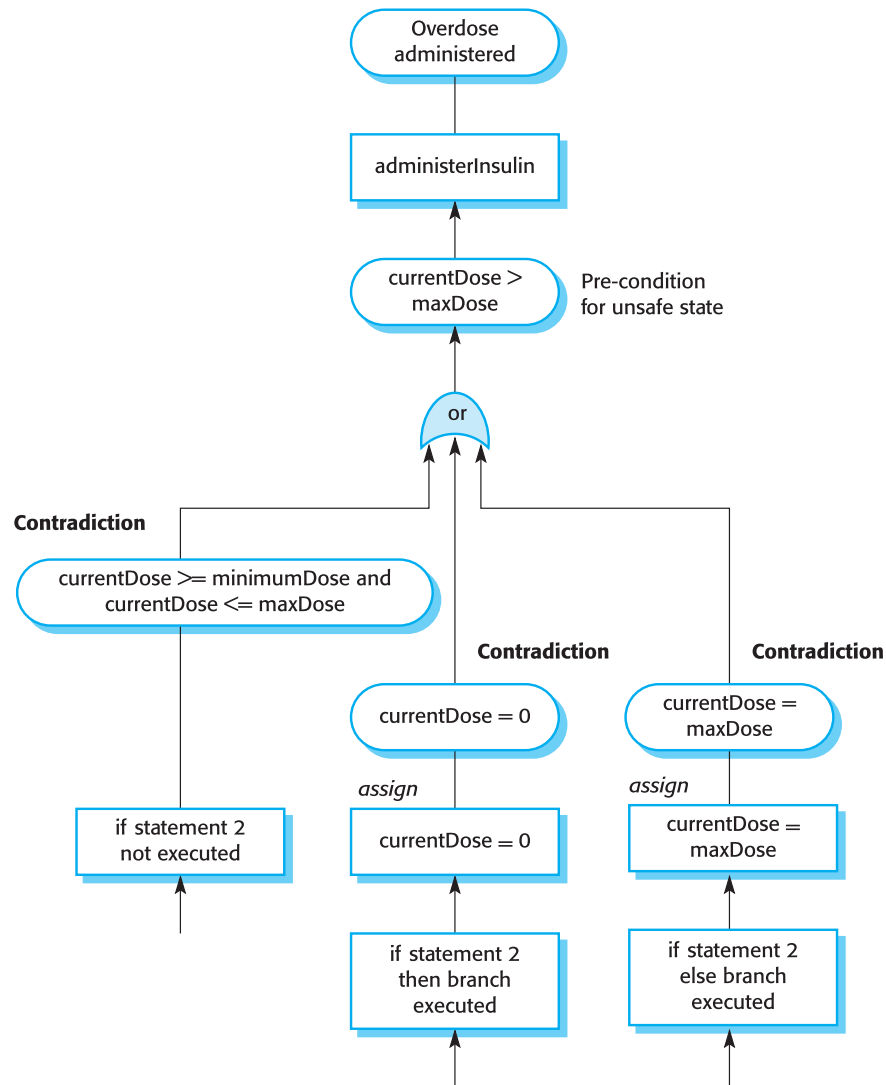
// Câu lệnh if-2

if (currentDose < minimumDose)
    currentDose = 2;
else if (currentDose > maxDose)
    currentDose = maxDose;
administerInsulin(currentDose);
```

Hình 6.6 Mã phân phối Insulin

Một ví dụ, xem xét mã trên hình 6.6, nó có thể là một phần thực thi của hệ thống phân phối insulin. Phát triển một luận chứng cho mã này bao gồm việc chứng minh liều lượng thuốc được quản lý không bao giờ nhiều hơn mức lớn nhất đã được lập cho mỗi bệnh nhân. Do đó, không cần thiết để chứng minh rằng hệ thống phân phối đúng liều lượng thuốc, mà chỉ đơn thuần là nó không bao giờ phân phối quá liều lượng cho bệnh nhân.

Để xây dựng những luận chứng về tính an toàn, bạn xác định tiên điều kiện cho trạng thái không an toàn, trong trường hợp đó như là  $\text{currentDose} > \text{maxDose}$ . Sau đó, bạn chứng minh rằng tất cả các đường dẫn chương trình đưa đến sự mâu thuẫn của điều khẳng định tính không an toàn đó. Nếu đó là một trường hợp, điều kiện không an toàn không thể là đúng. Do đó, hệ thống là an toàn. Bạn có thể cấu trúc và đưa ra những luận chứng về tính an toàn bằng đồ thị như trên hình 6.7.



Hình 6.7 Các luận chứng tính an toàn dựa trên sự mô tả những mâu thuẫn

Những luận chứng về tính an toàn, như được chỉ ra trên hình 6.7, là ngắn hơn nhiều so với việc thẩm tra hệ thống theo đúng trình tự. Đầu tiên, bạn xác định tất cả các đường dẫn có thể mà đưa tới trạng thái không an toàn tiềm năng. Bạn làm việc về phía sau từ trạng thái không an toàn và xem xét kết quả cuối cùng tới tất cả các biến trạng thái trên mỗi đường dẫn dẫn tới nó. Bạn có thể bỏ qua các tính toán trước đó (như câu lệnh if thứ nhất trong hình 6.7) trong các luận chứng tính an toàn. Trong ví dụ này, tất cả những gì bạn cần liên quan tới là một tập các giá trị có thể xảy ra của  $\text{currentDose}$  ngay lập tức trước khi phương thức  $\text{administerInsulin}$  được thực thi.

Trong các luận chứng về tính an toàn chỉ ra trên hình 6.7, có ba đường dẫn chương trình có thể dẫn tới việc gọi tới phương thức `administerInsulin`. Chúng tôi muốn chứng minh rằng lượng insulin phân phối không bao giờ vượt quá `maxDose`. Tất cả các đường dẫn chương trình có thể dẫn tới phương thức `administerInsulin` đã được xem xét:

1. Không có nhánh nào của câu lệnh `if` thứ hai được thực thi. Nó chỉ có thể xảy ra nếu một trong hai điều sau xảy ra: `currentDose` là lớn hơn hoặc bằng `minimumDose` và nhỏ hơn hoặc bằng `maxDose`.
2. Nhánh `then` của câu lệnh `if` thứ 2 được thực thi. Trong trường hợp đó, việc gán `currentDose` bằng 0 được thực thi. Do đó, hậu điều kiện đó là `currentDose = 0`.
3. Nhánh `else-if` của câu lệnh `if` thứ 2 được thực thi. Trong trường hợp đó, việc gán `currentDose` bằng `maxDose` được thực thi. Do đó, hậu điều kiện đó là `currentDose=maxDose`.

Trong cả ba trường hợp trên, các hậu điều kiện mâu thuẫn với các tiền điều kiện về tính không an toàn là liều lượng thuốc được phân phối là nhiều hơn `maxDose`, vì vậy hệ thống là an toàn.

#### 6.3.2. Đảm bảo quy trình

Tôi đã thảo luận tầm quan trọng của việc đảm bảo chất lượng của quá trình phát triển hệ thống trong phần giới thiệu chương này. Đây là điều quan trọng với tất cả các hệ thống quan trọng nhưng nó đặc biệt quan trọng với các hệ thống tính an toàn quan trọng. Có 2 lý do cho điều này:

1. Các rủi ro ít xảy ra trong các hệ thống quan trọng và nó có thể là không thể xảy ra được trong thực tế để mô phỏng chúng trong lúc kiểm thử hệ thống. Bạn không thể dựa trên kiểm thử bao quát để tạo ra các điều kiện có thể dẫn tới một tai nạn.
2. Các yêu cầu về tính an toàn, như tôi đã thảo luận trong chương 9, đôi khi là những yêu cầu “sẽ không” (*shall not*) loại trừ hành vi không an toàn của hệ thống. Nó không thể được chứng minh thuyết phục thông qua kiểm thử và các hoạt động thẩm định khác rằng các yêu cầu đó đã được thực thi.

Mô hình vòng đời cho quá trình phát triển hệ thống tính an toàn quan trọng làm cho điều đó rõ ràng rằng sự chú ý rõ ràng nên dành cho tính an toàn trong tất cả các bước của quy trình phần mềm. Điều đó có nghĩa là các hoạt động đảm bảo tính an toàn phải được bao gồm trong quy trình. Nó bao gồm:

1. Việc tạo thành một đoạn rủi ro và giám sát hệ thống lần theo những rủi ro từ phân tích tính rủi ro ban đầu thông qua kiểm thử và thẩm định hệ thống.
2. Bổ nhiệm vào dự án các kỹ sư về tính an toàn, những người có trách nhiệm rõ ràng về tính an toàn của hệ thống.
3. Sử dụng tính an toàn bao quát xem xét lại trong toàn bộ quy trình phát triển.
4. Tạo thành sự chứng nhận tính an toàn hệ thống bởi tính an toàn của các thành phần quan trọng tính an toàn đã chính thức được chứng nhận.
5. Sử dụng hệ thống quản lý cấu hình rất chi tiết (xem chương 29), mà đã được sử dụng để theo dõi tất cả các tài liệu về tính an toàn liên quan và giữ nó trong từng bước với tài liệu kỹ thuật liên quan. Có một điểm nhỏ trong thủ tục thẩm định

nghiêm ngặt nếu một lỗi trong cấu hình quản lý có nghĩa là một hệ thống không đúng được phân phối tới khách hàng.

Hazard Log	Trang 4: được in ngày 20.02.2003
Hệ thống: Hệ thống bơm Insulin	File: Insulin/Safety/HazardLog
Kỹ sư đảm bảo: James Brown	Phiên bản Log: 1/3
Xác định rủi ro: Lượng Insulin được phân phối quá liều lượng tới bệnh nhân	
Xác định bởi: JaneWilliams	
Mức quan trọng: 1	
Xác định sự rủi ro: Cao	
Xác định cây khiếm khuyết: Có ngày 24.01.99 Vị trí: Hazard Log, trang 5	
Người tạo cây khiếm khuyết: Jane Williams và Bill Smith	
Kiểm tra cây khiếm khuyết: Ngày 28.01.99 Người kiểm tra James Brown.	

Các yêu cầu thiết kế tính an toàn của hệ thống

1. Hệ thống sẽ bao gồm phần mềm tự kiểm thử mà sẽ kiểm tra hệ thống cảm biến, đồng hồ và hệ thống phân phối Insulin.
2. Phần mềm tự kiểm tra sẽ được thực thi ít nhất một lần mỗi phút.
3. Khi phần mềm tự kiểm tra phát hiện một sai sót trong bất kỳ một thành phần nào, một cảnh báo sẽ được phát ra và bơm hiển thị sẽ cho biết tên của thành phần mà sai sót đã được phát hiện. Việc phân phối Insulin sẽ bị trì hoãn.
4. Hệ thống sẽ kết hợp với hệ thống ghi đề để cho phép người sử dụng hệ thống sửa đổi liều lượng Insulin đã tính để phân phối bởi hệ thống.
5. Lượng ghi đề nên được giới hạn không lớn hơn một giá trị định trước là một tập mà hệ thống đã được cấu hình bởi các chuyên gia y tế.

Hình 6.8 Một trang đơn giản Hazard log

Để minh họa việc đảm bảo tính an toàn, tôi đã sử dụng quá trình phân tích rủi ro mà nó là một phần thiết yếu của quá trình phát triển các hệ thống tính tin cậy quan trọng. Phân tích rủi ro liên quan đến việc xác định các rủi ro, khả năng có thể xảy ra của chúng và khả năng mà các rủi ro đó sẽ dẫn đến tai nạn. Nếu quá trình phát triển bao gồm các dấu hiệu rõ ràng từ nhận dạng rủi ro trong chính hệ thống đó, thì một luận cứ có thể chứng minh được tại sao các rủi ro đó không dẫn đến các tai nạn. Đây có thể được bổ sung vào các luận cứ về tính an toàn, như đã thảo luận trong phần 6.2.1. Khi sự xác nhận bên ngoài được yêu cầu trước khi hệ thống được sử dụng (ví dụ, một máy bay), nó thường là điều kiện xác nhận rằng các dấu vết này có thể được chứng minh.

Các tài liệu tính an toàn trung tâm là hazard log, nơi mà các rủi ro được xác định trong quá trình đặc tả được chứng minh và chỉ ra. Sau đó, Hazard log được sử dụng tại mỗi giai đoạn trong quá trình phát triển phần mềm để đánh giá rằng giai đoạn phát triển đó đã đưa các rủi ro đó vào bản kê khai. Một ví dụ đơn giản của một hazard log đầu vào cho hệ thống phân phối insulin được chỉ ra trên hình 6.8. Mẫu tài liệu này chứng minh quá trình phân tích rủi ro và chỉ ra các yêu cầu thiết kế mà đã được sinh ra trong quá trình này. Các yêu cầu thiết kế đó được dự định để đảm bảo rằng hệ thống điều khiển có thể không bao giờ phân phối quá liều lượng insulin tới người dùng.

Như chỉ ra trên hình 6.8, các cá nhân chịu trách nhiệm về tính an toàn nên được xác định rõ ràng. Các dự án phát triển hệ thống tính an toàn quan trọng nên bổ nhiệm một kỹ sư về tính an toàn, người mà không liên quan trong việc phát triển hệ thống. Trách nhiệm của kỹ sư này là đảm bảo việc kiểm tra tính an toàn thích hợp đã được tạo thực hiện và chứng minh. Hệ thống đó cũng có thể yêu cầu một người thẩm định tính an toàn độc lập được bổ nhiệm từ một tổ chức bên ngoài, người này sẽ báo cáo trực tiếp tới khách hàng các vấn đề về tính an toàn.

Trong một số lĩnh vực, các kỹ sư hệ thống có trách nhiệm về tính an toàn phải được cấp giấy chứng nhận. Ở Anh, điều này có nghĩa là họ phải được thừa nhận như là một thành viên của một viện kỹ nghệ (về điện, cơ khí,...) và phải là các kỹ sư có đủ tư cách hành nghề. Các kỹ sư thiếu kinh nghiệm, chất lượng kém có thể không đảm bảo trách nhiệm về tính an toàn.

Hiện nay, điều này không được áp dụng với các kỹ sư phần mềm, mặc dù nó đã được thảo luận rộng rãi về giấy phép của các kỹ sư ở một số bang của nước Mỹ (Knight và Leveson, 2002; Begert, 2002). Tuy nhiên, trong tương lai, các tiêu chuẩn của quá trình phát triển phần mềm tính an toàn quan trọng có thể yêu cầu các kỹ sư về tính an toàn của dự án nên là các kỹ sư đã được cấp giấy chứng nhận chính thức với một cấp độ thấp nhất của quá trình đào tạo.

### 6.3.3. Kiểm tra tính an toàn khi thực hiện

Kỹ thuật tương tự có thể được sử dụng để giám sát động các hệ thống tính an toàn quan trọng. Các mã kiểm tra có thể được thêm vào hệ thống để kiểm tra một ràng buộc về tính an toàn. Nó đưa một ngoại lệ nếu ràng buộc đó bị vi phạm. Các ràng buộc về tính an toàn nên luôn được giữ tại các điểm cụ thể trong một chương trình có thể được biểu thị như các xác nhận. Các xác nhận đó mô tả các điều kiện phải được đảm bảo trước khi các câu lệnh tiếp theo có thể được thực hiện. Trong các hệ thống tính an toàn quan trọng, các xác nhận nên được sinh ra từ các đặc tả tính an toàn. Nó được dự định để đảm bảo hành vi an toàn hơn hành vi theo các đặc tả.

Các xác nhận có thể có giá trị đặc biệt trong việc đảm bảo tính an toàn trong giao tiếp giữa các thành phần của hệ thống. Ví dụ, trong hệ thống phân phối insulin, liều thuốc của người quản lý insulin cùng với các tín hiệu được sinh ra tới bơm insulin để phân phối lượng tăng xác định insulin (hình 6.9). Lượng tăng insulin cùng với liều lượng insulin lớn nhất cho phép có thể được tính toán trước và được tính đến như một xác nhận trong hệ thống.

Nếu có một lỗi trong việc tính toán *currentDose*, *currentDose* là một biến trạng thái giữ lượng insulin được phân phối, hoặc nếu giá trị này đã bị sửa đổi theo một cách nào đó, thì nó sẽ bị chặn lại tại bước này. Một liều lượng insulin quá mức sẽ không được phân phối, khi phương thức kiểm tra đảm bảo rằng bơm sẽ không phân phối nhiều hơn *maxDose*.



```

static void administerInsulin() throw SafetyException {
    int maxIncrements = InsulinPump.maxDose / 8;
    int increments = InsulinPump.currentDose / 8;

    // xác nhận currentDose <= InsulinPump.maxDose;

    if (InsulinPump.currentDose > InsulinPump.maxDose)
        throw new SatefyException (Pump.doseHigh);
    else
        for (int i = 1; i <= increments; i++)
        {
            generateSignal();
            if (i > maxIncrements)
                throw new SatefyException (Pump.incorrectIncrements);
        } // for loop
} // administerInsulin

```

**Hình 6.9** Quản lý insulin bằng cách kiểm tra lúc thực thi

Từ các xác nhận tính an toàn được bao gồm như các lời chú giải chương trình, viết mã để kiểm tra các xác nhận đó có thể được sinh ra. Bạn có thể xem hình 6.9, câu lệnh if sau các chú giải xác nhận kiểm tra xác nhận đó. Về nguyên tắc, việc sinh các mã này có thể được sinh ra một cách tự động bằng cách sử dụng bộ tiền xử lý xác nhận. Tuy nhiên, các công cụ thường phải được viết riêng biệt và thông thường mã xác nhận được sinh bằng tay.

#### 6.4. Các trường hợp an toàn và tin cậy được

Các trường hợp an toàn và, tổng quát hơn, các trường hợp tin cậy được cấu trúc thành tài liệu, đưa ra các luận chứng và chứng cứ chi tiết để chứng minh hệ thống là an toàn hoặc mức yêu cầu của tính tin cậy được đã đạt được. Với nhiều loại hệ thống quan trọng, đưa ra một trường hợp an toàn là một yêu cầu theo luật định, và trường hợp đó phải thỏa mãn một số chứng nhận chính trước khi hệ thống có thể được triển khai.

Những người điều chỉnh được tạo ra bởi chính phủ để đảm bảo rằng kỹ nghệ mật không được lợi dụng sự thiếu các tiêu chuẩn quốc gia về tính an toàn, tính bảo mật,... Có nhiều người điều chỉnh trong nhiều lĩnh vực kỹ nghệ khác nhau. Ví dụ, ngành hàng không được điều chỉnh bởi những người trong lĩnh vực hàng không quốc gia như FAA (tại Mỹ) và CAA (tại Anh). Những người điều chỉnh ngành đường sắt tồn tại để đảm bảo tính an toàn trong ngành đường sắt, những người điều chỉnh hạt nhân phải chứng nhận tính an toàn của khu vực xử lý hạt nhân trước khi nó có thể hoạt động. Trong lĩnh vực ngân hàng, các ngân hàng nhà nước phụ vụ với vai trò như những người điều chỉnh, thiết lập các thủ tục và các bài tập để giảm khả năng gian

**lặn và bảo vệ khách hàng trước những rủi ro. Khi các hệ thống phần mềm ngày càng tăng tầm quan trọng trong cơ sở hạ tầng của các quốc gia, những người điều chỉnh trở nên liên quan nhiều hơn tới các trường hợp an toàn và tin cậy được của các hệ thống phần mềm.**

Thành phần	Mô tả
Mô tả hệ thống	Tổng quan về hệ thống và mô tả các thành phần quan trọng của nó.
Các yêu cầu tính an toàn	Các yêu cầu tính an toàn rút ra từ đặc tả yêu cầu của hệ thống.
Phân tích các rủi ro và các nguy cơ	Các tài liệu mô tả các rủi ro và nguy cơ đã được xác định và các tiêu chuẩn được đưa ra để giảm các nguy cơ đó.
Xác nhận và thẩm định	Mô tả việc sử dụng thủ tục V & V và, chỗ thích hợp, các kiểm thử được thực hiện với hệ thống.
Báo cáo xem xét lại	Các hồ sơ của tất cả thiết kế và tính an toàn được xem xét lại
Nhóm năng lực	Chứng cứ năng lực của tất cả các nhóm cùng với việc phát triển và thẩm định hệ thống tính an toàn liên quan.
Quá trình QA	Các hồ sơ của quá trình đảm bảo chất lượng được thực hiện trong khi phát triển hệ thống.
Quá trình quản lý sự thay đổi	Các hồ sơ của tất cả các thay đổi được đưa ra và các hành động thực hiện và nơi thích hợp, sự biện minh tính an toàn của các thay đổi đó.
Các trường hợp kết hợp tính an toàn	Xem xét tới các trường hợp tính an toàn khác có thể tác động tới các trường hợp an toàn.

Hình 6.11. Các thành phần của một trường hợp an toàn của phần mềm

**Vai trò của những người điều chỉnh là kiểm tra xem hệ thống đã hoàn thành là an toàn và có thể thực hiện được, vì vậy họ chủ yếu được tập hợp khi một dự án phát triển được hoàn thành. Tuy nhiên, những người điều chỉnh và những người phát triển hiếm khi làm việc độc lập, họ liên lạc với đội phát triển để xác minh những gì phải tính đến trong một trường hợp an toàn. Những người điều chỉnh và những người phát triển cùng nhau thẩm tra các quá trình và các thủ tục để đảm bảo rằng nó đã được ban hành và chứng minh thảo luận người điều khiển.**

**Tất nhiên, bản thân phần mềm không nguy hiểm. Nó chỉ nguy hiểm khi nó được nhúng vào trong một hệ thống lớn, hệ thống dựa trên máy tính hoặc hệ thống chuyên môn xã hội mà những sai sót của phần mềm có thể dẫn đến sai sót của các thiết bị khác hoặc cuar các quá trình mà có thể gây ra tổn hại và cái chết. Vì vậy, một trường hợp an toàn của phần mềm luôn luôn là một phần của trường hợp an toàn hệ thống rộng hơn để chứng minh tính an toàn của toàn bộ hệ thống. Khi xây dựng một trường hợp an toàn của phần mềm, bạn phải liên hệ những sai sót của phần mềm với những sai sót của hệ thống lớn hơn và chứng minh rằng hoặc những sai sót của phần mềm sẽ**

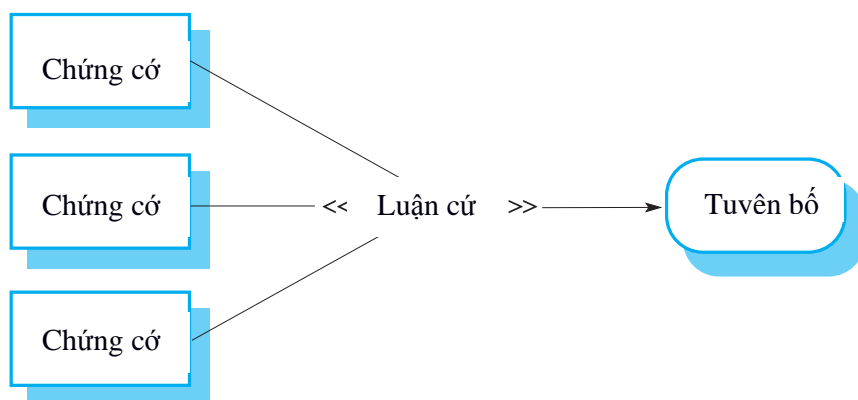
không xảy ra hoặc nó sẽ không làm lan rộng ra theo cách làm cho các sai sót nguy hiểm của hệ thống có thể xảy ra.

Một trường hợp an toàn là một tập các tài liệu bao gồm mô tả hệ thống đã được chứng nhận, thông tin về các quá trình sử dụng để phát triển hệ thống và các luận chứng hợp logic để chứng minh rằng hệ thống là có khả năng an toàn. Bishop và Bloomfield đã đưa ra định nghĩa ngắn gọn về một trường hợp an toàn như sau:

Một tài liệu nhiều bằng chứng cung cấp một luận chứng thuyết phục và hợp lệ rằng hệ thống thỏa mãn tính an toàn với ứng dụng đưa ra trong môi trường đưa ra.

Sự tổ chức và nội dung của một trường hợp an toàn phụ thuộc vào kiểu của hệ thống đã được chứng nhận và ngữ cảnh hoạt động của nó. Hình 6.11 chỉ ra một tổ chức có thể xảy ra với một trường hợp an toàn của phần mềm.

Thành phần then chốt của một trường hợp an toàn là một tập các luận chứng hợp logic về tính an toàn của hệ thống. Nó có thể là các luận chứng xác thực (sự kiện X sẽ hoặc sẽ không xảy ra) hoặc các luận chứng khả năng (xác suất của sự kiện X là 0.Y); khi được kết hợp, nó có thể chứng minh được tính an toàn. Như trên hình 24.12, một luận chứng là một mối liên hệ giữa cái gì được nghĩ là một trường hợp (một tuyên bố) và khung chứng cứ đã được thu thập. Về cơ bản, luận chứng đó giải thích tại sao tuyên bố đó (nói chung điều gì đó là an toàn) có thể được suy ra từ các chứng cứ đó. Tất nhiên, đưa ra nhiều mức tự nhiên của các hệ thống, các tuyên bố được tổ chức trong một hệ thống phân cấp. Để chứng minh rằng một tuyên bố mức cao là hợp lệ, đầu tiên bạn phải thực hiện với các luận chứng mức thấp hơn. Hình 24.13 chỉ ra một phần của hệ thống phân cấp tuyên bố phân phối kim tiêm Insulin.



Hình 6.12 Cấu trúc của một luận cứ

Như một thiết bị y tế, hệ thống bơm Insulin có thể phải được chứng nhận bên ngoài. Ví dụ, ở Anh, MDD phải đưa ra một chứng nhận an toàn với bất kỳ thiết bị y tế nào được bán tại Anh. Nhiều luận chứng khác nhau có thể phải đưa ra để chứng minh tính an toàn của hệ thống đó. Ví dụ, các luận chứng dưới đây có thể là một phần trường hợp an toàn của hệ thống bơm Insulin.

**Tuyên bố:** Một liều thuốc lớn nhất được tính bởi hệ thống bơm Insulin sẽ không vượt quá maxDose.

**Chứng cứ:** Luận chứng an toàn cho hệ thống Insulin (hình 6.7).

**Chứng cứ:** Các tập dữ liệu thử nghiệm cho hệ thống Insulin.

**Chứng cứ:** Báo cáo phân tích tĩnh cho phần mềm bơm Insulin.

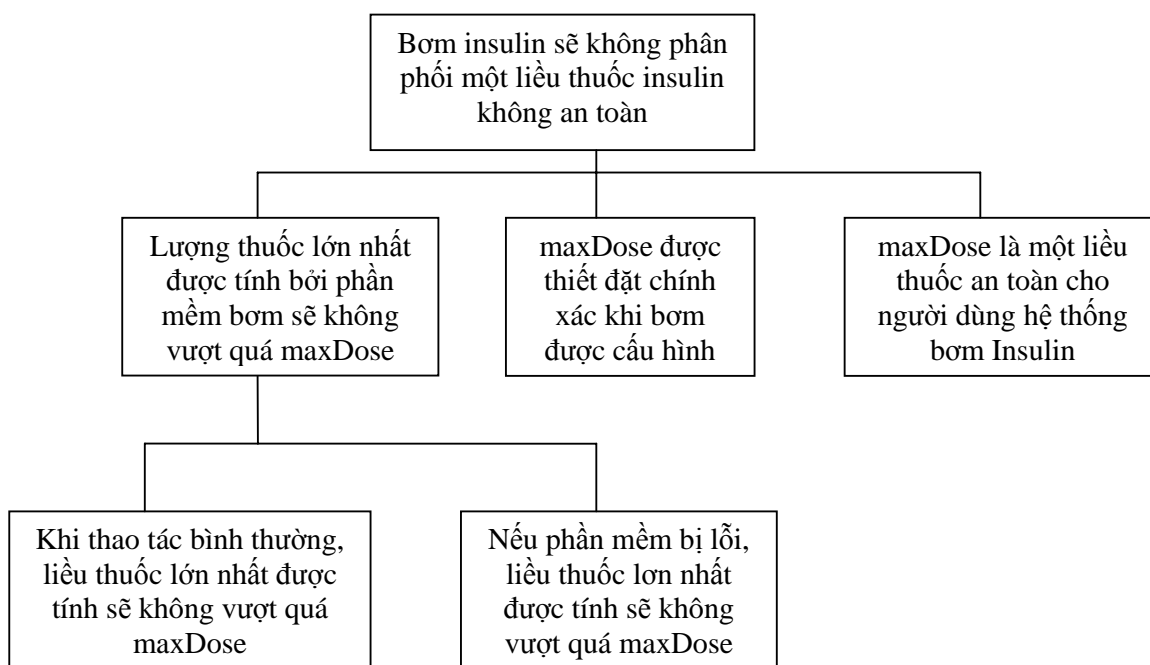
**Luận cứ:** Luận cứ an toàn chỉ ra liều lượng insulin lớn nhất có thể được tính bằng với maxDose.

Trong 400 thử nghiệm, giá trị của Dose được tính chính xác và không bao giờ vượt quá maxDose.

Phân tích tĩnh phần mềm điều khiển không xuất hiện dị thường.

Tất cả điều hợp lý để thừa nhận rằng tuyên bố đã được khẳng định.

**Tất nhiên, đây là một luận chứng rất đơn giản, và trong một trường hợp an toàn thực tế cụ thể tham khảo đến các chứng cứ sẽ được đưa ra. Bởi vì, chi tiết tự nhiên của nó, do đó, các trường hợp an toàn là các tài liệu rất dài và phức tạp. Các công cụ phân mềm khác nhau có khả năng giúp xây dựng chúng, và tôi đã bao gồm các liên kết tới các công cụ đó trong các trang web của cuốn sách này.**



Hình 6.13 Hệ thống phân cấp tuyên bố trong trường hợp tính an toàn của hệ thống bơm insulin

## Những vấn đề trọng tâm

- Kiểm thử thống kê được sử dụng đánh giá tính tin cậy của phần mềm. Nó dựa vào kiểm thử hệ thống với một tập dữ liệu thử nghiệm phản ánh sơ thảo hoạt động của phần mềm. Dữ liệu thử nghiệm có thể được sinh ra tự động.
- Các mô hình phát triển tính tin cậy biểu thị sự thay đổi tính tin cậy khi các thiếu sót được tháo gỡ từ phần mềm trong quá trình kiểm thử. Các mô hình tính tin cậy có thể được sử dụng để dự đoán khi các yêu cầu tính tin cậy sẽ đạt được.
- Chứng minh tính tin cậy là một kỹ thuật hiệu quả đảm bảo tính tin cậy của sản phẩm. Nó chỉ ra các điều kiện có tính rủi ro xác định có thể không bao giờ xuất hiện. Nó thường đơn giản hơn việc chứng minh rằng chương trình đáp ứng đầy đủ đặc tả.
- Điều quan trọng để có một định nghĩa rõ ràng, chứng nhận quá trình phát triển các hệ thống tính tin cậy quan trọng. Quá trình đó phải bao gồm sự xác nhận và giám sát các rủi ro tiềm năng.
- Thẩm định tính bảo mật có thể được thực hiện bằng cách sử dụng phân tích dựa trên kinh nghiệm, phân tích dựa trên công cụ hoặc “đội hổ”(tiger teams) mà mô phỏng việc tấn công vào hệ thống.
- Các trường hợp tính an toàn tập hợp tất cả các chứng cứ để chứng minh một hệ thống là an toàn. Các trường hợp an toàn đó được yêu cầu khi một bộ điều khiển bên ngoài phải xác nhận hệ thống trước khi nó được sử dụng.

*‘Best practices in code inspection for safety-critical software’.* Bài báo giới thiệu một bản liệt kê các mục cần kiểm tra của các nguyên tắc để kiểm tra và xem xét phần mềm tính an toàn quan trọng. (J.R. de Almeida, *IEEE Software*, 20(3), May/June 2003).

*‘Statically scanning Java code: Finding security vulnerabilities’.* Đây là một bài báo hay về vấn đề ngăn ngừa việc tấn công tính bảo mật. Nó thảo luận cách các tấn công đó được thực hiện và cách phát hiện chúng bằng một phân tích tĩnh. (J. Viega, *IEEE Software*, 17(5), September/October 2000).

*‘Software Reliability Engineering: More Reliable Software, Faster Development and Testing’.* Đây chắc chắn là rõ ràng về việc sử dụng các sơ thảo thao tác và mô hình tính tin cậy để đánh giá tính tin cậy. Nó bao gồm kinh nghiệm chi tiết về kiểm thử thốn kê. (J.D. Musa, 1998, McGraw-Hill).

*‘Safety-critical Computer System’.* Đây là một cuốn sách giáo khoa rất hay bao gồm một số chương hay về vị trí của những phương thức hình thức trong quá trình phát triển phần mềm tính tin cậy quan trọng. (N. Storey, 1996, Addison-Wesley).

*Safeware: System Safety and Computers.* Cuốn sách này bao gồm một số chương về việc thẩm định các hệ thống tính an toàn quan trọng với rất nhiều chi tiết hơn tôi đã đưa ra ở đây về việc sử dụng những luận chứng về tính an toàn dựa trên cây thiếu sót. (N. Leveson, 1995, Addison-Wesley).

## **Bài tập**

1. Mô tả cách bạn sẽ sử dụng để thẩm định đặc tả tính tin cậy của hệ thống siêu thị mà bạn đã xác định trong bài tập 9.8. Câu trả lời của bạn nên bao gồm sự mô tả các công cụ thẩm định có thể được sử dụng.

2. Giải thích tại sao thực tế không thể làm được việc thẩm định đặc tả tính tin cậy khi có giới hạn rõ ràng của rất ít lỗi qua toàn bộ cuộc đời của một hệ thống.
3. Sử dụng tác phẩm văn học như thông tin nền tảng, viết một báo cáo cho người quản lý (những người chưa có kinh nghiệm trong lĩnh vực này) về cách sử dụng mô hình phát triển tính tin cậy.
4. Có hợp với đạo đức không khi một kỹ sư đồng ý để phân phối một hệ thống phần mềm mà đã biết có các thiếu sót tới khách hàng? Điều đó có thể tạo nên nhiều điều khác nhau nếu khách hàng nói có tồn tại các thiếu sót trong phần mềm? Như thế nào là hợp lý để đưa ra tuyên bố về tính tin cậy của phần mềm trong hoàn cảnh đó?
5. Giải thích tại sao đảm bảo tính tin cậy của hệ thống không phải là một sự đảm bảo về tính an toàn của hệ thống.
6. Cơ cấu điều khiển việc khóa cửa trong điều kiện lưu trữ chất thải hạt nhân được thiết kế để hoạt động an toàn. Nó đảm bảo lối đi vào kho lưu trữ chỉ được cho phép khi tấm chắn sự phóng xạ được đặt hoặc khi mức độ phóng xạ trong phòng giảm xuống đến giá trị đã cho (dangerLevel). Đó là:
  - a. Nếu tấm chắn điều khiển tự động là được đặt trong một phòng, cửa có thể được mở bởi người điều hành có thẩm quyền.

## CHƯƠNG 7: KIỂM THỬ PHẦN MỀM TRONG CÔNG NGHIỆP

Trong phần trước chúng tôi đã giới thiệu tổng quan về các mức và loại kiểm tra phần mềm (KTPM) cơ bản. Thực tế đi sâu vào từng mức và loại kiểm tra, còn có rất nhiều kiểm tra đặc thù khác nữa, mang tính chuyên biệt cho từng vấn đề hoặc từng loại ứng dụng. Trong phần này, chúng tôi sẽ giới thiệu chi tiết về những bước cơ bản của một quy trình KTPM, làm thế nào để đánh giá và cải tiến năng lực KTPM của một tổ chức thông qua mô hình TMM (Testing Maturity Model), được các chuyên gia đánh giá khá tốt, dành riêng cho hoạt động KTPM

### 7.1. QUY TRÌNH KIỂM TRA PHẦN MỀM CƠ BẢN

Trước khi tìm hiểu một quy trình kiểm tra phần mềm cơ bản, ta cần hiểu hai khái niệm sau: Test Case và Test Script.

#### 7.1.1. Test Case

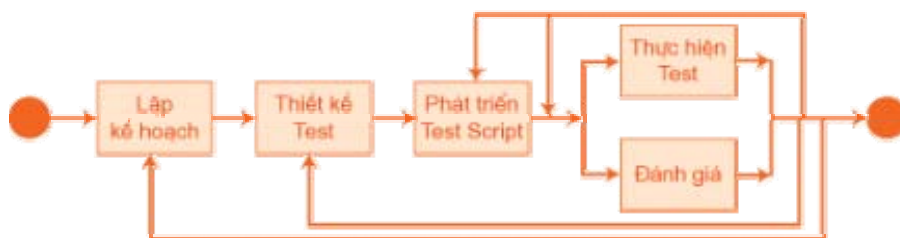
Một Test Case có thể coi nôm na là một tình huống kiểm tra, được thiết kế để kiểm tra một đối tượng có thỏa mãn yêu cầu đặt ra hay không. Một Test Case thường bao gồm 3 phần cơ bản:

- **Mô tả:** đặc tả các điều kiện cần có để tiến hành kiểm tra.
- **Nhập:** đặc tả đối tượng hay dữ liệu cần thiết, được sử dụng làm đầu vào để thực hiện việc kiểm tra.
- **Kết quả mong chờ:** kết quả trả về từ đối tượng kiểm tra, chứng tỏ đối tượng đạt yêu cầu.

#### 7.1.2. Test Script

Một Test Script là một nhóm mã lệnh dạng đặc tả kịch bản dùng để tự động hóa một trình tự kiểm tra, giúp cho việc kiểm tra nhanh hơn, hoặc cho những trường hợp mà kiểm tra bằng tay sẽ rất khó khăn hoặc không khả thi. Các Test Script có thể tạo thủ công hoặc tạo tự động dùng công cụ kiểm tra tự động. (Hình 04)

Phần sau sẽ giải thích rõ hơn các bước cơ bản của một quy trình kiểm tra.



**Hình 7.1:** Một quy trình kiểm tra cơ bản có thể áp dụng rộng rãi cho nhiều hệ thống PM với những đặc trưng khác nhau.

### 7.1.3. Lập kế hoạch kiểm tra

**Mục đích:** Nhằm chỉ định và mô tả các loại kiểm tra sẽ được triển khai và thực hiện. Kết quả của bước lập kế hoạch là bản tài liệu kế hoạch KTPM, bao gồm nhiều chi tiết từ các loại kiểm tra, chiến lược kiểm tra, cho đến thời gian và phân định lực lượng kiểm tra viên.

Bản kế hoạch kiểm tra đầu tiên được phát triển rất sớm trong chu trình phát triển phần mềm (PTPM), ngay từ khi các yêu cầu đã tương đối đầy đủ, các chức năng và luồng dữ liệu chính đã được mô tả. Bản kế hoạch này có thể được coi là bản kế hoạch chính (master test plan), trong đó tất cả các kế hoạch chi tiết cho các mức kiểm tra và loại kiểm tra khác nhau đều được đề cập (hình 05).

Lưu ý, tùy theo đặc trưng và độ phức tạp của mỗi dự án, các kế hoạch kiểm tra chi tiết có thể được gom chung vào bản kế hoạch chính hoặc được phát triển riêng.

Sau khi bản kế hoạch chính được phát triển, các bản kế hoạch chi tiết lần lượt được thiết kế theo trình tự thời gian phát triển của dự án. (Hình 06 minh họa thời điểm phù hợp để thiết lập các kế hoạch kiểm tra, gắn liền với quá trình phát triển của dự án. Quá trình phát triển các kế hoạch kiểm tra không dừng lại tại một thời điểm, mà liên tục được cập nhật chỉnh sửa cho phù hợp đến tận cuối dự án.).



**Hình 05:** Bản kế hoạch chính và các bản kế hoạch chi tiết

#### Các bước lập kế hoạch:

- **Xác định yêu cầu kiểm tra:** chỉ định bộ phận, thành phần của PM sẽ được kiểm tra, phạm vi hoặc giới hạn của việc kiểm tra. Yêu cầu kiểm tra cũng được dùng để xác định nhu cầu nhân lực.
- **Khảo sát rủi ro:** Các rủi ro có khả năng xảy ra làm chậm hoặc cản trở quá trình cũng như chất lượng kiểm tra. Ví dụ: kỹ năng và kinh nghiệm của kiểm tra viên quá yếu, không hiểu rõ yêu cầu.
- **Xác định chiến lược kiểm tra:** chỉ định phương pháp tiếp cận để thực hiện việc kiểm tra trên PM, chỉ định các kỹ thuật và công cụ hỗ trợ kiểm tra, chỉ định các phương pháp dùng để đánh giá chất lượng kiểm tra cũng như điều kiện để xác định thời gian kiểm tra.
- **Xác định nhân lực, vật lực:** kỹ năng, kinh nghiệm của kiểm tra viên; phần cứng, phần mềm, công cụ, thiết bị giả lập... cần thiết cho việc kiểm tra.



- Lập kế hoạch chi tiết: ước lượng thời gian, khối lượng công việc, xác định chi tiết các phần công việc, người thực hiện, thời gian tất cả các điểm mốc của quá trình kiểm tra.
- Tổng hợp và tạo các bản kế hoạch kiểm tra: kế hoạch chung và kế hoạch chi tiết.
- Xem xét các kế hoạch kiểm tra: phải có sự tham gia của tất cả những người có liên quan, kể cả trưởng dự án và có thể cả khách hàng. Việc xem xét nhằm bảo đảm các kế hoạch là khả thi, cũng như để phát hiện (và sửa chữa sau đó) các sai sót trong các bản kế hoạch.

#### 7.1.4. Thiết kế Test

Mục đích: Nhằm chỉ định các Test Case và các bước kiểm tra chi tiết cho mỗi phiên bản PM. Giai đoạn thiết kế test là hết sức quan trọng, nó bảo đảm tất cả các tình huống kiểm tra "quét" hết tất cả yêu cầu cần kiểm tra.

Hình 06 cho thấy việc thiết kế test không phải chỉ làm một lần, nó sẽ được sửa chữa, cập nhật, thêm hoặc bớt xuyên suốt chu kỳ PTPM, vào bất cứ lúc nào có sự thay đổi yêu cầu, hoặc sau khi phân tích thấy cần được sửa chữa hoặc bổ sung.



**Hình 7:** Thời điểm phù hợp để thiết lập các kế hoạch kiểm tra

Các bước thiết kế test bao gồm:

- **Xác định và mô tả Test Case:** xác định các điều kiện cần thiết lập trước và trong lúc kiểm tra. Mô tả đối tượng hoặc dữ liệu đầu vào, mô tả các kết quả mong chờ sau khi kiểm tra.
- **Mô tả các bước chi tiết để kiểm tra:** các bước này mô tả chi tiết để hoàn thành một Test Case khi thực hiện kiểm tra. Các Test Case như đã nói ở trên thường chỉ mô tả đầu vào, đầu ra, còn cách thức tiến hành như thế nào thì không được định nghĩa. Thao tác này nhằm chi tiết hóa các bước của một Test Case, cũng như chỉ định các loại dữ liệu nào cần có để thực thi các Test Case, chúng bao gồm các loại dữ liệu trực tiếp, gián tiếp, trung gian, hệ thống...
- **Xem xét và khảo sát độ bao phủ của việc kiểm tra:** mô tả các chỉ số và cách thức xác định việc kiểm tra đã hoàn thành hay chưa? bao nhiêu phần trăm PM đã được kiểm tra? Để xác định điều này có hai phương pháp: căn cứ trên yêu cầu của phần mềm hoặc căn cứ trên số lượng code đã viết.

- Xem xét Test Case và các bước kiểm tra: Việc xem xét cần có sự tham gia của tất cả những người có liên quan, kể cả trưởng dự án nhằm bảo đảm các Test Case và dữ liệu yêu cầu là đủ và phản ánh đúng các yêu cầu cần kiểm tra, độ bao phủ đạt yêu cầu, cũng như để phát hiện (và sửa chữa) các sai sót.

#### 7.1.5. Phát triển Test Script

kết quả nhận được là đáng tin cậy, cũng như nhận biết được những lỗi xảy ra không phải do PM mà do dữ liệu dùng để kiểm tra, môi trường kiểm tra hoặc các bước kiểm tra (hoặc Test Script) gây ra. Nếu thực sự lỗi xảy ra do quá trình kiểm tra, cần phải sửa chữa và kiểm tra lại từ đầu.

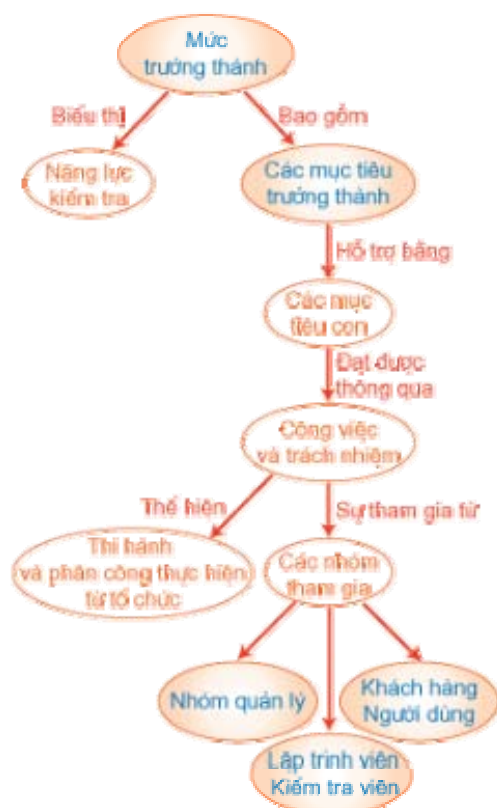
#### 7.1.6. Đánh giá quá trình kiểm tra

Mục đích: Đánh giá toàn bộ quá trình kiểm tra, bao gồm xem xét và đánh giá kết quả kiểm tra, liệt kê lỗi, chỉ định các yêu cầu thay đổi, và tính toán các số liệu liên quan đến quá trình kiểm tra (chẳng hạn số giờ, thời gian kiểm tra, số lượng lỗi, phân loại lỗi...).

Lưu ý, mục đích của việc đánh giá kết quả kiểm tra ở bước này hoàn toàn khác với bước thẩm định kết quả kiểm tra sau khi hoàn tất một vòng kiểm tra. Đánh giá kết quả kiểm tra ở giai đoạn này mang tính toàn cục và nhằm vào bản thân giá trị của các kết quả kiểm tra.

Hình 06 cho thấy, việc đánh giá quá trình và kết quả kiểm tra được thực hiện song song với bất kỳ lần kiểm tra nào và chỉ chấm dứt khi quá trình kiểm tra đã hoàn tất.

Đánh giá quá trình kiểm tra thường thông qua các bước sau:



Hình 07: Cấu trúc của một mức trưởng thành trong mô hình TMM

- **Phân tích kết quả kiểm tra và đề xuất yêu cầu sửa chữa:** Chỉ định và đánh giá sự khác biệt giữa kết quả mong chờ và kết quả kiểm tra thực tế, tổng hợp và gửi thông tin yêu cầu sửa chữa đến những người có trách nhiệm trong dự án, lưu trữ để kiểm tra sau đó.

- **Đánh giá độ bao phủ:** Xác định quá trình kiểm tra có đạt được độ bao phủ yêu cầu hay không, tỷ lệ yêu cầu đã được kiểm tra (tính trên các yêu cầu của PM và số lượng code đã viết).

- **Phân tích lỗi:** Đưa ra số liệu phục vụ cho việc cải tiến các qui trình phát triển, giảm sai sót cho các chu kỳ phát triển và kiểm tra sau đó. Ví dụ, tính toán tỷ lệ phát sinh lỗi, xu hướng gây ra lỗi, những lỗi "ngoan cố" hoặc thường xuyên tái xuất hiện.

- **Xác định quá trình kiểm tra có đạt yêu cầu hay không:** Phân tích đánh giá để xem các Test Case và chiến lược kiểm tra đã thiết kế có bao phủ hết những điểm cần kiểm tra hay không? Kiểm tra có đạt yêu cầu dự án không? Từ những kết quả này, kiểm tra viên có thể sẽ phải thay đổi chiến lược hoặc cách thức kiểm tra.

- **Báo cáo tổng hợp:** Tổng hợp kết quả các bước ở trên và phải được gửi cho tất cả những người có liên quan.

**Tóm lược:** Trên đây là tóm tắt các bước cơ bản của một quy trình KTPM. Tùy theo đặc thù của dự án, loại kiểm tra và mức độ kiểm tra, quy trình kiểm tra trong thực tế có thể chi tiết hơn nhiều, tuy nhiên các bước trên là xương sống của bất kỳ quy trình kiểm tra nào.

Sau đây, chúng tôi sẽ giới thiệu một mô hình giúp các tổ chức đánh giá và nâng cao năng lực KTPM của mình, đó là mô hình TMM (Testing Maturity Model).

## **7.2. MÔ HÌNH KIỂM TRA PHẦN MỀM TMM (TESTING MATURITY MODEL)**

Mặc dù không ít người trong cũng như ngoài ngành biết hoặc đã từng nghe về mô hình CMM/CMMi (Capability Maturity Model/Intergration) của SEI (Software Engineering Institute – Viện công nghệ phần mềm của Mỹ) dùng để đánh giá và nâng cao năng lực PTPM, song có lẽ ít người biết về TMM - mô hình được các chuyên gia đánh giá là khá tốt – được dùng để đánh giá và nâng cao năng lực KTPM của một tổ chức.

TMM thực ra không mới, phần lớn nội dung của mô hình này đã được phát triển từ năm 1996, tuy nhiên chúng không được chấp nhận rộng rãi. Một trong những lý do chính đó là tài liệu về TMM rất ít. Các bài báo, sách về nó thường được viết dưới dạng nặng về lý thuyết. Một lý do nữa là phần lớn các tổ chức đều "say mê" mô hình CMM/CMMi và nghĩ rằng quá đủ cho qui trình PTPM của mình.

**Thực tế cho thấy không hoàn toàn như vậy**

KTPM là một bộ phận sống còn của quy trình PTPM, sự hỗ trợ quan trọng để đảm bảo chất lượng của PM. Nhiều tổ chức PM trong thực tế vẫn chưa nhận thấy tính non nớt yếu kém trong quy trình cũng như năng lực KTPM của họ. Các mô hình hàng đầu hiện nay như CMM/CMMi/ISO9000 thực tế vẫn không chú tâm đầy đủ vào các vấn đề của KTPM.

TMM được phát triển tại IIT (Illinois Institute of Technology – Viện công nghệ Illinois) vào giữa thập niên 90 trong bối cảnh hầu như chưa có quy trình PM nào đề cập một cách toàn diện vấn đề kiểm tra trong PTPM. Tương tự SW-CMM, nó có một cấu trúc cơ bản bao gồm 5 mức trưởng thành. Vì TMM là mô hình chuyên biệt cho lĩnh vực KTPM, các mức trưởng thành này trực tiếp mô tả các mục tiêu trưởng thành của một quy trình KTPM. Trong một tổ chức PM, TMM không mâu thuẫn mà có thể dùng độc lập hoặc phối hợp với CMM/CMMi.

Mục đích của TMM là hỗ trợ tổ chức PM đánh giá và cải tiến các quy trình và năng lực PM của mình, mục tiêu cuối cùng là giúp tổ chức có thể:

- Hoàn thành sản phẩm đúng hạn và trong phạm vi ngân sách đã định.

Tạo ra sản phẩm phần mềm có chất lượng cao hơn.

Xây dựng nền tảng cho việc cải tiến quy trình ở phạm vi rộng trong một tổ chức.

TMM bao gồm hai thành phần chính:

1. Tập hợp 5 mức độ trưởng thành, định nghĩa năng lực KTPM của một tổ chức. Mỗi mức độ bao gồm:

Mục tiêu

Hoạt động để hiện thực các mục tiêu

Công việc và phân công trách nhiệm

2. Mô hình đánh giá năng lực KTPM của một tổ chức, bao gồm

Bảng câu hỏi đánh giá

Thủ tục tiến hành đánh giá

Hướng dẫn để chọn lựa và huấn luyện nhóm đánh giá.

Phần sau ta sẽ khảo sát rõ hơn về các mức độ trưởng thành của TMM

#### 7.2.1. Cấu trúc của một mức trưởng thành

Các mức trưởng thành cấu thành TMM, vậy bản thân một mức trưởng thành là gì và cấu trúc của nó ra sao? (Hình 07).

Mỗi mức độ, ngoại trừ mức độ thấp nhất là 1, có cấu trúc bao gồm các thành phần sau:

- Mục tiêu trưởng thành: Xác định các mục tiêu cần phải đạt trong việc cải tiến quy trình KTPM. Để đạt một mức trưởng thành, tổ chức phải đạt tất cả các mục tiêu của mức trưởng thành đó.

- Mục tiêu con: Các mục tiêu trưởng thành đã nói ở trên có tầm bao quát rộng. Do vậy để làm rõ hơn phạm vi cũng như những công việc cần làm để đạt được một mục tiêu, mỗi mục tiêu lại được mô tả rõ hơn thông qua những mục tiêu con, dễ hiểu và cụ thể hơn. Nếu ta đạt được tất cả mục tiêu con của một mục tiêu nghĩa là ta đã đạt được mục tiêu đó.

- Công việc và trách nhiệm: Mô tả rõ hơn các công việc cần làm, cũng như ai trong dự án (trưởng dự án, lập trình viên, kiểm tra viên...) sẽ thực hiện các công việc đó. Nghĩa là, để đạt được một mục tiêu con, ta cần thực hiện tất cả các công việc được đề nghị cho mục tiêu con đó.

• Sự tham gia của các nhóm khác nhau: TMM cho rằng có 3 nhóm người quan trọng với cách nhìn và quan điểm khác nhau ảnh hưởng đến công việc KTPM, đó là người quản lý/quản lý dự án, lập trình viên/kiểm tra viên, và khách hàng/người sử dụng. Do vậy mô hình TMM yêu cầu các công việc phải được phân trách nhiệm cho 3 nhóm người này.

### 7.2.2. Ý nghĩa và tổ chức của các mức trưởng thành

5 mức độ trưởng thành do TMM quy định được xác định như hình 08.

#### Mức trưởng thành 1: Khởi đầu

Mức khởi đầu của đa số tổ chức PM, không có mục tiêu nào đặt ra cho mức này. Quy trình KTPM hoàn toàn hỗn độn. KTPM được thực hiện một cách không dự tính và phi thể thức sau khi code được viết xong; không có kế hoạch, không có quy trình. Nói chung ở mức này KTPM đồng nghĩa với tìm lỗi (debugging). Một lập trình viên viết code và sau đó tìm lỗi, sửa chữa, dò lỗi... cho đến khi tin rằng mọi thứ đạt yêu cầu. Kiểm tra viên không được huấn luyện, tài nguyên cần thiết cũng không đầy đủ.

Do hầu như chỉ có lập trình viên làm mọi thứ, chi phí kiểm tra hầu như không biết trước hoặc được bao gồm trong chi phí PTPM.

#### Mức trưởng thành 2: Định nghĩa

KTPM là một quy trình riêng biệt, là một chặng của toàn bộ chu trình PTPM và hoàn toàn phân biệt với công việc dò tìm lỗi (debug). Mục tiêu của kiểm tra nhằm chứng minh PM hoặc hệ thống đáp ứng được các yêu cầu.

KTPM được lập kế hoạch chi tiết và được theo dõi chặt chẽ. Quy trình kiểm tra có thể được sử dụng lặp lại trong các dự án khác nhau. Kế hoạch kiểm tra thường được hoàn thành sau khi đã xong giai đoạn viết code. Kỹ thuật và phương pháp kiểm tra cơ bản được thiết lập và đưa vào sử dụng. Các mục tiêu của mức 2 bao gồm:

- Phát triển các mục tiêu dò lỗi và kiểm tra phần mềm

Quy trình lập kế hoạch kiểm tra

Thể chế hóa các kỹ thuật và phương pháp kiểm tra cơ bản

#### Mức trưởng thành 3: Tích hợp

TMM Mức 2: Định nghĩa	CMM Mức 2: Có thể lặp lại
<ul style="list-style-type: none"> <li>• Kỹ thuật và phương pháp kiểm tra cơ bản</li> <li>• Quy trình lập kế hoạch kiểm tra</li> <li>• Mục tiêu dò lỗi và kiểm tra phần mềm</li> </ul>	<ul style="list-style-type: none"> <li>• Quản lý yêu cầu</li> <li>• Lập kế hoạch dự án</li> <li>• Giám sát và theo dõi dự án</li> <li>• Quản lý thầu phụ</li> <li>• Đảm bảo chất lượng</li> <li>• Quản lý cấu hình</li> </ul>

Một nhóm kiểm tra viên được thành lập như một bộ phận trong công ty. Kiểm tra viên được huấn luyện kỹ và đặc biệt. KTPM không còn là một chặng, mà được thực hiện xuyên suốt toàn bộ chu kỳ PTPM. Việc sử dụng công cụ kiểm tra tự động bắt đầu được tính đến. Kế hoạch kiểm tra được thực hiện sớm hơn nhiều so với mức

**trưởng thành 2. Quy trình kiểm tra được giám sát, tiến độ và hiệu quả kiểm tra được kiểm soát chặt chẽ. Mục tiêu của mức 3 bao gồm:**

**Thiết lập bộ phận KTPM**

**Thiết lập chương trình huấn luyện kỹ thuật**

**Tích hợp KTPM vào chu kỳ PTPM**

**Kiểm soát và giám sát quy trình kiểm tra**

### 7.2.3. So sánh mức 3 giữa TMM và CMM:

TMM Mức 3: Tích hợp	CMM Mức 3: Được định nghĩa
<ul style="list-style-type: none"> <li>• Kiểm soát và giám sát quy trình kiểm tra</li> <li>• Tích hợp kiểm tra phần mềm</li> <li>• Thiết lập chương trình huấn luyện kỹ thuật</li> <li>• Thiết lập tổ chức kiểm tra phần mềm</li> </ul>	<ul style="list-style-type: none"> <li>• Tập trung quy trình cấp tổ chức</li> <li>• Định nghĩa quy trình cấp tổ chức</li> <li>• Chương trình huấn luyện</li> <li>• Tích hợp quản lý phần mềm</li> <li>• Kỹ thuật phát triển sản phẩm</li> <li>• Điều phối liên nhóm</li> <li>• Xem xét ngang hàng</li> </ul>

#### Mức trưởng thành 4: Quản lý và đo lường

Một chương trình xem xét cấp công ty được thành lập với mục tiêu loại bỏ sai sót trong sản phẩm kể cả sản phẩm trung gian bằng kỹ thuật xem xét ngang hàng (peer review – kỹ thuật phổ biến để phát hiện lỗi sớm trên các sản phẩm và sản phẩm trung gian không thi hành được như yêu cầu khách hàng, bản thiết kế, mã nguồn, kế hoạch kiểm tra... được thực hiện bởi một nhóm người cùng làm việc).

Quy trình kiểm tra là một quy trình định lượng. Các chỉ số liên quan đến KTPM được định nghĩa và thu thập nhằm phân tích, khảo sát chất lượng và hiệu quả của quy trình kiểm tra. Một số ví dụ về các chỉ số này như: tỷ lệ lỗi trên một đơn vị kích thước PM, số lượng lỗi do kiểm tra viên tìm thấy trên tổng số lỗi của PM (bao gồm lỗi do khách hàng phát hiện), thời gian trung bình để sửa chữa một lỗi... Mục tiêu của mức 4 bao gồm:

- Thiết lập chương trình xem xét xuyên suốt các dự án trong công ty
- Thiết lập chương trình đo lường việc KTPM
- Đánh giá chất lượng PM

#### So sánh mức 4 giữa TMM và CMM

TMM Mức 4: Quản lý và đo lường	CMM Mức 4: Được quản lý
<ul style="list-style-type: none"> <li>• Đánh giá chất lượng phần mềm</li> <li>• Đo lường việc kiểm tra phần mềm</li> <li>• Chương trình xem xét xuyên dự án</li> </ul>	<ul style="list-style-type: none"> <li>• Quản lý quy trình theo lượng hóa</li> <li>• Quản lý chất lượng phần mềm</li> </ul>

#### Mức trưởng thành 5: Tối ưu hóa, phòng ngừa lỗi và kiểm soát chất lượng

Dữ liệu liên quan đến các sai sót đã thu thập (ở mức 4) được phân tích để tìm ra nguyên nhân gốc phát sinh các sai sót đó. Căn cứ vào các nguyên nhân này, hành động phòng ngừa được thiết lập và thi hành. Các phép thống kê được dùng để ước

lượng tính tin cậy của phần mềm, cũng như làm cơ sở cho các quyết định liên quan đến xác định các mục tiêu về độ tin cậy của phần mềm. Chi phí và tính hiệu quả của KTPM được giám sát chặt chẽ, công cụ kiểm tra tự động được sử dụng rộng rãi.

Mặt khác, ở mức 5, quy trình KTPM phải được cải tiến một cách liên tục, nhằm khắc phục những yếu kém của quy trình, cũng như hướng đến những mục tiêu xa hơn. Mục tiêu của mức 5 bao gồm:

- Sử dụng dữ liệu thu thập để phòng ngừa sai sót.
- Kiểm soát chất lượng
- Tối ưu hóa quy trình KTPM

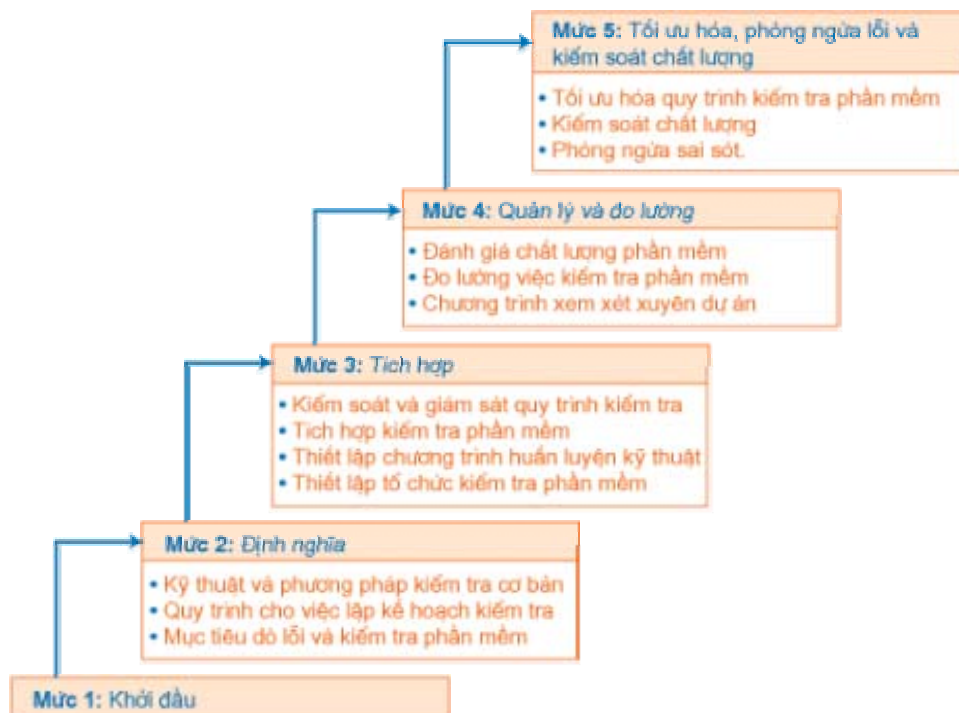
#### So sánh mức 5 giữa TMM và CMM:

TMM Mức 5: Tối ưu hóa, phòng ngừa lỗi và kiểm soát chất lượng	CMM Mức 5: Tối ưu hóa
<ul style="list-style-type: none"> <li>• Phòng ngừa sai sót.</li> <li>• Kiểm soát chất lượng</li> <li>• Tối ưu hóa quy trình kiểm tra phần mềm</li> </ul>	<ul style="list-style-type: none"> <li>• Phòng ngừa sai sót.</li> <li>• Quản lý thay đổi kỹ thuật/công nghệ</li> <li>• Quản lý thay đổi quy trình</li> </ul>

#### Tóm lại

KTPM là một lĩnh vực rất quan trọng trong hoạt động sản xuất cũng như gia công PM. Các mức kiểm tra và loại kiểm tra rất phong phú, phục vụ mục tiêu đảm bảo chất lượng toàn diện cho một PM hoặc một hệ thống. Trong thực tế, để triển khai tất cả các mức và loại kiểm tra đã liệt kê cho một dự án PM đòi hỏi sự đầu tư rất lớn cả về thời gian lẫn công sức. Các tổ chức "còn non" trong quy trình kiểm tra thường cố gắng tiết kiệm tối đa đầu tư vào KTPM, thường lơ việc lập kế hoạch kiểm tra đến khi hoàn thành việc viết code, bỏ qua một vài hoặc hầu hết các chặng kiểm tra. PM giao cho khách hàng trong điều kiện như thế thường nghèo nàn về chất lượng. Kết quả thường là sự đột biến về chi phí bỏ ra cho việc sửa chữa lỗi, hoặc bảo trì PM, tuy nhiên sự mất mát lớn nhất là sự thất vọng của khách hàng hoặc những người dùng cuối.





**Hình 08:** 5 mức độ trưởng thành trong TMM

### 7.3. Các công cụ kiểm thử (Test tools)

Ngày nay tự động hóa được ứng dụng ở rất nhiều lĩnh vực, mục đích thường rất đa dạng và tùy theo nhu cầu đặc thù của từng lĩnh vực, tuy nhiên điểm chung nhất vẫn là giảm nhân lực, thời gian và sai sót. Ngành CNTT mà cụ thể là phát triển phần mềm (PTPM) cũng không ngoại lệ. Như chúng ta biết, để tạo ra sản phẩm CNTT hay PM có chất lượng thì hoạt động kiểm thử phần mềm (KTPM) đóng vai trò rất quan trọng, trong khi đó hoạt động này lại tiêu tốn và chiếm tỷ trọng khá lớn công sức và thời gian trong một dự án. Do vậy, nhu cầu tự động hoá qui trình KTPM cũng được đặt ra.

Qua thực tế cho thấy việc áp dụng kiểm thử tự động (KTTĐ) hợp lý sẽ mang lại thành công cho hoạt động KTPM. KTTĐ giúp giảm bớt công sức thực hiện, tăng độ tin cậy, giảm sự nhầm lẫn và rèn luyện kỹ năng lập trình cho kiểm thử viên (KTV). Bài viết này sẽ giới thiệu các khái niệm cơ bản của KTTĐ, đồng thời giới thiệu một công cụ KTTĐ khá mạnh hiện nay là QuickTest Professional 8.2 (QTP) của Mercury.

#### 7.3.1. TẠI SAO PHẢI DÙNG TEST TOOL

Test Tool (TT) trong lĩnh vực PTPM là công cụ giúp thực hiện việc kiểm thử PM một cách tự động. Tuy nhiên không phải mọi việc kiểm thử đều có thể tự động hóa, câu hỏi đặt ra là trong điều kiện hoặc tình huống nào dùng TT là thích hợp? Việc dùng TT thường được xem xét trong một số tình huống sau:

##### 1. Không đủ tài nguyên

Khi số lượng tình huống kiểm thử (test case) quá nhiều mà các KTV không thể hoàn tất bằng tay trong thời gian cụ thể nào đó.

Có thể lấy một dẫn chứng là khi thực hiện kiểm thử chức năng của một website. Website này sẽ được kiểm thử với 6 môi trường gồm 3 trình duyệt và 2 hệ điều hành

Tình huống này đòi hỏi số lần kiểm thử tăng lên và lặp lại 6 lần so với việc kiểm thử cho một môi trường cụ thể.

### Kiểm thử hồi qui

Trong quá trình PTPM, nhóm lập trình thường đưa ra nhiều phiên bản PM liên tiếp để kiểm thử. Thực tế cho thấy việc đưa ra các phiên bản PM có thể là hàng ngày, mỗi phiên bản bao gồm những tính năng mới, hoặc tính năng cũ được sửa lỗi hay nâng cấp. Việc bổ sung hoặc sửa lỗi code cho những tính năng ở phiên bản mới có thể làm cho những tính năng khác đã kiểm thử tốt chạy sai mặc dù phần code của nó không hề chỉnh sửa. Để khắc phục điều này, đối với từng phiên bản, KTV không chỉ kiểm tra chức năng mới hoặc được sửa, mà phải kiểm tra lại tất cả những tính năng đã kiểm tra tốt trước đó. Điều này khó khả thi về mặt thời gian nếu kiểm tra thủ công.

x WinXP, IE WinXP, Netscape WinXP, Opera Linux, IE Linux, Netscape Linux, Opera

### 2. Kiểm tra khả năng vận hành PM trong môi trường đặt biệt

Đây là kiểm tra nhằm đánh giá xem vận hành của PM có thỏa mãn yêu cầu đặt ra hay không. Thông qua đó KTV có thể xác định được các yếu tố về phần cứng, phần mềm ảnh hưởng đến khả năng vận hành của PM. Có thể liệt kê một số tình huống kiểm thử tiêu biểu thuộc loại này như sau:

- Đo tốc độ trung bình xử lý một yêu cầu của web server.
- Thiết lập 1000 yêu cầu, đồng thời gửi đến web server, kiểm tra tình huống 1000 người dùng truy xuất web cùng lúc.
- Xác định số yêu cầu tối đa được xử lý bởi web server hoặc xác định cấu hình máy thấp nhất mà tốc độ xử lý của PM vẫn có thể hoạt động ở mức cho phép.

Việc kiểm tra thủ công cho những tình huống trên là cực khó, thậm chí "vô phương".

Cần lưu ý là hoạt động KTTĐ nhằm mục đích kiểm tra, phát hiện những lỗi của PM trong những trường hợp đoán trước. Điều này cũng có nghĩa là nó thường được thực hiện sau khi đã thiết kế xong các tình huống (test case). Tuy nhiên, như đã nói, không phải mọi trường hợp kiểm tra đều có thể hoặc cần thiết phải tự động hóa, trong tất cả test case thì KTV phải đánh giá và chọn ra những test case nào phù hợp hoặc cần thiết để áp dụng KTTĐ dựa trên những tiêu chí đã đề cập bên trên.

### 7.3.2. KHÁI QUÁT VỀ KTTĐ

Việc phát triển KTTĐ cũng tuân theo các bước PTPM, chúng ta phải xem việc phát triển KTTĐ giống như phát triển một dự án. Bạn đọc có thể tham khảo bài viết về kiểm tra phần mềm trên TGVN A tháng 12/2005 (ID: A0512\_110). Hình 1 cho chúng ta thấy mối tương quan giữa KTTĐ và toàn bộ chu trình KTPM.



Hình 1

Giống như PTPM, để thành công trong KTTĐ chúng ta nên thực hiện các bước cơ bản sau:

- Thu thập các đặc tả yêu cầu hoặc test case; lựa chọn những phần cần thực hiện KTTĐ.
- Phân tích và thiết kế mô hình phát triển KTTĐ.
- Phát triển lệnh đặc tả (script) cho KTTĐ.
- Kiểm tra và theo dõi lỗi trong script của KTTĐ.

Bảng sau mô tả rõ hơn các bước thực hiện KTTĐ:

## 1

### Tạo test script

Giai đoạn này chúng ta sẽ dùng test tool để ghi lại các thao tác lên PM cần kiểm tra và tự động sinh ra test script.

## 2

### Chỉnh sửa test script

Chỉnh sửa để test script thực hiện kiểm tra theo đúng yêu cầu đặt ra, cụ thể là làm theo test case cần thực hiện.

## 3

### Chạy test script để KTTĐ

Giám sát hoạt động kiểm tra PM của test script.

## 4

### Đánh giá kết quả

Kiểm tra kết quả thông báo sau khi thực hiện KTTĐ. Sau đó bổ sung, chỉnh sửa những sai sót.

**KTTĐ có một số thuận lợi và khó khăn cơ bản khi áp dụng:**

- KTPM không cần can thiệp của KTV.
- Giảm chi phí khi thực hiện kiểm tra số lượng lớn test case hoặc test case lặp lại nhiều lần.
- Giảm lập tình huống khó có thể thực hiện bằng tay.
- Mất chi phí tạo các script để thực hiện KTTĐ.
- Tốn chi phí dành cho bảo trì các script.
- Đòi hỏi KTV phải có kỹ năng tạo script KTTĐ.
- Không áp dụng được trong việc tìm lỗi mới của PM.

### 7.3.3. GIỚI THIỆU CÔNG CỤ KTTĐ: QUICKTEST PROFESSIONAL

Trong lĩnh vực KTTĐ hiện có khá nhiều TT thương mại nổi tiếng, phổ biến như QuickTest Professional, WinRunner, Rational Robot, SilkTest, JTest,... Trong số đó, QuickTest Professional (QTP) phiên bản 8.2 của hãng Mercury khá tốt và mạnh, bao gồm nhiều chức năng điển hình của một công cụ kiểm tra tự động. Lưu ý là QTP 8.2 đã có một cái tên mới hơn là Mercury Functional Testing 8.2.

QTP là TT dùng để kiểm tra chức năng (functional test) và cho phép thực hiện kiểm tra hồi qui (regression test) một cách tự động. Đây cũng là công cụ áp dụng phương pháp Keyword-Driven, một kỹ thuật scripting (lập trình trong KTTĐ) hiện đại, cho phép KTV bổ sung test case bằng cách tạo file mô tả cho nó mà không cần phải chỉnh sửa hay bổ sung bất cứ script nào cả. Nó cũng phù hợp trong tình huống chuyển giao công việc mà người mới tiếp nhận chưa có thời gian hoặc không hiểu script vẫn có thể thực hiện kiểm tra PM theo đúng yêu cầu.

#### 1. Loại phần mềm hỗ trợ

QTP giúp chúng ta KTPM theo hướng chức năng trên rất nhiều loại chương trình phần mềm khác nhau. Tuy nhiên Mercury chỉ hỗ trợ sẵn một số loại chương trình thông dụng như:

- Ứng dụng Windows chuẩn/Win32.
- Ứng dụng web theo chuẩn HTML, XML chạy trong trình duyệt Internet Explorer, Netscape hoặc AOL.
- Visual Basic.
- ActiveX.
- QTP hỗ trợ Unicode (UTF-8, UTF-16).

Một số loại chương trình khác đòi hỏi chúng ta phải cài đặt thêm thành phần bổ sung của QTP thì mới thực hiện kiểm tra được. Các loại chương trình đó là:

#### **.NET**

- NET Framework 1.0, 1.1, 2.0 beta
- Các đối tượng chuẩn của .NET và các đối tượng khác thừa kế từ các đối tượng chuẩn.

#### **Java**

- Sun JDK 1.1 – 1.4.2
- IBM JDK 1.2 – 1.4

#### **Oracle**

- Oracle Applications 11.5.7, 11.5.8, 11.5.9

#### **People Soft**

- PeopleSoft Enterprise 8.0 – 8.8

#### **SAP**

- SAP GUI HMTL 4.6D, 6.10, 6.20

## **Đặc điểm**

- Dễ sử dụng, bảo trì, tạo test script nhanh. Cung cấp dữ liệu kiểm tra rõ ràng và dễ hiểu.
- Kiểm tra phiên bản mới của ứng dụng với rất ít sự thay đổi. Ví dụ khi ứng dụng thay đổi nút tên "Login" thành "Đăng nhập", thì chỉ cần cập nhật lại Object Repository (OR - được giải thích ở phần sau) để QTP nhận ra sự thay đổi đó mà không cần thay đổi bất cứ test script nào.
- Hỗ trợ làm việc theo nhóm thông qua sự chia sẻ thư viện, thống nhất quản lý Object Repository.
- Thực tế cho thấy, QTP thực hiện KTTĐ trên nhiều trình duyệt cùng lúc tốt hơn những TT khác.
- Với chức năng Recovery Scenarios, QTP cho phép xử lý những sự kiện hoặc lỗi không thể đoán trước có thể làm script bị dừng trong khi đang chạy.
- QTP có khả năng hiểu test script của Mercury Winrunner (một công cụ kiểm tra khác của Mercury).

**Đặc biệt phiên bản v.8.2 có một số tính năng mới nổi bật:**

### ***Quản trị Object Repository***

- Phối hợp giữa các KTV qua việc đồng bộ hóa dữ liệu, khả năng trộn, nhập/xuất ra file XML

### ***Thư viện hàm mới***

- Chia sẻ các thư viện hàm giữa các nhóm KTV

### ***Kiểm tra tài nguyên***

- Kiểm tra tài nguyên cần thiết trước khi thực thi lệnh kiểm tra tự động.

### ***Nâng cấp khả năng kéo thả***

- Kéo thả các bước kiểm tra trong môi trường ngôn ngữ tự nhiên.

### ***Hỗ trợ XML cho báo cáo***

- Lưu trữ kết quả kiểm tra dưới dạng XML, HTML, từ đó cho phép tùy biến báo cáo.

### ***Trình phát triển mới (IDE)***

- Môi trường soạn thảo mới, mềm dẻo cho tùy biến và sử dụng.

### ***Trình dò lỗi mới***

- Cho phép KTV kiểm soát lỗi khi viết test case.

### ***Quản trị từ khóa***

- Quản lý từ khóa trong quá trình sử dụng

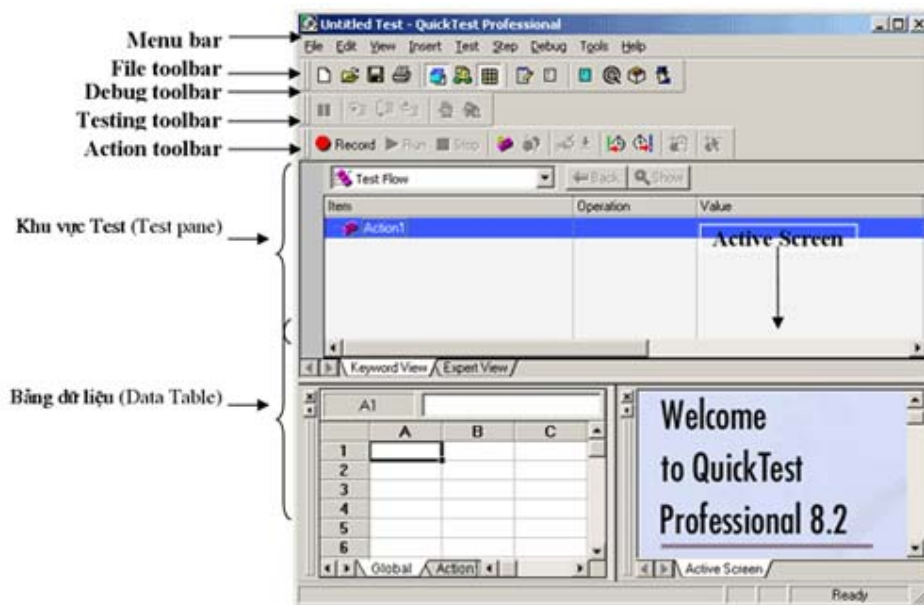
### ***Hỗ trợ đa giao tiếp***

- Cho phép người dùng mở và soạn thảo đồng thời nhiều hàm thư viện và Object Repository.

### ***Hỗ trợ Unicode***

- Hỗ trợ Unicode với các ứng dụng đa ngôn ngữ (multi-language).

### ***Hỗ trợ các môi trường mới***



Menu bar Cấu hình thao tác với QTP và script File toolbar Hỗ trợ quản lý script Debug toolbar Hỗ trợ kiểm tra lỗi trong test script (debug) Testing toolbar Hỗ trợ quá trình tạo test script hoặc thực hiện KTTĐ Action toolbar Xem một Action (thủ tục, hàm) hoặc toàn bộ chu trình của test script Test pane Soạn thảo script ở một trong 2 chế độ Keyword View hoặc Expert View Data Table Nơi lưu trữ dữ liệu cho test script Active Screen Xem lại giao diện PM được kiểm tra

### 3. Các thành phần quan trọng trong QTP

#### a. Action:

Giống như thủ tục hay hàm trong các ngôn ngữ lập trình khác, Action ghi lại các bước thực hiện KTTĐ và nó có thể được sử dụng lại nhiều lần. Trong một test script có thể có nhiều Action.

#### b. DataTable:

Nơi lưu trữ dữ liệu phục vụ cho KTTĐ. Một test script sẽ có một DataTable được dùng chung cho tất cả các Action. Bên cạnh đó mỗi Action cũng có một DataTable cho riêng mình.

#### c. Object Repository (OR):

Cấu trúc theo dạng cây, mô tả các đối tượng trong PM được kiểm tra. Đây được xem là cầu nối để test script tương tác với PM được kiểm tra.

Khi ra lệnh cho QTP ghi lại thao tác người dùng lên PM thì trong OR sẽ tự động phát sinh thành phần đại diện cho những đối tượng trên PM vừa được thao tác.

OR có thể tổ chức thành 2 loại, một loại dùng chung trong nhiều test script, loại

khác dùng theo từng Action.

Để xem OR, chọn menu Tools > Object Repository.

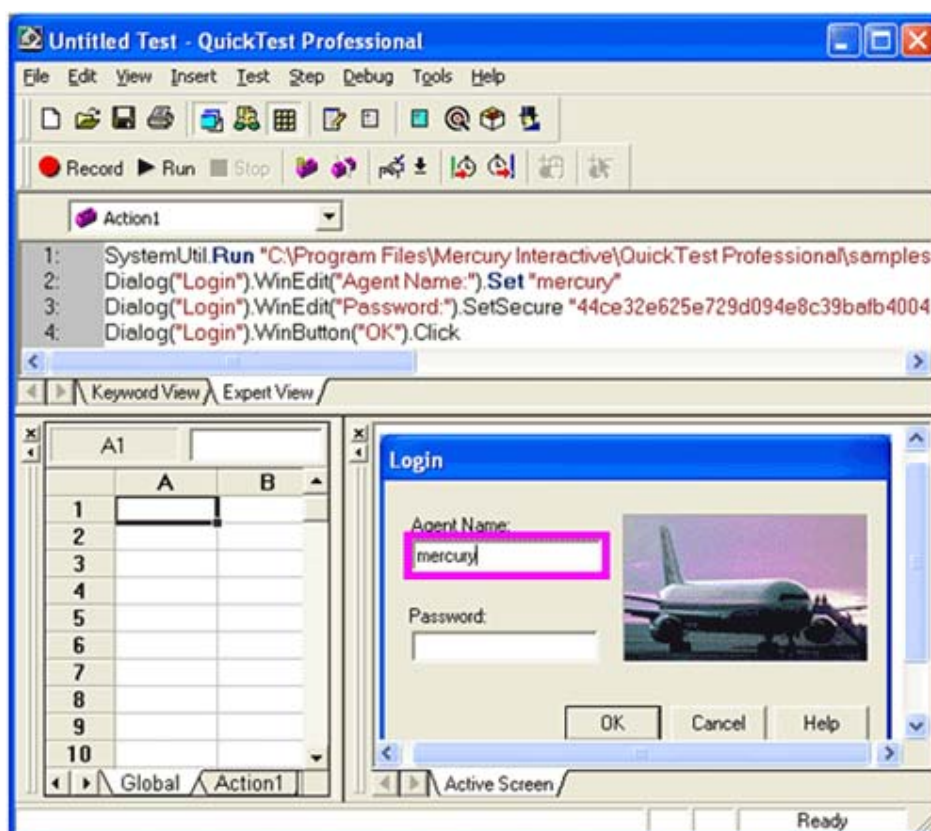
#### d. Checkpoint:

Có thể hiểu là nơi kiểm tra trong test script, khi chạy nó sẽ thực hiện so sánh kết quả thực tế khi kiểm tra PM với kết quả mong đợi. Sau khi tiến hành so sánh QTP sẽ tự động ghi lại kết quả vào Test Results (nơi lưu kết quả khi chạy test script).

## 4. Ngôn ngữ sử dụng viết script

QTP sử dụng ngôn ngữ VBScript để viết test script. Đây là ngôn ngữ dễ học; rất giống ngôn ngữ VBA. Chế độ Expert View của QTP là chế độ soạn thảo dành cho VBScript. Ngoài việc dùng VBScript để tương tác với PM được kiểm tra, QTP còn có khả năng cấu hình hệ thống bằng ngôn ngữ Windows Script.

Chi tiết về ngôn ngữ VBScript, người đọc có thể dễ dàng tìm trong các sách hiện có trên thị trường, thậm chí ngay chính trong phần help của QTP.



## 6. Sử Dụng QTP

a. Yêu cầu cấu hình hệ thống:

b. Bản quyền sử dụng:



Hệ điều hành Windows 2000 SP3, SP4; Windows XP SP1, SP2 hoặc Windows 2003 Server RAM 256 MB trở lên Dung lượng đĩa Tối thiểu 250MB cho ứng dụng, 120MB trên ổ đĩa hệ điều hành. Sau khi cài QTP, dung lượng cần thiết thêm trên ổ đĩa cài hệ điều hành là 150 MB Trình duyệt IE 5.5 SP 2 trở lên • Bạn có thể vào <http://www.mercury.com> để đăng ký và tải về bản dùng thử trong 14 ngày. Các bước cài đặt theo sự hướng dẫn của chương trình. Sau thời gian dùng thử, để có thể tiếp tục sử dụng QTP chúng ta cần phải mua bản quyền, giá tham khảo từ nhà cung cấp như sau: cho một máy 9.000 USD; cho nhiều máy dùng cùng lúc 12.000 USD.

#### 7.3.4. Kiểm thử đơn vị với JUnit

##### Định nghĩa

JUnit là một framework đơn giản dùng cho việc tạo các unit testing tự động, và chạy các test có thể lặp đi lặp lại. Nó chỉ là một phần của họ kiến trúc XUnit cho việc tạo các Unit Testing. JUnit là một chuẩn trên thực tế cho Unit Testing trong Java.

##### Đặc điểm

JUnit là công cụ giúp ta thử nghiệm, gỡ rối chương trình Java. Với JUnit, bạn dễ dàng theo dõi diễn biến của chương trình, nhanh chóng dàn dựng hàng loạt phép thử (test case) để kiểm tra xem mọi việc có xảy ra đúng như dự định hay không. Thuở ban đầu, JUnit được xây dựng bởi Kent Beck và Erich Gamma. Sau đó, giới lập trình viên Java biến JUnit thành một đề án nguồn mở. Ngày nay, JUnit trở thành một thứ "công cụ chuẩn" mà mọi lập trình viên Java đều nên biết cách dùng.

JUnit là một mã nguồn mở, regression-testing framework nhằm giúp cho các java developer viết những unit test để kiểm tra từng modul của project khi phát triển hệ thống. Framework giúp đỡ trong việc thiết lập một close-relationship giữa testing và development. Đầu tiên bạn viết ra các đoạn code của bạn sẽ làm việc. Sau đó bạn viết code và dùng JUnit test runner để kiểm tra xem nó bị chệch hướng so với dự định ban đầu như thế nào. Intergration testing xác nhận rằng những hệ thống con khác nhau đó sẽ làm việc tốt khi kết hợp chúng với nhau. Acceptance testing đơn giản xác nhận chính xác rằng một ứng dụng có làm việc đúng như khách hàng mong đợi không. Unit test được gọi như vậy bởi vì họ test từng đoạn code đơn lẻ một, nó có thể chỉ là một class đơn trong java.

Khác với các Unit Test đặc thù, ở đó bạn có khuynh hướng rằng sẽ viết test sau khi hoàn thành module, JUnit khuyến khích bạn kết hợp coding và testing trong suốt quá trình phát triển. Kể từ đây, mục đích chính là kiểm tra module ở mức nhỏ là kiểm tra các chức năng, và hơn là kiểm tra các khối cơ bản của hệ thống tại một thời điểm nào đó. Đề án này hướng dẫn việc phát triển một bộ test toàn diện mà bạn có thể dùng bất cứ khi nào sau khi thay đổi một đoạn code và tin tưởng rằng sản phẩm hoặc đoạn code sửa đổi đó không phá vỡ những hệ thống con khách mà bạn không được biết.



Một số đặc điểm quan trọng của Junit:

- Xác nhận (assert) việc kiểm tra kết quả được mong đợi.
- Các Test Suite cho phép chúng ta dễ dàng tổ chức và chạy các test.
- Hỗ trợ giao diện đồ họa và giao diện dòng lệnh.

Các test case của JUnit là các lớp của Java, các lớp này bao gồm một hay nhiều các phương thức unit testing, và những test này lại được nhóm thành các Test Suite.

Mỗi phương thức test trong JUnit phải được thực thi nhanh chóng. Tốc độ là điều tối quan trọng vì càng nhiều test được viết và tích hợp vào bên trong quá trình xây dựng phần mềm, cần phải tốn nhiều thời gian hơn cho việc chạy toàn bộ Test Suite. Các lập trình viên không muốn bị ngắt quãng trong một khoảng thời gian dài trong khi các test chạy, vì thế các test mà chạy càng lâu thì sẽ có nhiều khả năng là các lập trình viên sẽ bỏ qua bước cũng không kém phần quan trọng này.

Các test trong JUnit có thể là các test được chấp nhận hay thất bại, các test này được thiết kế để khi chạy mà không cần có sự can thiệp của con người. Từ những thiết kế như thế, bạn có thể thêm các bộ test vào quá trình tích hợp và xây dựng phần mềm một cách liên tục và để cho các test chạy một cách tự động.

### Một số phương thức trong Junit

#### Các phương thức assertXXX()

Các phương thức assertXXX() được dùng để kiểm tra các điều kiện khác nhau. junit.framework.TestCase, lớp cha cho tất cả các test case, thừa kế từ lớp junit.framework.Assert. Lớp này định nghĩa khá nhiều các phương thức assertXXX(). Các phương thức test hoạt động bằng cách gọi những phương thức này.

Một số mô tả các phương thức assertXXX() khác nhau có trong lớp junit.framework.

❖ *assert:assertEquals(): So sánh 2 giá trị để kiểm tra bằng nhau. Test sẽ được chấp nhận nếu các giá trị bằng nhau*

❖ *assertFalse(): Đánh giá biểu thức luận lý. Test sẽ được chấp nhận nếu biểu thức sai*  
*assertNotNull(): So sánh tham chiếu của một đối tượng với null. Test sẽ được chấp nhận nếu tham chiếu đối tượng khác null*

❖ *assertNotSame(): So sánh địa chỉ vùng nhớ của 2 tham chiếu đối tượng bằng cách sử dụng toán tử ==. Test sẽ được chấp nhận nếu cả 2 đều tham chiếu đến các đối tượng khác nhau*

❖ *assertNull(): So sánh tham chiếu của một đối tượng với giá trị null. Test sẽ được chấp nhận nếu tham chiếu là null*

❖ *assertSame(): So sánh địa chỉ vùng nhớ của 2 tham chiếu đối tượng bằng cách sử dụng toán tử ==. Test sẽ được chấp nhận nếu cả 2 đều tham chiếu đến cùng một đối tượng*

❖ *assertTrue(): Đánh giá một biểu thức luận lý. Test sẽ được chấp nhận nếu biểu thức đúng*

❖ *fail(): Phương thức này làm cho test hiện hành thất bại, phương thức này thường được sử dụng khi xử lý các biệt lệ*

Mặc dù bạn có thể chỉ cần sử dụng phương thức assertTrue() cho gần như hầu hết các test, tuy nhiên thì việc sử dụng một trong các phương thức assertXXX() cụ thể sẽ làm cho các test của bạn dễ hiểu hơn và cung cấp các thông điệp thất bại rõ ràng hơn.

Tất cả các phương thức trên đều nhận vào một String không bắt buộc làm tham số đầu tiên. Khi được xác định, tham số này cung cấp một thông điệp mô tả test thất bại.

Ví dụ:

```
assertEquals(employeeA, employeeB);  
assertEquals("Employees should be equal after the clone() operation.", employeeA,  
employeeB).
```

### .Set Up và Tear Down

Hai phương thức `setUp()` và `tearDown()` là một phần của lớp `junit.framework.TestCase` Bằng cách sử dụng các phương thức `setUp` và `tearDown`. Khi sử dụng 2 phương thức `setUp()` và `tearDown()` 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 thủ theo một dãy có thứ tự các sự kiện khi chạy các test. Đầu tiên, nó tạo ra một thể hiện mới của test case ứng với mỗi phương thức test. Từ đó, nếu bạn có 5 phương thức test thì JUnit sẽ tạo ra 5 thể hiện của test case. Vì lý do đó, các biến thể hiện không thể được sử dụng để chia sẻ trạng thái giữa các phương thức test. Sau khi tạo xong tất cả các đối tượng test case, JUnit tuân theo các bước sau cho mỗi phương thức test:

- ❖ Gọi phương thức `setUp()` của test case
- ❖ Gọi phương thức test
- ❖ Gọi phương thức `tearDown()` của test case

Quá trình này được lặp lại đối với mỗi phương thức test trong test case.

Sau đây chúng ta sẽ xem xét 1 ví dụ tính cộng, trừ và nhân hai số nguyên:

### CODE

```
public class CongTruNhan {  
  
    int add (int x, int y){  
        return (x + y);  
    }  
  
    int multiply (int x, int y){  
        return (x*y);  
    }  
  
    int subtract (int x, int y){  
        return (x-y);  
    }  
}
```

Mã Code TestCase của lớp trên có sử dụng phương thức `setUp()` và `tearDown()`

## CODE

```
import junit.framework.*;
public class Test CongTruNhan extends TestCase {
    public Test CongTruNhan (String name) {
        super(name);
    }
    // Initialize common test data
    int x;
    int y;
    protected void setUp() {
        System.out.println("setUp - Intialize common test data");
        x = 7;
        y = 5;
    }
    protected void tearDown(){
        System.out.println("tearDown - Clean up");
    }
    /**
     * Test of add method, of class CongTruNhan.
     */
    public void testAdd() {
        System.out.println("add");

        CongTruNhan instance = new CongTruNhan ();

        int expResult = 12;
        int result = instance.add(x, y);
        assertEquals(expResult, result);

    }

    /**
     * Test of multiply method, of class CongTruNhan.
     */
    public void testMultiply() {
        System.out.println("multiply");

        CongTruNhan instance = new CongTruNhan ();

        int expResult = 35;
        int result = instance.multiply(x, y);
        assertEquals(expResult, result);

    }

    /**
     * Test of subtract method, of class CongTruNhan.
     */
}
```

```

    */
    public void testSubtract() {
        System.out.println("subtract");

        CongTruNhan instance = new CongTruNhan();

        int expResult = 2;
        int result = instance.subtract(x, y);
        assertEquals(expResult, result);
    }

    public static void main(String[] args){

        System.out.println("Running the test using junit.textui.TestRunner.run() method...");
        junit.textui.TestRunner.run(TestCongTruNhan.class);

    }
}

```

Thông thường chúng có thể bỏ qua phương thức `tearDown()` vì mỗi unit test riêng không phải là những tiến trình chạy tốn nhiều thời gian, và các đối tượng được thu dọn khi JVM thoát. `tearDown()` có thể được sử dụng khi test của bạn thực hiện những thao tác như mở kết nối đến cơ sở dữ liệu hay sử dụng các loại tài nguyên khác của hệ thống và bạn cần phải dọn dẹp ngay lập tức. Nếu chúng chạy một bộ bao gồm một số lượng lớn các unit test, thì khi bạn trở tham chiếu của các đối tượng đến null bên trong thân phương thức `tearDown()` sẽ giúp cho bộ dọn rác lấy lại bộ nhớ khi các test khác chạy

Đôi khi bạn muốn chạy vài đoạn mã khởi tạo chỉ một lần, sau đó chạy các phương thức test, và bạn chỉ muốn chạy các đoạn mã dọn dẹp chỉ sau khi tất cả test kết thúc. Ở phần trên, JUnit gọi phương thức `setUp()` trước mỗi test và gọi `tearDown()` sau khi mỗi test kết thúc, vì thế để làm được điều như trên, chúng ta sẽ sử dụng lớp `junit.extensions.TestSetup` để đạt được yêu cầu trên.

Ví dụ sau sẽ minh họa việc sử dụng lớp trên

## CODE

```

import junit.extensions.TestSetup;
import junit.framework.*;

public class TestPerson extends TestCase {
    public TestPerson(String name)
    { super(name); }

    public void testGetFullName() {
        Person p = new Person("Aidan", "Burke");
        assertEquals("Aidan Burke", p.getFullName());
    }

    public void testNullsInName() {
        Person p = new Person(null, "Burke");
    }
}

```

```

assertEquals("? Burke", p.getFullName());
p = new Person("Tanner", null);
assertEquals("Tanner ?", p.getFullName());
}

public static Test suite() {
    TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {
        protected void setUp() throws Exception
        {
            //Thực hiện các đoạn mã khởi tạo một lần ở đây
        }

        protected void tearDown() throws Exception {
            //Thực hiện các đoạn mã dọn dẹp ở đây
        }
    };
    return setup;
}
}

```

**TestSetup** là một lớp thừa kế từ lớp `junit.extension.TestDecorator`, Lớp `TestDecorator` là lớp cơ sở cho việc định nghĩa các test biến thể. Lý do chính để mở rộng `TestDecorator` là để có được khả năng thực thi đoạn mã trước và sau khi một test chạy. Các phương thức `setUp()` và `tearDown()` của lớp `TestSetup` được gọi trước và sau khi bất kỳ Test nào được truyền vào constructor,

Trong ví dụ trên chúng ta đã truyền một tham số có kiểu `TestSuite` vào constructor của lớp `TestSetup`

```
TestSetup setup = new TestSetup(new TestSuite(TestPerson.class))
```

Điều này có nghĩa là 2 phương thức `setUp()` được gọi chỉ một lần trước toàn bộ bộ test và `tearDown()` được gọi chỉ một lần sau khi các test trong bộ test kết thúc.

Chú ý: các phương thức `setUp()` và `tearDown()` bên trong lớp `TestPerson` vẫn được thực thi trước và sau mỗi phương thức test bên trong lớp `TestPerson`.

### Tạo test class và tạo bộ test

#### Test các chức năng của một lớp

Bạn muốn viết các unit test với JUnit. Việc đầu tiên bạn phải tạo một lớp con thừa kế từ lớp `junit.framework.TestCase`. Mỗi unit test được đại diện bởi một phương thức `testXXX()` bên trong lớp con của lớp `TestCase`

Ta có một lớp `Person` như sau:

#### **CODE**

```

public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        if (firstName == null && lastName == null) {
            throw new IllegalArgumentException("Both names cannot be null");
        }
        this.firstName = firstName;
    }
}

```

```

this.lastName = lastName;
}

public String getFullName() {
String first = (this.firstName != null) ? this.firstName : "?";
String last = (this.lastName != null) ? this.lastName : "?";
return first + last;
}

public String getFirstName() {
return this.firstName;
}

public String getLastName() {
return this.lastName;
}
}

```

Sau đó ta sẽ viết một test case đơn giản để test một số phương thức của lớp trên:

#### CODE

```

import junit.framework.TestCase;
public class TestPerson extends TestCase
{
public TestPerson(String name) {
super(name);
}

/**
 *
 * Xac nhan rang name duoc the hien dung dinh dang
 */
public void testGetFullName() {
Person p = new Person("Aidan", "Burke");
assertEquals("Aidan Burke", p.getFullName());
}

/**
 * Xac nhan rang nulls da duoc xu ly chinh xac
 */
public void testNullsInName() {
Person p = new Person(null, "Burke");
assertEquals("? Burke", p.getFullName());
p = new Person("Tanner", null);
assertEquals("Tanner ?", p.getFullName());
}
}

```

**Lưu ý:** mỗi unit test là một phương thức public và không có tham số được bắt đầu bằng tiếp đầu ngữ test. Nếu bạn không tuân theo quy tắc đặt tên này thì JUnit sẽ không xác định được các phương thức test một cách tự động.

Để biên dịch TestPerson, chúng ta phải khai báo gói thư viện junit trong biến đường môi trường classpath

```
set classpath=%classpath%;.;junit.jar
javac TestPerson
```

### Tổ chức các test vào các test suite

Thông thường JUnit tự động tạo ra các Test Suite ứng với mỗi Test Case. Tuy nhiên bạn muốn tự tạo các Test Suite của riêng mình bằng cách tổ chức các Test vào Test Suite. JUnit cung cấp lớp junit.framework.TestSuite hỗ trợ việc tạo các Test Suite

Khi bạn sử dụng giao diện text hay graphic, JUnit sẽ tìm phương thức sau trong test case của bạn trên, JUnit sẽ sử dụng kỹ thuật reflection để tự động xác định tất cả các phương

```
public static Test suite() { ... }
```

Nếu không thấy phương thức testXXX() trong test case của bạn, rồi thêm chúng vào một test suite. Sau đó nó sẽ chạy tất cả các test trong suite này. Bạn có thể tạo ra bản sao hành vi của phương thức suite() mặc định như sau:

```
public class TestGame extends TestCase{
...
public static Test suite() {
return new TestSuite(TestGame.class);
}
}
```

Bằng cách truyền đối tượng TestGame.class vào construtor TestSuite, bạn đang thông báo cho JUnit biết để xác định tất cả các phương thức testXXX() trong lớp đó và thêm chúng vào suite. Đoạn mã trên không làm khác gì so với việc JUnit tự động làm, tuy nhiên bạn có thể thêm các test cá nhân để chỉ chạy các test nhất định nào đó hay là điều khiển thứ tự thực thi

### CODE

```
import junit.framework.*;

public class TestGame extends TestCase {
private Game game;
private Ship fighter;

public TestGame(String name) {
super(name);
}
...
public static Test suite() {
TestSuite suite = new TestSuite();
suite.addTest(new TestGame("testCreateFighter"));
suite.addTest(new TestGame("testSameFighters"));
return suite;
}
}
```

Bạn có thể kết hợp nhiều suite vào các suite khác. Bạn có ra ở đây đã sử dụng mẫu Composite. Ví dụ:

```
public static Test suite() {  
    TestSuite suite = new TestSuite(TestGame.class);  
    suite.addTest(new TestSuite(TestPerson.class));  
    return suite;  
}
```

Bây giờ khi bạn chạy test case này, bạn sẽ chạy tất cả các test bên trong lớp TestGame và lớp TestPerson

### Chạy các Test lặp đi lặp lại

Trong một vài trường hợp, chúng ta muốn chạy một test nào đó lặp đi lặp lại nhiều lần để đo hiệu suất hay phân tích các vấn đề trực trặc. JUnit cung cấp cho chúng ta lớp junit.extension.RepeatedTest để làm được điều này. Lớp RepeatedTest giúp chúng ta thực hiện điều này một cách dễ dàng

#### CODE

```
public static Test suite() {  
    //Chạy toàn bộ test suite 10 lần  
    return new RepeatedTest(new TestSuite(TestGame.class), 10);  
}
```

Tham số đầu tiên của RepeatedTest là một Test cần chạy, tham số thứ 2 là số lần lặp lại. Vì TestSuite cài đặt interface Test nên chúng ta có thể lặp lại toàn bộ test như trên. Tiếp theo là ví dụ mô tả cách xây dựng một test suite mà trong đó các test khác nhau được lặp đi lặp lại khác nhau:

#### CODE

```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    //Lặp lại testCreateFighter 100 lần  
    suite.addTest(new RepeatedTest(new TestGame("testCreateFighter"), 100));  
    //Chạy testSameFighters 1 lần  
    suite.addTest(new TestGame("testSameFighters"));  
    //Lặp lại testGameInitialState 20 lần  
    suite.addTest(new RepeatedTest(new TestGame("testGameInitialState"), 20);  
    return suite;  
}
```

### Test các Exception

Phần tiếp đến chúng ta sẽ xem xét đến một phần test cũng quan trọng không kém trong lập trình là test các exception. Chúng ta sử dụng cặp từ khóa try/catch để bắt các exception như mong đợi, chúng ta sẽ gọi phương thức fail() khi exception chúng ta mong đợi không xảy ra. Trong ví dụ sau, constructor của lớp Person nên tung ra IllegalArgumentException khi cả 2 tham số của nó đều mang giá trị null. Test sẽ thất bại nếu nó không tung ra exception.

#### CODE

```
public void testPassNullsToConstructor() {  
    try {  
        Person p = new Person(null, null);  
    }
```



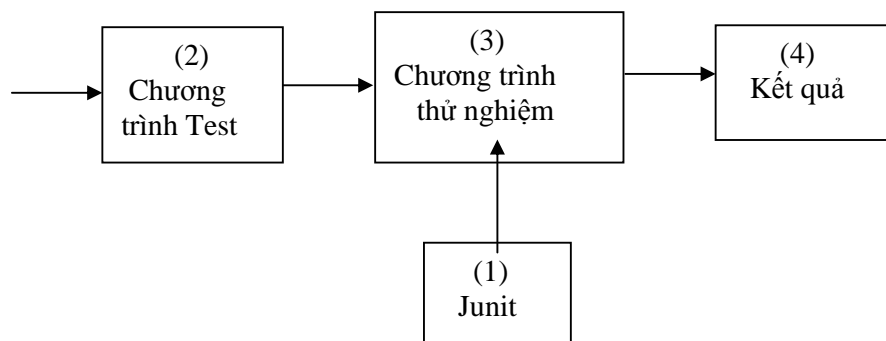
```
fail("Expected IllegalArgumentException because both args are null");
}
catch (IllegalArgumentException expected) {
//Bỏ qua phần này không xử lý vì có nghĩa là test được chấp nhận
}
}
```

Nói chung bạn chỉ nên sử dụng kỹ thuật này khi bạn mong đợi một exception xảy ra. Đối với các điều kiện lỗi khác bạn nên để exception chuyển sang cho JUnit. Khi đó JUnit sẽ bắt lấy và tường trình 1 lỗi test.

### Thực hiện kiểm thử

#### Sơ đồ kiểm thử

Sơ đồ sau biểu diễn cách test chương trình:



- (1) Junit: Tool dùng để test. Dự án này sử dụng bộ tool Junit 4.0
- (2) Chương trình Test
- (3) Chương trình thử nghiệm: Viết một TestCase để test chương trình trên.
- (4) Kết quả: Sau khi kiểm thử chương trình xong thì sẽ đưa ra kết quả là chương trình đúng hay sai.

#### II.3.2.Lập kế hoạch kiểm thử

Bước 1: Đầu tiên, ta phải tạo ra một dự án.

Bước 2: Add Junit 4.0 vào thư mục chứa đường dẫn của chương trình.

Bước 3: Mở chương trình Test và tạo TestCase cho chương trình.

Bước 4: Chạy chương trình

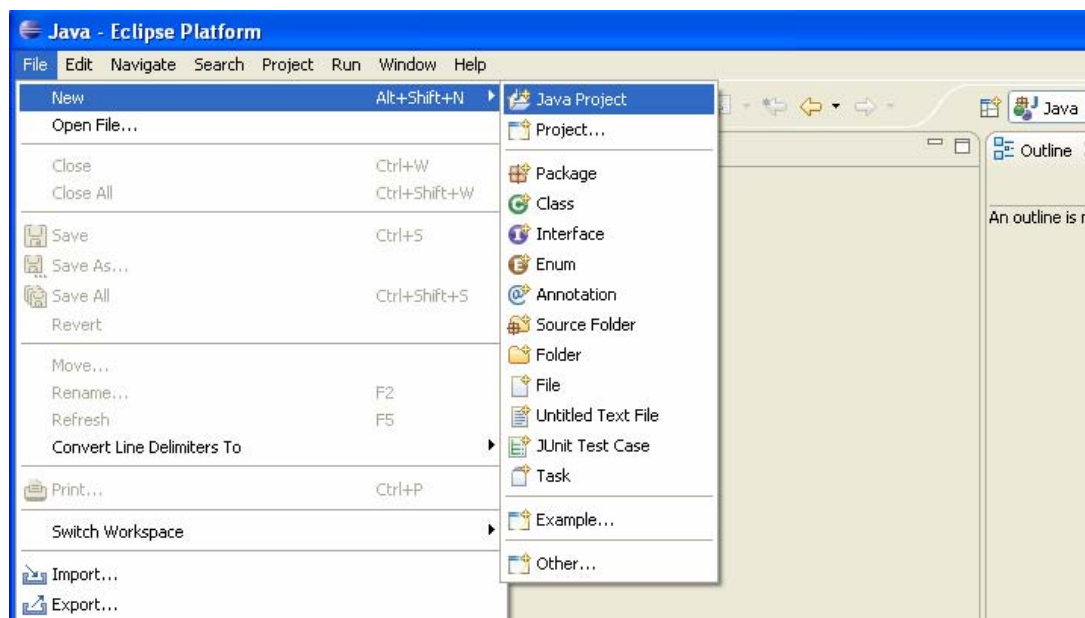
#### II.3.3. Quá trình kiểm thử

**Bước 1:** Khởi tạo dự án:

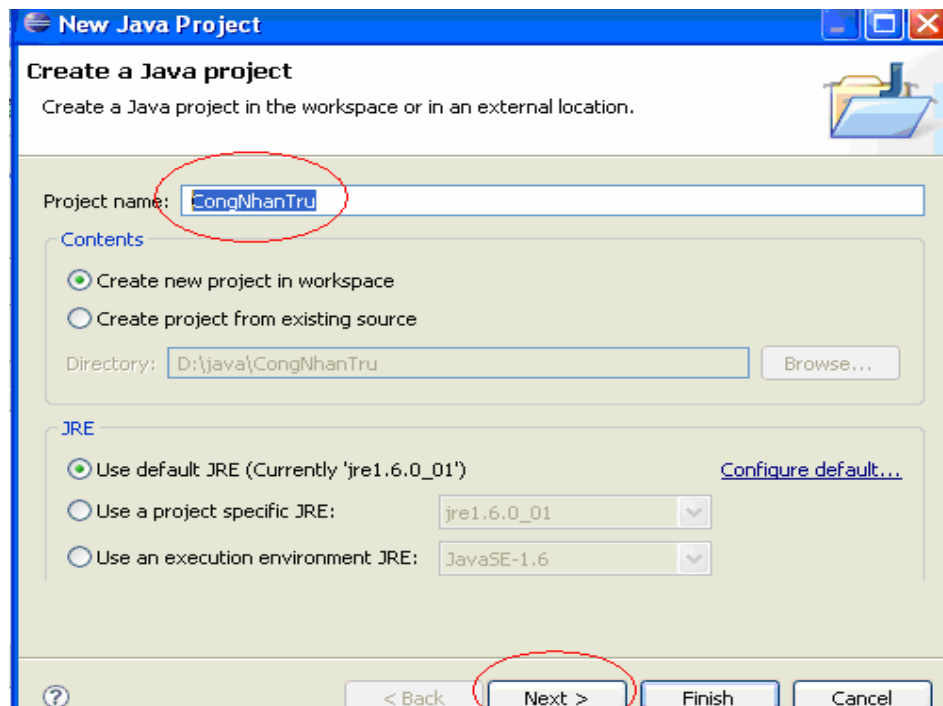
Chạy Eclipse, chọn đường dẫn để chứa dự án, nên chọn đường dẫn dễ nhớ để sau này Add Junit 4.0 vào đường dẫn của chương trình



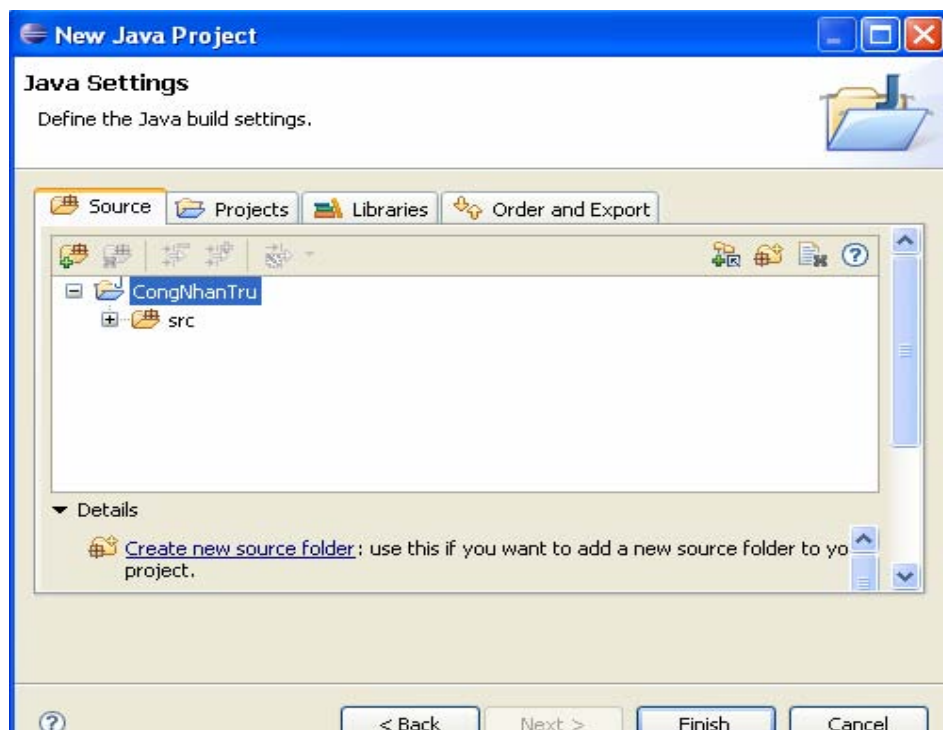
Tiếp theo, ta tạo một dự án để lưu chương trình test và chương trình kiểm thử. Từ menu chương trình, chọn File → New → Java Project.



Ở cửa sổ hiện ra, bạn đặt tên cho dự án của mình.



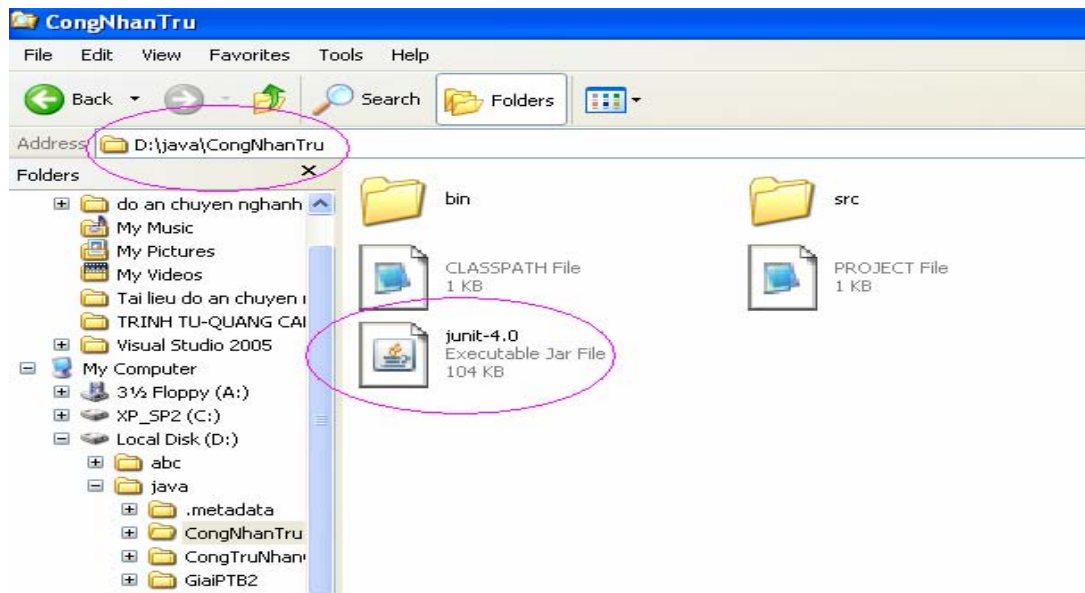
Tiếp tục, bạn nhấn nút Next. Chương trình sẽ hiện ra như bên dưới:



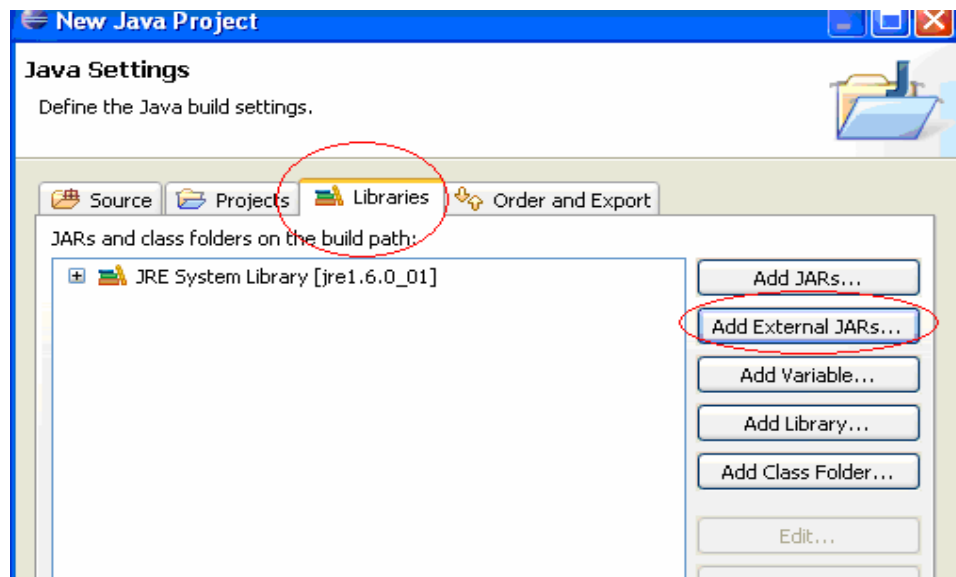
Đến đây, ta phải add tool Junit vào chương trình để tiến hành kiểm thử.

**Bước 2:** Add Junit 4.0 vào thư mục chứa đường dẫn của chương trình

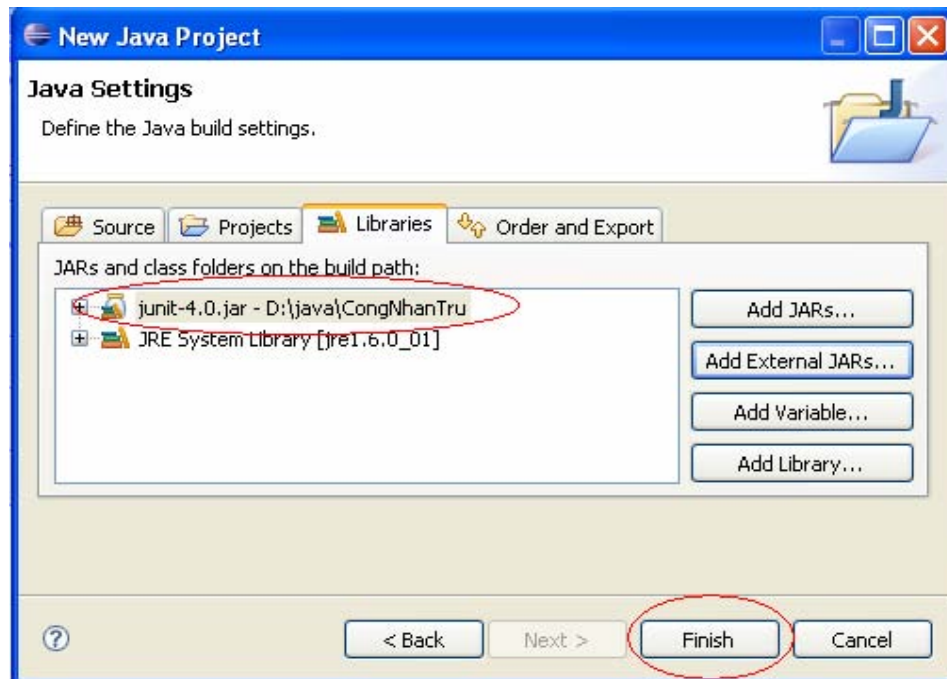
Trước tiên, chúng ta phải copy junit 4.0 vào thư mục của chương trình



Chọn thẻ **Libraries**, chọn **Add External Jars** để add junit 4.0 và chương trình

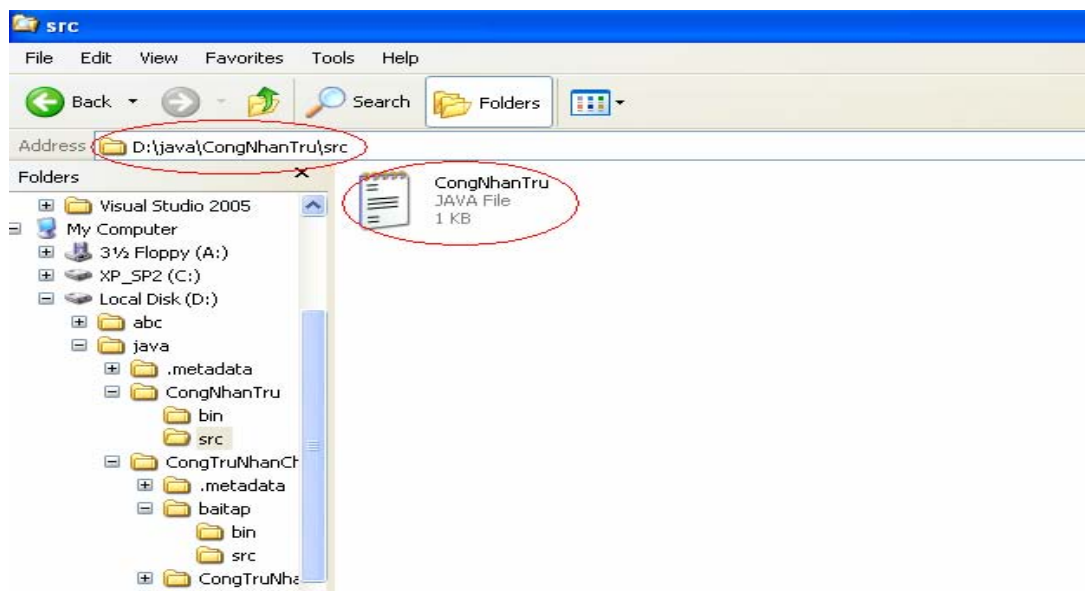


Sau khi add junit 4.0 thì chương trình có hình như bên dưới. Nhấn nút Finish để kết thúc giai đoạn thêm junit vào chương trình.

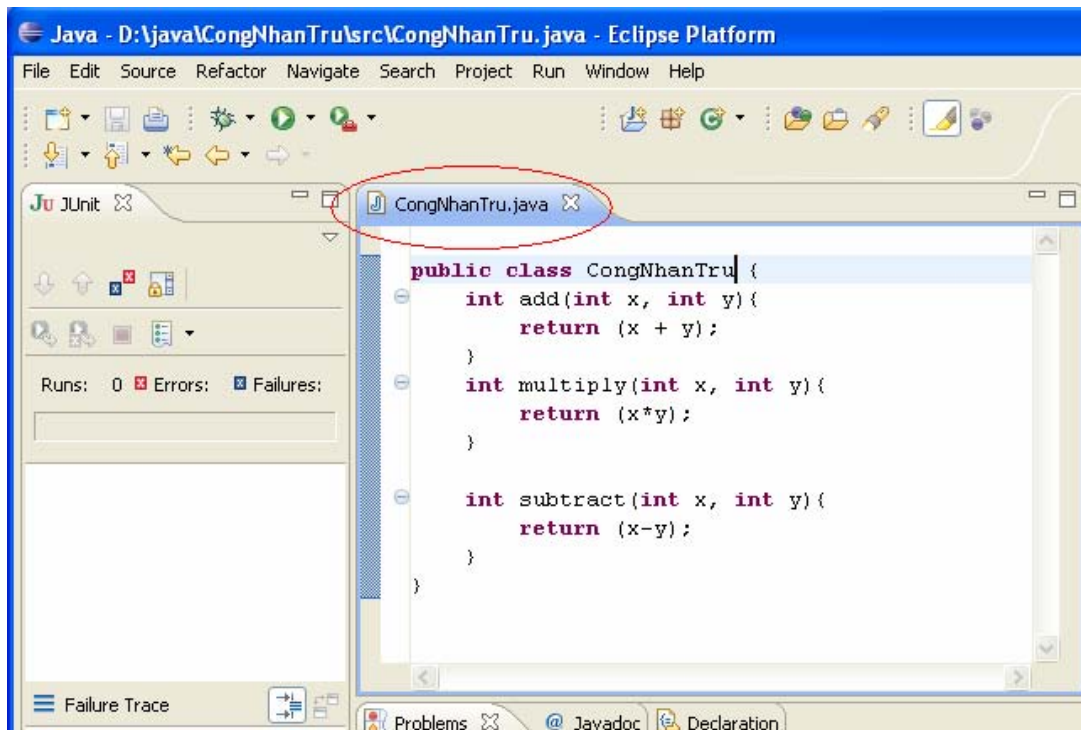


### **Bước 3: Mở chương trình Test và tạo TestCase cho chương trình.**

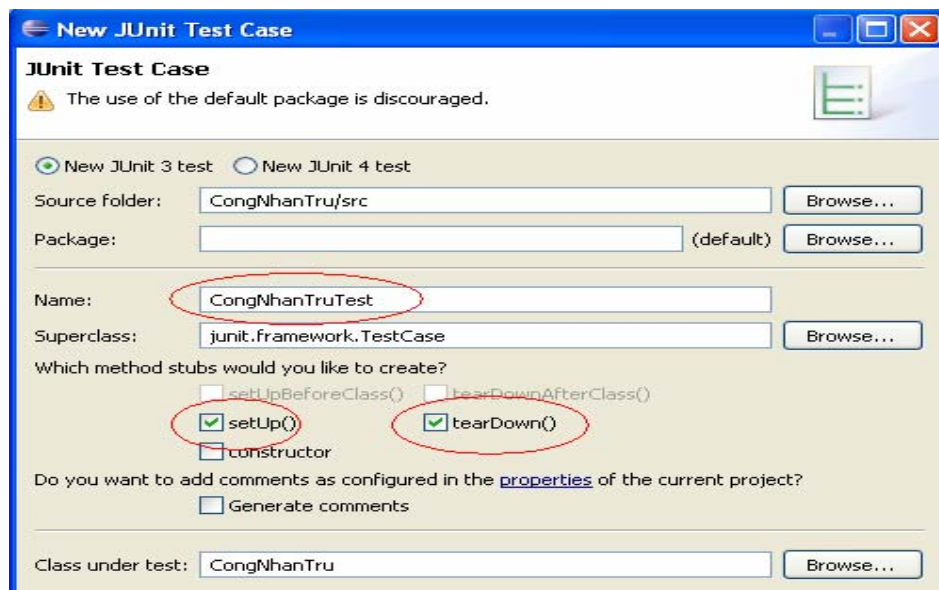
Trước tiên, bạn phải copy chương trình test vào đường dẫn của dự án.



Từ menu chương trình, chọn File → Open File để mở chương trình Test.

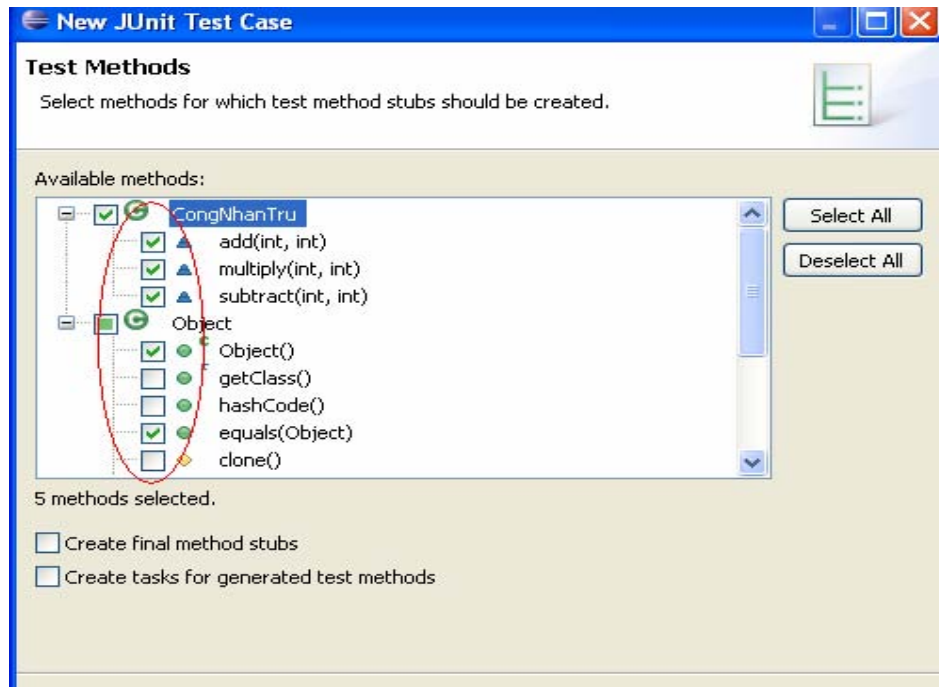


Chúng ta tiếp tục tạo TestCase cho chương trình test. Từ menu chương trình, chọn File → New → JUnit Test Case. Ở cửa sổ hiện ra, chương trình tự tạo ra tên test case, với tên có phần đầu trùng với tên của lớp cần test và phần cuối thêm đuôi Test. Trong cửa sổ này, bạn check vào hai phương thức setUp() và tearDown().

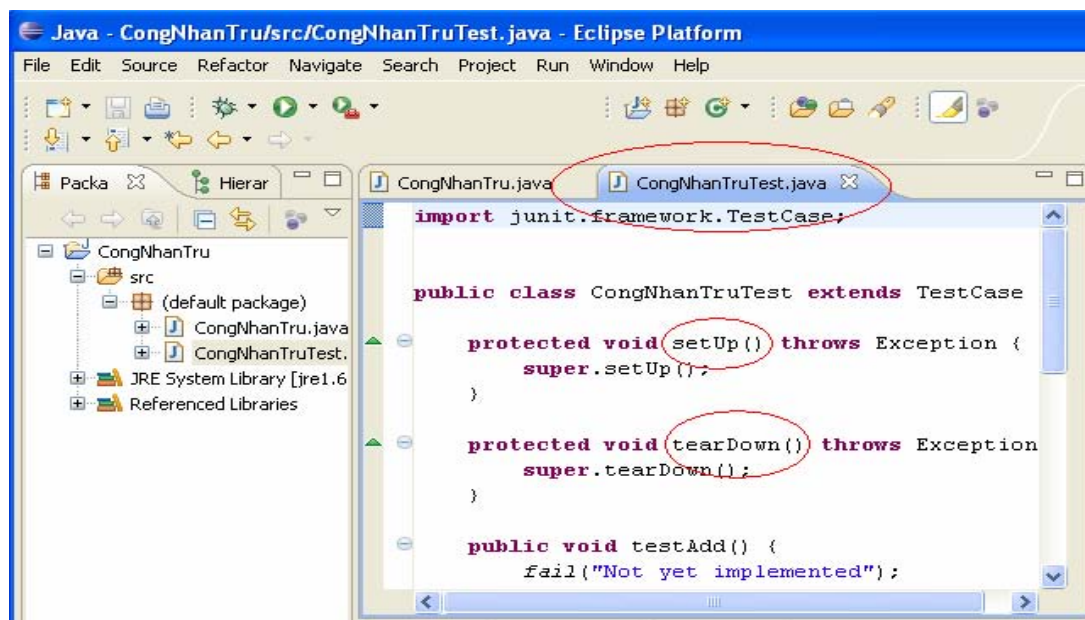


Sau khi thực hiện xong thì ta nhấn nút Next để chọn phương thức cho chương trình. Đối với chương trình này, chúng ta check vào các phương thức được minh họa bằng hình dưới:




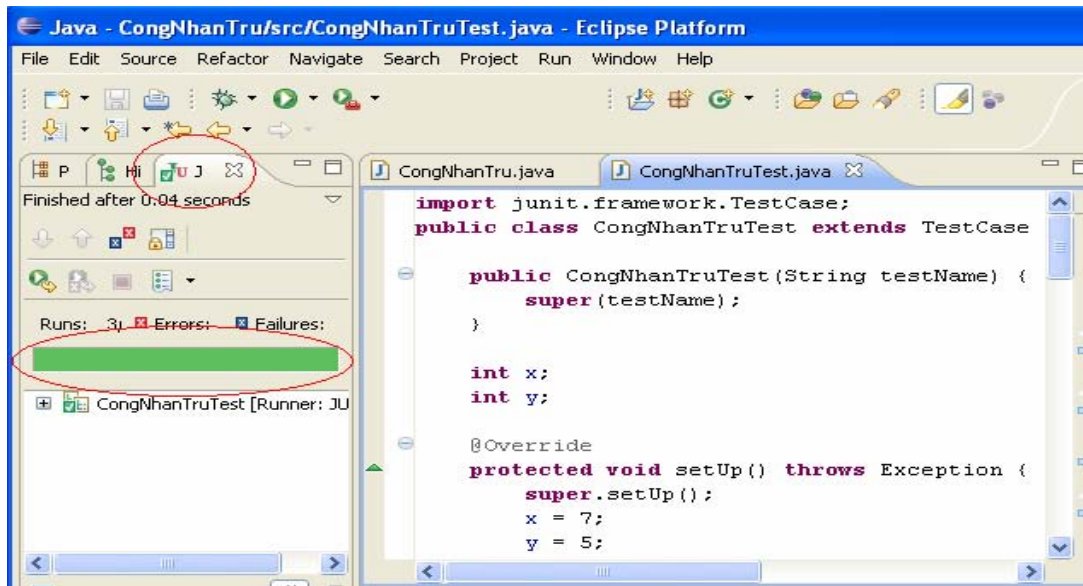


Nhấn nút Finish, chương trình sẽ có hình như bên dưới. Tiếp theo chúng ta sẽ viết TestCase dựa trên những phương thức hỗ trợ của Junit để test chương trình.

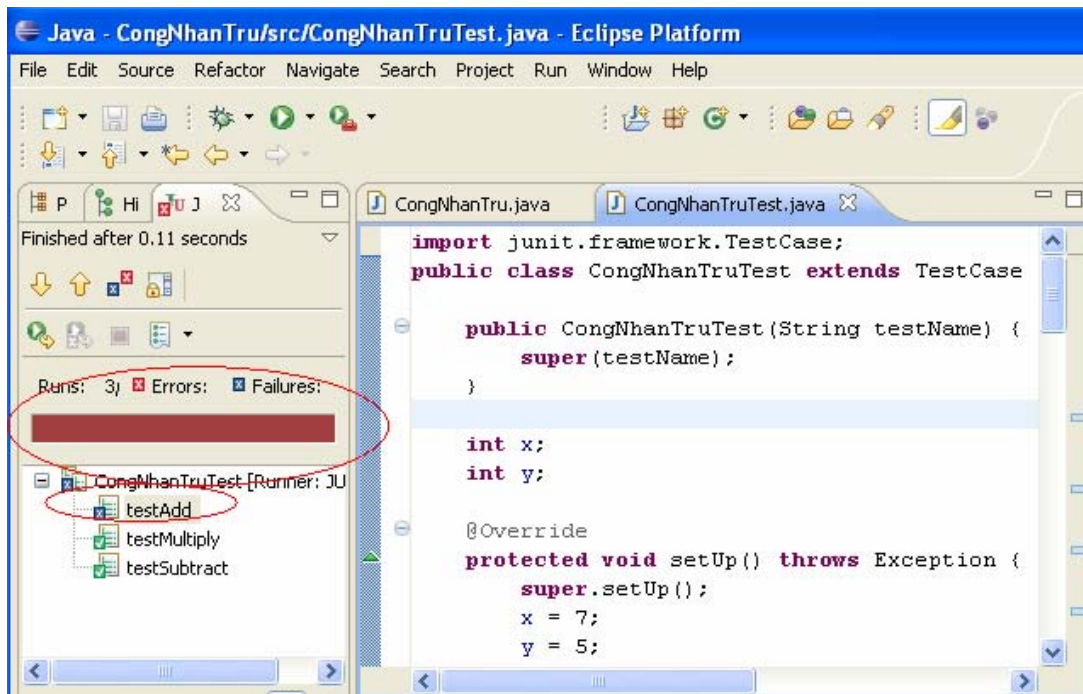


#### **Bước 4:** Chạy chương trình

Sau khi đã tạo và viết TestCase cho chương trình Test, chúng ta sẽ chạy chương trình kiểm thử. Từ menu chương trình, nhấn nút  hoặc nhấn F5. Nếu chương trình test không có lỗi thì chương trình sẽ thể hiện một màu xanh, thể hiện sự thử thành công.



Nếu chương trình Test bị lỗi thì Junit sẽ hiển thị một màu đỏ, báo hiệu chương trình bị lỗi. Đồng thời, chương trình còn chỉ rõ phương thức hay lớp nào bị lỗi.





## CHƯƠNG 8: ƯỚC LƯỢNG GIÁ THÀNH PHẦN MỀM

### 8.1. Giới thiệu

Trong chương trước, tôi đã giới thiệu về quá trình lập kế hoạch cho dự án, những công việc của dự án được chia thành các hoạt động riêng biệt. Sự thảo luận sớm về việc lập kế hoạch cho dự án tập trung vào việc miêu tả các hoạt động, sự phụ thuộc và sự phân phối của con người để thực hiện các nhiệm vụ. Trong chương này, tôi muốn nói đến vấn đề kết hợp ước lượng của sự cố gắng và thời gian với các hoạt động của dự án. Sự ước lượng bao gồm trả lời các câu hỏi sau:

1. Bao nhiêu cố gắng được yêu cầu để hoàn thành mỗi hoạt động.
2. Bao nhiêu thời gian cần thiết để hoàn thành mỗi hoạt động.
3. Tổng giá trị của mỗi hoạt động.

Ước lượng chi phí dự án và lập kế hoạch cho dự án thường được tiến hành cùng nhau. Chi phí của sự phát triển là chi phí chính trong các thứ bao gồm, do đó sự tính toán được sử dụng là trong cả ước đoán chi phí và ước đoán lịch trình. Tuy nhiên, bạn có thể phải làm một số sự ước lượng chi phí trước khi lịch trình được đi vào cụ thể. Sự ước lượng khởi đầu có thể sử dụng là ngân sách cho dự án hoặc tập giá trị phần mềm cho khách hàng.

Có ba tham số trong sự tính toán tổng chi phí của dự án phát triển phần mềm:

- Chi phí phần cứng và phần mềm kể cả bảo dưỡng.
- Chi phí đi lại và đào tạo.
- Chi phí cho kỹ sư (Chi phí chi trả chi kỹ sư phần mềm).

Hầu hết các dự án, chi phí chủ yếu là chi phí cho kỹ sư. Máy tính mà nó đủ mạnh cho phát triển phần mềm là phải tương đối rẻ. Mặc dù chi phí cho sự đi lại có thể cần thiết khi dự án được phát triển ở các địa điểm khác nhau, chi phí đi lại thường chỉ là một phần nhỏ so với chi phí cho kỹ sư. Hơn nữa, sử dụng các hệ thống giao tiếp điện tử như thư điện tử, các trang web và hội nghị trực tuyến có thể làm giảm đáng kể yêu cầu cho sự đi lại. Hội thảo điện tử cũng có nghĩa là thời gian đi lại giảm và có thể có năng suất cao hơn cho việc phát triển phần mềm. Trong một dự án, nơi tôi đã làm việc, tạo ra tất cả các hội thảo trực tuyến thay vì các hội thảo gặp mặt trực tiếp, đã giảm được chi phí đi lại và thời gian đến 50%.

Chi phí cho kỹ sư không hẳn là lương của các kỹ sư phần mềm mà họ có trong dự án. Tổ chức tính toán chi phí cho kỹ sư trong nhóm dựa trên tất cả chi phí mà họ có được để thực hiện dự án và chia nó cho số lượng năng suất nhân viên. Do đó, các chi phí sau đây là tất cả các thành phần của chi phí cho kỹ sư.

1. Chi phí cho điều kiện nhiệt độ, ánh sáng của công sở.
2. Chi phí hỗ trợ nhân viên như nhân viên kế toán, nhà quản trị, người quản lý hệ thống, người dọn dẹp, và các kỹ thuật viên.
3. Chi phí kết nối mạng và giao tiếp
4. Chi phí cho những trung tâm tạo điều kiện tốt cho làm việc như thư viện hay nơi giải trí.

5. Chi phí cho sự bảo vệ chung và lợi ích cho người làm việc như tiền trợ cấp và bảo hiểm sức khỏe.

**Chi phí chính thường ít nhất bằng hai lần lương của kỹ sư phần mềm, phụ thuộc và kích thước tổ chức và sự kết hợp của nó. Vì thế, nếu một công ty trả chi kỹ sư phần mềm 90.000 bảng/Năm thì tổng chi phí ít nhất là 180.000 Bảng/Năm hay 15.000 bảng/Tháng.**

Khi một dự án đang được thực hiện, người quản lý dự án nên thường xuyên và đều đặn cập nhật ước lượng chi phí và lịch trình. Điều này giúp cho việc lập kế hoạch cho tiến trình và việc sử dụng hiệu quả tài nguyên. Nếu thực sự có sự tiêu dùng vượt đáng kể so với dự tính, thì người quản lý dự án phải có một số hành động. Nó có thể bao gồm việc tăng ngân sách cho dự án hoặc sửa đổi lại công việc cho nó trở nên đúng đắn.

Dự toán phần mềm nên được thực hiện khách quan với mục đích dự đoán chính xác chi phí phát triển phần mềm. Nếu chi phí dự án được tính toán như giá khách hàng phải trả cho dự án, thì việc quyết định định giá là việc đặt giá cho khách hàng. Giá cả đơn giản là chi phí cộng với lợi nhuận. Tuy nhiên, mối quan hệ giữa chi phí dự án và giá chi khách hàng thường không đơn giản.

Việc định giá cho phần mềm phải bao gồm cả tổ chức, kinh tế, chính trị, thương mại, mà nó được thể hiện trong hình 26.1. Bởi vậy, nó có thể không chỉ là sự quan hệ đơn giản giữa giá cả khách hàng phải trả cho dự án và chi phí phát triển dự án. Bởi vì bao gồm nhiều lý do trong tổ chức, việc định giá dự án nên có sự góp mặt của những nhà quản lý kinh nghiệm (để có những quyết định chiến lược).

Ví dụ, một công ty phần mềm nhỏ phục vụ dầu thuê 10 kỹ sư hồi đầu năm, nhưng chỉ có những hợp đồng trong lĩnh vực mà nó yêu cầu 5 thành viên trong việc mở rộng nhân viên. Tuy nhiên, đó là sự trả giá cho những hợp đồng rất lớn với công ty dầu lớn mà nó yêu cầu 30 người làm việc cố gắng trong 2 năm. Dự án sẽ không thể bắt đầu sau 12 tháng, và như thế điều đó sẽ thay đổi tài chính của công ty nhỏ. Công ty cung cấp dầu có cơ hội trả giá cho dự án mà nó yêu cầu 6 người và phải hoàn thành trong 10 tháng. Chi phí (bao gồm tất cả trong dự án) được ước tính là 1.2 triệu Đô. Tuy nhiên, để tăng khả năng cạnh tranh, công ty cung cấp dầu định giá cho khách hàng là 0.8 triệu Đô. Điều đó có nghĩa là, mặc dù mất tiền trong hợp đồng này, nhưng nó đã được những chuyên viên hữu ích cho những dự án trong tương lai.

Hình 8.1 Nhân tố ảnh hưởng giá phần mềm

Nhân tố	Diễn tả
Cơ hội trong thị trường	Một tổ chức phát triển có thể đặt một mức giá thấp bởi vì nó mong muốn chuyển tới một giai đoạn mới trong thị trường phần mềm. Chấp nhận một mức lợi thấp trong một dự án có thể cho tổ chức cơ hội có lợi nhuận lớn hơn trong tương lai, và vốn kinh nghiệm thu được có thể sẽ rất hữu ích trong tương lai.
Sự không chắc chắn của việc ước lượng chi phí	Nếu một tổ chức không dám chắc và sự ước tính chi phí, điều đó có thể làm gia tăng giá thành bởi một số sự cố bất ngờ.
Những yêu cầu dễ thay đổi.	Nếu các yêu cầu có khả năng thay đổi, một tổ chức có thể giảm giá thành để có được hợp đồng. Sau đó, lại tăng giá thành để thay đổi

	các yêu cầu.
Tình trạng tài chính	Nhà phát triển trong sự khó khăn tài chính có thể giảm giá thành để đạt được hợp đồng. Điều đó tốt hơn để tạo ra các lợi nhuận nhỏ hơn là bình thường hoặc sẽ không thể tiếp tục kinh doanh.

## 8.2. Năng suất phần mềm

Bạn có thể tính toán năng suất trong hệ thống bằng cách đếm số lượng các đơn vị được sản xuất và chia chúng cho lượng thời gian yêu cầu làm ra chúng. Tuy nhiên, với bất kỳ vấn đề phần mềm nào, sẽ có nhiều giải pháp khác nhau, mỗi trong số chúng lại có những tính chất khác nhau. Một giải pháp có thể thực hiện một cách hiệu quả trong khi những cái khác lại có thể dễ đọc và dễ bảo trì. Khi những giải pháp với những thuộc tính khác nhau được đưa ra, việc so sánh chúng thường không có thật nhiều ý nghĩa.

Tuy nhiên, là một nhà quản lý dự án, bạn có thể phải đối mặt với vấn đề về ước lượng năng suất của kỹ sư phần mềm. Bạn có thể cần sự ước lượng này để giúp đưa ra chi phí cho dự án và lập lịch trình, để quyết định sự đầu tư hoặc để đánh giá phát triển quá trình và kỹ thuật thêm hiệu quả.

Ước lượng năng suất thường có cơ sở dựa trên việc tính toán các thuộc tính của phần mềm và chia chúng cho tổng sự cố gắng yêu cầu cho việc phát triển. Có hai loại thang đo được sử dụng:

1. *Thước đo liên quan đến kích thước* Đó là sự liên quan đến kích thước một số đầu ra từ một số hoạt động. Cái chung nhất trong thước đo này là các đường phân chia mã nguồn. Các thước đo khác có thể sử dụng số lượng của phân chia lệnh mã mục tiêu hoặc số lượng của các trang của tài liệu hệ thống.
2. *Thước đo liên quan đến hàm* Đó là sự liên quan đến tổng thể các chức năng trong sự phân chia phần mềm. Năng suất giới hạn bởi số lượng chức năng hữu dụng được sử dụng trong thời gian hạn định. Điểm hàm và điểm mục tiêu là điều quan trọng nhất trong loại thước đo này.

Các đường mã nguồn / lập trình viên-tháng ( Lines of source code per programmer-month – LOC/pm) được sử dụng rất nhiều trong việc đo năng suất phần mềm. Bạn có thể tính toán LOC/pm bằng cách đếm tổng số đường mã nguồn mà nó được phân chia, sau đó chia nó cho tổng thời gian làm việc của lập trình viên trong một tháng yêu cầu để hoàn thành dự án. Thời gian này vì thế bao gồm tất cả các hoạt động yêu cầu (Nhu cầu, thiết kế, viết mã, kiểm tra và dẫn chứng tư liệu) cần thiết trong việc phát triển phần mềm.

Điều đó tiến gần đến sự phát triển đầu tiên khi hầu hết các chương trình được lập bằng FORTRAN, hợp ngữ hoặc COBOL. Sau đó, các chương trình như một loại thẻ, với mỗi câu lệnh trên một thẻ. Số lượng của mã dễ dàng đếm được: Nó tương ứng với số lượng của các thẻ trong chương trình. Tuy nhiên, các chương trình trong các ngôn ngữ như JAVA hoặc C++ bao gồm các khai báo, các câu lệnh và các chú thích. Nó có thể cũng bao gồm các macro lệnh mà nó mở rộng thêm vài dòng trong đoạn mã. Nó cũng có thể có nhiều hơn một câu lệnh trên một dòng. Vì thế, quan hệ đơn giản giữa các lệnh chương trình và các dòng ở trong danh sách.

So sánh năng suất thông qua ngôn ngữ lập trình có thể cũng mắc các sai lầm do cảm giác về năng suất chương trình. Các ngôn ngữ lập trình càng cao thì càng khó xác định năng suất. Điều không bình thường này nảy sinh bởi vì tất cả các hoạt động phát triển phần mềm đều được tính toán đến trong khi tính toán thời gian phát triển, nhưng thang đo LOC chỉ áp dụng cho quá trình lập trình. Vì thế, nếu một ngôn ngữ yêu cầu nhiều dòng hơn các cái khác để bổ sung cùng một chức năng, việc ước lượng năng suất sẽ trở nên bất thường.

Ví dụ, xem xét một hệ thống ghi thời gian thực nó có thể có mã trong 5000 dòng trong hợp ngữ hoặc 1500 dòng trong C. Thời gian phát triển cho các mặt khác nhau được diễn tả trong hình 8.2. Người lập trình hợp ngữ có năng suất là 714 dòng/tháng và người lập trình ngôn ngữ bậc cao thì ít hơn một nửa là 300 dòng/tháng. Vào lúc này chi phí phát triển cho phát triển hệ thống trong C là thấp hơn và nó được phát biểu sớm.

Hình 8.2 Thời gian phát triển hệ thống

	Phân tích	Thiết kế	Viết mã	Kiểm tra	Báo cáo
Hợp ngữ	3 tuần	5 tuần	8 tuần	10 tuần	2 tuần
Ngôn ngữ cấp cao	3 tuần	4 tuần	4 tuần	6 tuần	2 tuần
	Kích thước		Thời gian làm		Năng suất
Hợp ngữ	5000 dòng		28 tuần		714 dòng/tháng
Ngôn ngữ cấp cao	1500 dòng		20 tuần		300 dòng/tháng

Cách khác sử dụng kích cỡ mã như việc ước lượng thuộc tính sản phẩm là sử dụng các tính toán chức năng của mã. Điều này tránh các bất thường nêu trên, như các chức năng là độc lập với ngôn ngữ bổ sung. MacDonell (MacDonell, 1994) mô tả ngắn gọn và so sánh vài tính toán về các hàm cơ bản. Hiểu biết tốt nhất về tính toán này là việc đếm điểm-hàm. Nó được trình bày bởi Albrecht (Albrecht, 1979) và được sửa lại bởi Albrecht và Gaffney (Albrecht và Gaffney, 1983). Garmus và Herron (Garmus và Herron, 2000) mô tả thực tiễn sử dụng điểm-hàm trong các dự án phần mềm.

Năng suất thông qua điểm-hàm là sự bổ sung của người làm việc trên tháng. Điểm hàm không chỉ là đặc điểm mà là sự tính toán một vài sự đo đạc và ước lượng. Bạn tính tổng số điểm-hàm trong chương trình bằng cách đo hoặc ước lượng các đặc điểm sau của chương trình:

- Giao tiếp vào và ra
- Sử dụng tương tác
- Giao diện giao tiếp
- Văn bản sử dụng trong hệ thống

Hiển nhiên, một số đầu vào và đầu ra, các tương tác, là những thứ phức tạp và tốn nhiều thời gian hơn để bổ sung. Thước đo điểm-hàm đưa vào báo cáo bằng cách nhân điểm-hàm khởi tạo với hệ số-phức tạp-tác dụng. Bạn nên ước định mỗi trong các đặc điểm cho sự phức tạp và sau đó gán hệ số tác dụng từ 3 (cho giao tiếp đầu vào đơn giản) tới 15 cho các văn bản nội bộ phức tạp. Có thể lựa chọn giá trị tác dụng đề xuất bởi Albrecht hoặc các giá trị cơ sở trong kinh nghiệm để sử dụng.

Sau đó bạn có thể tính toán cái gọi là đếm điểm hàm không thích ứng (unadjusted function-point count-UFC) bằng cách nhân mỗi giá trị khởi tạo với ước lượng tác dụng và cộng tổng tất cả các giá trị.

$$UFC = \Sigma(\text{số lượng của các thành phần có kiểu nhất định}) * (\text{hệ số tác dụng})$$

Bạn có thể thay đổi UFC bằng cách thêm vào hệ số phức tạp mà nó gần với độ phức tạp của toàn bộ hệ thống. Điều này đưa vào báo cáo mức độ phân bố tiến trình, tổng số dùng lại, hiệu suất,... Đếm điểm hàm không thích ứng (UFC) được nhân với hệ số phức tạp dự án để đưa ra điểm hàm cho toàn bộ hệ thống.

Symons (Symons, 1988) chú ý rằng ước lượng độ phức tạp bằng cách đếm điểm-hàm trong chương trình phụ thuộc vào người đưa ra ước lượng. Con người khác nhau sẽ có những ý niệm khác nhau về độ phức tạp. Vì thế sự đa dạng trong việc đếm điểm-hàm phụ thuộc vào quyết định của người ước lượng và kiểu của hệ thống được phát triển. Hơn thế nữa, điểm hàm thường có xu hướng về hệ thống sử lý dữ liệu mà nó bị chi phối bởi các toán tử vào và ra. Và thật khó để ước lượng điểm hàm cho các hệ thống hướng sự kiện. Vì lý do đó, một số người nghĩ điểm hàm là không thực sự thích hợp để tính toán năng suất phần mềm (Furey và Kitchenham, 1997; Armour, 2002). Tuy nhiên, người sử dụng điểm hàm tranh luận gây gắt rằng đó là một cách hiệu quả trong thực tiễn (Banker,... 1993, Garmus và Herron, 2000)

Điểm mục tiêu (Banker,...,1994) là một khả năng khác của điểm hàm. Chúng có thể được sử dụng với các ngôn ngữ như các ngôn ngữ lập trình cơ sở dữ liệu, ngôn ngữ kịch bản. Điểm mục tiêu không phải là lớp mục tiêu mà nó có thể sản xuất khi tiến gần định hướng mục tiêu được đưa lại trong sự phát triển phần mềm. Số lượng của điểm hàm trong chương trình là việc ước lượng tác dụng của:

1. *Số lượng các màn ảnh riêng biệt được hiển thị* một màn hình đơn được coi như một điểm mục tiêu, các màn ảnh phức tạp vừa phải được đếm là 2, và các màn ảnh rất phức tạp được đếm là 3.
2. *Số lượng các biên bản được đưa ra* cho một biên bản đơn lẻ đếm thêm 2 điểm mục tiêu, cho các biên bản phức tạp vừa phải là 5 và các biên bản rất khó để đưa ra là 8.
3. *Số lượng các modul trong các ngôn ngữ lập trình bắt buộc như Java hoặc C++ mà nó phải được phát triển để bổ xung cho mã lập trình cơ sở dữ liệu* mỗi trong các modul được đếm 10 điểm mục tiêu.

Điểm mục tiêu được sử dụng trong mô hình ước lượng COCOMO II (được gọi là điểm ứng dụng) được nói đến trong các chương sau. Lợi thế của điểm mục tiêu hơn điểm hàm là nó dễ ràng hơn để ước lượng cho các chi tiết kỹ thuật phần mềm cấp cao. Điểm mục tiêu thường chỉ liên quan đến các màn hình, biên bản và các modul trong các ngôn ngữ lập trình. Nó thường không liên quan đến các chi tiết bổ xung, và các ước lượng hệ số phức tạp là đơn giản.

Nếu điểm hàm hoặc điểm mục tiêu được sử dụng, chúng có thể được ước lượng trong giai đoạn sớm của quá trình phát triển trước khi quyết định ảnh hưởng đến kích thước chương trình được tạo ra. Ước lượng các thông số có thể làm cùng giao tiếp tương tác của hệ thống khi được thiết kế. Trong giai đoạn này, là rất khó để đưa ra kích thước của chương trình và số dòng của mã nguồn.

Điểm hàm và điểm mục tiêu có thể được sử dụng kết hợp với mô hình ước lượng số dòng của mã. Kích thước mã cuối cùng được tính toán từ số lượng điểm hàm. Sử dụng phân tích dữ liệu lịch sử, số lượng dòng trung bình của mã, AVC, trong các ngôn ngữ riêng

biệt để bổ xung các điểm hàm có thể ước lượng được. Giá trị của AVC thay đổi từ 200 đến 300 LOC/FP trong hợp ngữ, từ 2 đến 40 LOC/FP trong ngôn ngữ lập trình cơ sở dữ liệu như SQL. Ước lượng kích thước mã cho ứng dụng mới được tính toán như sau:

Kích thước mã = AVC\*số lượng điểm hàm.

Năng suất lập trình trong làm việc cá nhân trong tổ chức là phụ thuộc vào nhiều yếu tố. Một số trong các yếu tố quan trọng được đưa ra trong hình 26.3. Tuy nhiên, cá nhân khác nhau trong khả năng thường có nhiều ý nghĩa hơn bất kỳ nhân tố nào. Trong sự ước định sớm của năng suất, Sackman (Sackman, 1968) tìm thấy một số lập trình viên mà họ có năng suất gấp hơn 10 lần so với những người khác. Kinh nghiệm của tôi cho thấy điều này vẫn đúng. Những nhóm lớn thường có khả năng hoà hợp khả năng và kinh nghiệm và sẽ có được năng suất trung bình. Trong những nhóm nhỏ, tuy nhiên, năng suất chung thường phụ thuộc rất nhiều vào các khả năng và năng khiếu cá nhân.

Hình 8.3 Các nhân tố ảnh hưởng đến năng suất thiết kế phần mềm.

Nhân tố	Mô tả
Kinh nghiệm miền ứng dụng	Hiểu biết về miền ứng dụng là cần thiết để phát triển phần mềm hiệu quả. Những kỹ sư đã hiểu về miền thường làm việc có hiệu quả cao.
Chất lượng quá trình	Sử dụng quá trình phát triển có thể có hiệu quả đặc biệt trong năng suất. Điều này được nói đến trong Chương 28
Kích thước dự án	Một dự án lớn sẽ cần nhiều thời gian hơn yêu cầu cho việc giao tiếp trong nhóm. Ít thời gian có thể sử dụng cho việc phát triển dẫn đến suy giảm năng suất cá nhân.
Hỗ trợ kỹ thuật	Hỗ trợ kỹ thuật tốt như công cụ CASE và hệ thống tổ chức quản lý cấu hình có thể tăng năng suất.
Môi trường làm việc	Như đã thảo luận trong Chương 25, một môi trường làm việc yên tĩnh với vùng làm việc riêng góp phần nâng cao năng suất.

Năng suất phát triển phần mềm thay đổi nhanh chóng giữa miền ứng dụng và các tổ chức. Với hệ thống lớn, phức tạp, năng suất được ước lượng dưới 30 LOC/pm. Với những hệ thống ứng dụng dễ hiểu, viết bằng ngôn ngữ như Java, nó có thể cao hơn 900 LOC/pm. Khi tính toán giới hạn của điểm hàm Boehm (Boehm, 1995) đề xuất năng suất thay đổi từ 4 điểm hàm trên tháng đến 50 trên tháng, phụ thuộc vào kiểu ứng dụng, công cụ hỗ trợ và khả năng phát triển.

Vấn đề với phép đo mà nó phụ thuộc vào số lượng sản phẩm trong thời gian hạn định là nó đưa vào báo cáo về đặc điểm chất lượng có thể duy trì và đáng tin cậy. Beck (Beck, 2000) trong thảo luận của ông về lập trình, đã tạo điểm xuất sắc về sự ước lượng. Nếu cách tiếp cận của bạn là cơ sở cho việc đơn giản mã và phát triển, do đó việc đến dòng trong mã không còn nhiều ý nghĩa.

Phép đo cũng không đưa vào báo cáo khả năng sử dụng lại các sản phẩm phần mềm, sử dụng những mã có sẵn và các công cụ khác để tạo ra phần mềm.

Là một nhà quản lý, bạn không nên sử dụng việc tính toán năng suất để có những quyết định vội vàng về khả năng của kỹ sư trong nhóm của bạn. Nếu bạn làm, kỹ sư có thể thoả hiệp về chất lượng trong kế hoạch trở nên hiệu quả hơn. Nó có thể trở thành trường

hợp mà người lập trình kém hiệu quả đưa ra mã đáng tin cậy hơn – mã mà dễ dàng để hiểu và có thể duy trì không tốn kém. Vì thế, bạn nên luôn nghĩ về đo năng suất như cung cấp các thông tin không hoàn chỉnh về năng suất của lập trình viên. Bạn cũng nên quan tâm đến các thông tin khác về chất lượng của chương trình họ đã làm ra.

### 8.3. Kỹ thuật ước lượng

Không có cách đơn giản nào tạo ra ước lượng một cách chính xác cho hệ thống phát triển phần mềm. Bạn có thể phải tạo ra ước lượng khởi tạo trên nền tảng của định nghĩa yêu cầu người dùng cấp cao. Phần mềm có thể phải thực thi trong các máy tính lạ hoặc sử dụng kỹ thuật phát triển mới. Những người trong dự án và kỹ năng của họ có thể sẽ không được biết đến. Tất cả những ý trên là không thể ước lượng chi phí phát triển hệ thống một cách chính xác trong giai đoạn đầu của dự án.

Hơn thế nữa, đó là cái khó cơ bản trong việc đánh giá đúng đắn của các cách tiếp cận khác nhau đến kỹ thuật ước lượng chi phí. Sự ước lượng được dùng để xác định ngân sách dự án, và sản phẩm sẽ được điều chỉnh hợp lý. Tôi không biết gì về các cuộc thí nghiệm điều khiển với việc định giá dự án nơi mà ước lượng chi phí không được sử dụng trong thí nghiệm. Cuộc thí nghiệm điều khiển sẽ không bộc lộ được ước lượng chi phí cho người quản lý dự án. Chi phí trên thực tế sẽ được so sánh với chi phí ước lượng. Tuy nhiên, như một cuộc thí nghiệm là có thể không khả thi bởi vì bao gồm chi phí cao và số lượng có thể thay đổi là không thể điều khiển.

Tuy nhiên, tổ chức cần phải tạo ra sự cố gắng trong phần mềm và ước lượng chi phí. Để làm việc đó, các kỹ thuật được đưa ra trong hình 8.4 có thể được sử dụng (Boehm, 1981). Tất cả các kỹ thuật đều dựa trên nền tảng kinh nghiệm quyết định bởi những người quản lý dự án mà họ sử dụng hiểu biết của họ để xem xét dự án và ước lượng tài nguyên yêu cầu cho dự án. Tuy nhiên, có thể có những khác biệt quan trọng giữa dự án quá khứ và tương lai. Rất nhiều phương thức và kỹ thuật phát triển mới được giới thiệu sau 10 năm. Một số thí dụ ảnh hưởng đến cơ sở ước lượng trong nhiệm vụ bao gồm:

1. Phân phối các hệ thống mục tiêu hơn là hệ thống chung.
2. Sử dụng dịch vụ web.
3. Sử dụng ERP hoặc các hệ thống trung tâm cơ sở dữ liệu
4. Sử dụng các phần mềm không bảo vệ hơn là phát triển hệ thống nguồn.
5. Phát triển cùng với việc sử dụng lại hơn là phát triển mới tất cả các thành phần của hệ thống.
6. Phát triển sử dụng các ngôn ngữ kịch bản như TCL hoặc Perl (Ousterhout, 1998)
7. Sử dụng công cụ CASE và người viết chương trình hơn là phát triển hệ thống không có hỗ trợ.

Hình 8.4 Kỹ thuật ước lượng chi phí.

Kỹ thuật	Mô tả
Mô hình hoá chi phí giải thuật	Mô hình được phát triển sử dụng thông tin chi phí trong quá khứ mà nó gần với một số thước đo phần mềm (thường là kích cỡ của nó) để tính chi phí dự án. Sự ước lượng được tạo bởi thước đo và mô hình này cho biết trước các yêu cầu cần thiết.

Sự phán quyết của chuyên gia	Một số chuyên gia đề xuất kỹ thuật phát triển phần mềm và miền ứng dụng để tham khảo. Mỗi người có các ước lượng chi phí dự án. Chúng được đưa ra so sánh và thảo luận. Quá trình ước lượng được nhắc lại đến khi được tán thành và trở nên hợp lý.
Ước lượng bằng cách loại suy	Kỹ thuật này được áp dụng khi các dự án khác có cùng miền ứng dụng đã hoàn tất. Chi phí của dự án mới được ước lượng bằng cách loại suy với các dự án đã hoàn thành. Myers (Myers, 1989) đưa ra sự mô tả rất chi tiết về phương hướng này
Sự dừng đỉnh	Trạng thái dừng đỉnh được mở rộng để lấp đầy thời gian có thể sử dụng. Chi phí được ấn định bởi tài nguyên có thể sử dụng hơn là đánh giá mục tiêu. Nếu phần mềm được phân phối trong 12 tháng và 5 người có thể sử dụng, thì yêu cầu được ước lượng.
Định giá để chiến thắng	Chi phí phần mềm được ước lượng cho bất kỳ khách hàng nào có thể trả tiền cho dự án. Chi phí cho sự cố gắng phụ thuộc vào ngân sách của khách hàng chứ không phụ thuộc vào chức năng phần mềm.

Nếu những nhà quản lý không làm việc về kỹ thuật, sự xem xét kinh nghiệm có thể không giúp gì được họ trong việc ước lượng chi phí dự án phần mềm. Điều này tạo ra khó khăn cho họ trong việc đưa ra ước lượng chi phí và lịch trình chính xác.

Bạn có thể khác phục các phương hướng ước lượng trong hình 8.4 sử dụng hoặc từ phương pháp từ trên xuống hoặc từ dưới lên. Phương pháp từ trên xuống bắt đầu từ cấp hệ thống. Bạn bắt đầu với việc khảo sát các chức năng của sản phẩm và làm thể nào cung cấp các chức năng này bằng các hàm con. Chi phí của các hoạt động cấp hệ thống coi như sự quản lý tích hợp, cấu hình và hướng dẫn trong báo cáo.

Ngược lại, phương pháp từ dưới lên, bắt đầu từ cấp các thành phần. Hệ thống được phân tách thành các thành phần, và bạn ước lượng các yêu cầu để phát triển cho mỗi thành phần. Sau đó bạn cộng các chi phí thành phần để tính toán yêu cầu cho phát triển toàn bộ hệ thống.

Sự bất lợi của phương pháp từ trên xuống lại là sự thuận lợi của phương pháp từ dưới lên và ngược lại. Ước lượng từ trên xuống có thể đánh giá thấp chi phí của việc giải quyết liên kết các vấn đề kỹ thuật khó với từng phần riêng biệt như là giao diện của phần cứng không chuẩn. Không có sự chứng minh cụ thể nào của phép ước lượng được đưa ra. Ngược lại, ước lượng từ dưới lên có sự chứng minh và quan tâm tới mỗi thành phần. Tuy nhiên, phương pháp này lại có thể đánh giá thấp chi phí các hoạt động của hệ thống như sự tích hợp. Ước lượng từ dưới lên cũng đắt hơn, nó phải được khởi tạo thiết kế hệ thống để nhận biết chi phí của các thành phần.

Mỗi phương pháp ước lượng đều có những điểm mạnh và điểm yếu. Sử dụng mỗi thông tin khác nhau về các dự án và các nhóm phát triển, nếu bạn sử dụng đơn lẻ một phương pháp và các thông tin này thì sẽ không chính xác, ước lượng cuối cùng sẽ bị sai. Vì thế, cho những dự án lớn, bạn nên sử dụng một vài kỹ thuật ước lượng và so sánh kết quả của chúng. Nếu kết quả là khác nhau về cơ bản, bạn có thể không có đủ thông tin về sản phẩm hoặc quá trình phát triển, bạn nên tiếp tục quá trình ước lượng cho đến khi các kết quả là đồng nhất.



Các kỹ thuật ước lượng được áp dụng nơi văn bản yêu cầu cho hệ thống được đưa ra. Nó nên xác định tất cả các người dung và yêu cầu hệ thống. Do đó bạn có thể có những ước lượng hợp lý về chức năng của hệ thống sẽ được phát triển. Nhìn chung, các dự án thiết kế hệ thống lớn sẽ có những vắng bản yêu cầu.

Tuy nhiên, trong rất nhiều trường hợp, chi phí của nhiều dự án phải được ước lượng sử dụng các yêu cầu người dùng không thoả đáng cho hệ thống. Điều đó có nghĩa là sự ước lượng có rất ít thông tin về công việc. Phân tích yêu cầu và cụ thể hoá là đắt, và người quản lý trong công ty có thể cần có các ước lượng chi phí khởi tạo cho hệ thống trước khi họ có thể chấp nhận ngân sách để phát triển các yêu cầu cụ thể hoặc các phiên bản đầu tiên của hệ thống.

Trường hợp dưới cùng, “trả giá để dành chiến thắng” là một chiến lược chung thường sử dụng. Chú ý của việc trả giá để dành chiến thắng là nó dường như không hợp lý và không có tính thương mại. Tuy nhiên, nó có một số thuận lợi. Chi phí của dự án được chấp nhận trên cơ sở mục đích trên nguyên tắc chung. Sự điều chỉnh sau đó do khách hàng thiết lập các đặc trưng cơ bản của dự án. Các chi tiết phụ thuộc vào sự chi phí chấp nhận. Người mua và người bán phải đồng ý cái gì là chức năng hệ thống chấp nhận được. Sự yêu cầu là có thể thay đổi nhưng chi phí là không được vượt quá.

Ví dụ, một công ty trả cho một hợp đồng phát triển hệ thống phân phối nhiên liệu cho công ty dầu mà lịch phân phối nhiên liệu là tới các trạm phục vụ. Không có văn bản yêu cầu cụ thể nào cho hệ thống và ước lượng chi phí là \$900.000 như là sự cạnh tranh ngân sách các công ty dầu. Sau sự công nhận như vậy trong hợp đồng, họ đàm phán các yêu cầu cụ thể cho hệ thống và các chức năng cơ bản được phân phối, sau đó họ ước lượng thêm các chi phí cho các yêu cầu khác. Công ty dầu không cần thiết mất cái trên bởi vì nó có hợp đồng. Các yêu cầu thêm có thể dung các ngân sách trong tương lai, vì thế ngân sách công ty dầu có thể bị phá vỡ bởi cái giá phần mềm khởi điểm quá cao.

#### **8.4. Mô hình hoá chi phí thuật toán**

Mô hình hoá chi phí giải thuật sử dụng các công thức toán học để mô tả chi phí dự án trên cơ sở ước lượng dựa trên kích thước dự án, số lượng kỹ sư phần mềm, và các quá trình và các hệ số sản phẩm khác. Mô hình chi phí giải thuật có thể xây dựng bằng cách phân tích chi phí và thuộc tính của các dự án đã hoàn tất và tìm kiếm các công thức gần nhất từ kinh nghiệm thực tế.

Các mô hình chi phí giải thuật sử dụng chính để tạo các ước lượng của chi phí phần mềm, nhưng Boehm (Boehm, 2000) thảo luận về các thứ khác sử dụng ước lượng chi phí giải thuật, bao gồm ước lượng cho người đầu tư trong công ty phần mềm, ước lượng cho các chiến lược khác để giúp đánh giá rủi ro, và ước lượng cho khai báo các quyết định về việc sử dụng lại, phát triển lại hoặc mượn tài nguyên.

Trong dạng chung nhất, ước lượng chi phí giải thuật cho chi phí phần mềm có thể được coi như:

$$\text{Sự cố gắng} = A * (\text{kích thước})^B * M$$

Trong đó, A là hằng số phụ thuộc vào thực tiễn tổ chức và loại phần mềm được phát triển. Kích thước có thể là sự đánh giá kích thước mã của phần mềm hoặc ước lượng chức năng trong các hàm hoặc điểm mục tiêu. Giá trị của số mũ B thường ở giữa 1 và 1.5. M là số nhân quá trình kết hợp, sản xuất và phát triển thuộc tính, có thể phụ thuộc yêu cầu cho phần mềm và kinh nghiệm của nhóm phát triển.

Hầu hết các mô hình ước lượng thuật toán có thành phần số mũ ( $B$  trong biểu thức trên) nó kết hợp với ước lượng kích thước. Nó phản ánh trên thực tế chi phí không đơn thuần phụ thuộc tuyến tính và kích thước dự án. Như khi kích thước dự án thay đổi, chi phí chịu thêm là nhiều hơn bình thường vì giao tiếp bao trùm các nhóm lớn, thêm các sự quản lý cấu hình phức tạp, thêm các tích hợp hệ thống khó khăn, vân vân. Vì thế, một hệ thống lớn có nghĩa là một số mũ lớn.

Không may là tất cả các mô hình thuật toán phụ thuộc chủ yếu vào các khó khăn sau:

1. *Thường khó ước lượng kích thước ở giai đoạn đầu của dự án khi chỉ có các đặc điểm kỹ thuật là có thể sử dụng*, ước lượng điểm hàm và điểm mục tiêu là có thể đưa ra sớm hơn ước lượng kích thước của mã nhưng thường không chính xác.

2. *Sự ước lượng các hệ số đóng góp  $B$  và  $M$  là mang tính chủ quan*. Sự ước lượng thay đổi khi người ước lượng thay đổi, phụ thuộc vào nền tảng và kinh nghiệm với kiểu hệ thống được phát triển.

Số dòng của mã nguồn trong hệ thống phân phối là thước đo cơ bản trong hầu hết các mô hình chi phí giải thuật. Ước lượng kích thước có thể bao gồm việc ước lượng loại suy từ các dự án khác, ước lượng bằng cách chuyển đổi các điểm hàm, mục tiêu thành kích thước mã, ước lượng bằng xếp hạng kích thước thành phần hệ thống và các thành phần tham khảo để ước lượng kích thước thành phần, hoặc nó có thể đơn giản là câu hỏi của sự quyết định trong thiết kế.

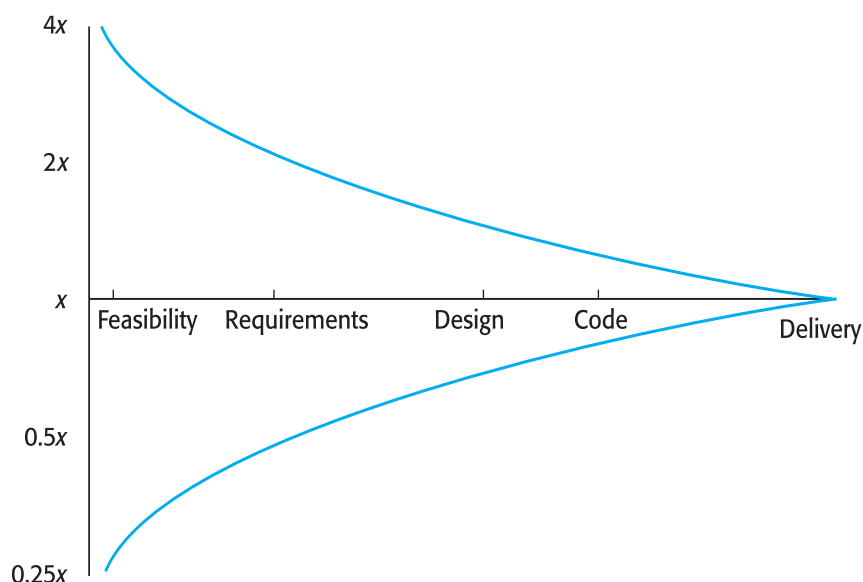
Ước lượng kích thước mã chính xác là khó trong giai đoạn đầu của dự án bởi vì kích thước mã chịu ảnh hưởng bởi các quyết định thiết kế mà thậm chí chưa có lúc này. Ví dụ, một ứng dụng mà yêu cầu sự quản lý dữ liệu phức tạp có thể sử dụng hoặc các cơ sở dữ liệu thương mại hoặc các công cụ quản lý dữ liệu của chính hệ thống. Nếu là các cơ sở dữ liệu thương mại được sử dụng, kích thước mã có thể nhỏ nhưng sự cố gắng công thêm có thể cần cố vượt qua giới hạn hiệu năng của sản phẩm thương mại.

Ngôn ngữ lập trình được sử dụng trong hệ thống phát triển cũng ảnh hưởng số dòng của mã được phát triển. Ngôn ngữ như Java có thể phải mất nhiều dòng mã cần thiết hơn C khi sử dụng. Tuy nhiên, mã bổ xung cho phép thời gian biên dịch kiểm tra chi phí có khả năng được giảm. Đưa chúng vào các báo cáo thế nào? Hơn thế nữa, nó có thể sử dụng lại đặc điểm số lượng mã từ việc xem xét lại các dự án và ước lượng kích thước có thể được điều chỉnh để đưa vào báo cáo.

Nếu bạn sử dụng mô hình ước lượng chi phí thuật toán, bạn nên phát triển việc sắp xếp các ước lượng (tồi, bình thường và tốt) hơn là các ước lượng đơn và áp dụng công thức chi phí cho tất cả. Các ước lượng có thể trở nên chính xác khi bạn hiểu về loại phần mềm được phát triển, khi bạn đã có sự kiểm tra mô hình chi phí sử dụng các dữ liệu cục bộ, và khi ngôn ngữ lập trình và phần cứng được chọn lựa từ trước.

Sự chính xác của việc đưa ra ước lượng bằng mô hình thuật toán phụ thuộc vào thông tin hệ thống có thể sử dụng. Như việc xuất phát quá trình phần mềm, nhiều trong tin trở nên có thể sử dụng và sự ước lượng trở nên chính xác hơn. Nếu sự ước lượng ban đầu cho sự cố gắng là  $x$  tháng, thì dãy này có thể từ  $0.25x$  đến  $4x$  khi hệ thống được đưa ra lần đầu. Giới hạn của quá trình phát triển được đưa ra trong Hình 8.5. Hình này được trích dẫn từ bài viết của Boehm (Boehm, 1995), phản ánh kinh nghiệm của số lượng lớn các dự án phát triển phần mềm. Tuy nhiên, trước hết nó phụ thuộc hệ thống được phân phối, sự ước lượng rất chính xác có thể được đưa ra.

Hình 8.5 Ước lượng không chắc chắn



### 8.5. Mô hình COCOMO

Một số các mô hình thuật toán được đưa ra như là nền tảng cho việc ước lượng sự cố gắng, lịch trình và chi phí cho dự án. Chúng dựa trên những khái niệm tương tự nhưng với những giá trị tham số khác nhau. Mô hình được thảo luận ở đây là mô hình COCOMO. Mô hình COCOMO là mô hình theo kinh nghiệm nhận được từ việc sưu tập dữ liệu từ một lượng lớn các dự án phần mềm. Các dữ liệu này được phân tích để thu về công thức tốt nhất phù hợp với các quan sát. Công thức liên kết kích thước của hệ thống với sản phẩm, dự án và các nhân tố trong nhóm để phát triển hệ thống.

Tôi đã chọn mô hình COCOMO vì một số lý do:

1. Đó là một tư liệu tốt, có thể sử dụng trong các miền chung và hỗ trợ bởi các miền chung và các công cụ thương mại.
2. Được sử dụng rộng rãi để đánh giá trong dãy các tổ chức.
3. Có sự kế thừa lâu dài từ những cài đặt đầu tiên vào năm 1981 (Boehm, 1981), thông qua việc chọn lọc những thứ thích hợp trong việc phát triển phần mềm Ada (Boehm và Royce, 1989), đến bản mới nhất gần đây COCOMO II, được đăng năm 2000 (Boehm và cộng sự, 2000).

Mô hình COCOMO là toàn diện, với lượng lớn các tham số. Nó rất phức tạp và không thể mô tả hoàn toàn ở đây. Hơn nữa, tôi thảo luận đơn thuần về các đặc điểm cần thiết để có thể giúp bạn hiểu về mô hình chi phí thuật toán.

Phiên bản đầu tiên của mô hình COCOMO (COCOMO 81) là mô hình cấp 3 là cấp tương ứng với phân tích chi tiết của việc ước lượng chi phí. Cấp đầu tiên (cơ sở) cung cấp ước lượng thô sơ đầu tiên; Cấp thứ 2 sửa đổi nó sử dụng một số các dự án và quá trình; và cấp chi tiết nhất đưa ra sự ước lượng cho nhiều phương diện của dự án. Hình 8.6 thể hiện công thức COCOMO cơ bản cho các loại khác nhau của dự án. Số nhân  $M$  phản ánh sản phẩm, dự án và đặc điểm nhóm.

Độ phức tạp dự án	Công thức	Mô tả
Đơn giản	$PM=2.4(KDSI)^{1.05}*M$	Các ứng dụng dễ hiểu phát triển bởi các nhóm nhỏ
Bình thường	$PM=3.0(KDSI)^{1.12}*M$	Dự án phức tạp hơn nơi các thành viên có thể có kinh nghiệm có giới hạn trong các hệ thống gần đó.
Phức tạp	$Pm=3.6(KDSI)^{1.20}*M$	Dự án phức tạp nơi phần mềm là phần mạnh trong nhóm phức tạp của phần cứng, phần mềm, sự điều chỉnh và các thủ tục hoạt động.

COCOMO 81 giả sử là phần mềm được phát triển theo quá trình thác nước ( xem Chương 4) sử dụng các ngôn ngữ lập trình cần thiết chuẩn như C hoặc FORTRAN. Tuy nhiên, nó có thay đổi cơ bản cho việc phát triển phần mềm từ bản đầu tiên được đề xuất. Từ phiên bản đầu tiên và các cải tiến đều dung các mô hình quá trình chung. Phần mềm bây giờ thường được phát triển bởi việc tập hợp các thành phần có thể dùng lại được với các hệ thống không bảo vệ và kết nối chúng với các ngôn ngữ kịch bản. Hệ thống dữ liệu chuyên sâu được phát triển sử dụng ngôn ngữ lập trình cơ sở dữ liệu như SQL và các hệ thống quản lý dữ liệu thương mại. Các phần mềm có sẵn được thiết kế lại để tạo các phần mềm mới. Công cụ CASE hỗ trợ cho các hoạt động quá trình phần mềm là có thể sử dụng được.

Để đưa sự thay đổi đó vào trong báo cáo, mô hình COCOMO II thừa nhận các phương pháp khác nhau để phát triển phần mềm như làm ra nguyên mẫu nguồn, phát triển bằng các thành phần kết cấu và sử dụng lập trình cơ sở dữ liệu. COCOMO II hỗ trợ mô hình xoắn ốc của sự phát triển (xem Chương 4) và bao gồm một vài mô hình con mà cung cấp thêm các ước lượng chi tiết. Nó có thể được sử dụng trong một loạt các sự phát triển xoắn ốc. Hình 8.7 thể hiện mô hình con COCOMO II và nơi chúng được sử dụng.

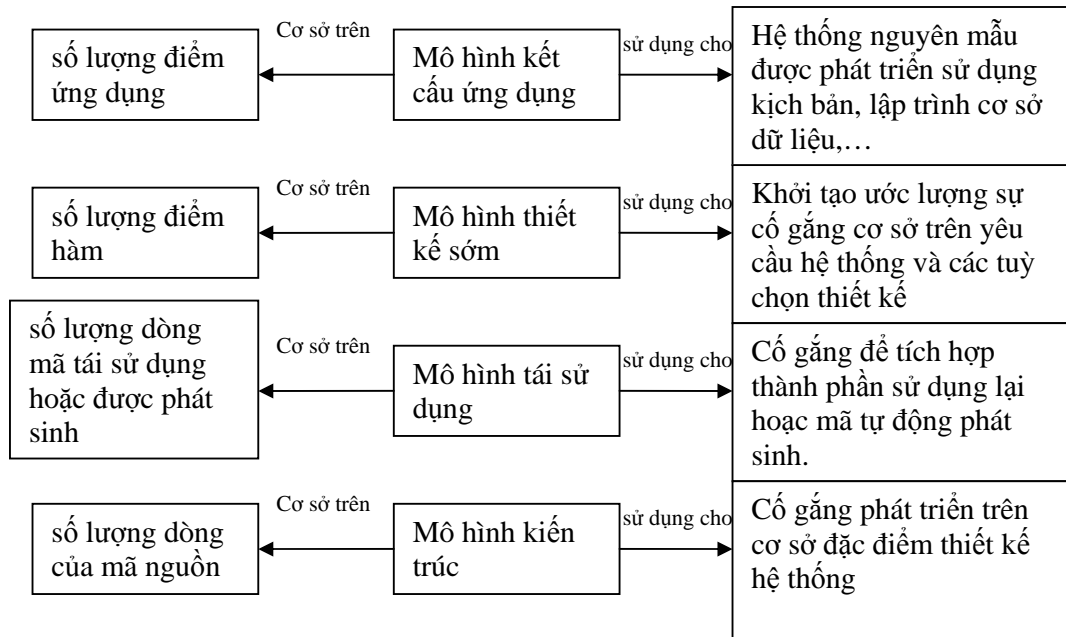
Các mô hình con mà là một phần của mô hình COCOMO II là:

1. *Mô hình kết cấu ứng dụng* Nó giả sử rằng hệ thống được tạo từ các thành phần, kịch bản hoặc lập trình cơ sở dữ liệu có thể sử dụng lại. Nó được thiết kế để tạo ước lượng của sự phát triển nguyên mẫu. Ước lượng kích thước phần mềm là cơ sở của điểm ứng dụng, và công thức kích thước/năng suất đơn giản được sử dụng để ước lượng yêu cầu sự cố gắng. Điểm ứng dụng là gần giống điểm mục tiêu được thảo luận trong phần 8.1, nhưng tên thay đổi để tránh sự lộn xộn với mục tiêu trong phát triển định hướng mục tiêu.
2. *Mô hình thiết kế sớm* Mô hình này được sử dụng trong giai đoạn đầu của thiết kế hệ thống sau khi các yêu cầu được thành lập. Sự ước lượng là cơ sở của điểm hàm, nó được chuyển đổi từ số dòng của mã nguồn. Công thức theo dạng chuẩn được thảo luận ở trên với tập hợp được làm đơn giản của 7 số nhân.
3. *Mô hình sử dụng lại* là mô hình sử dụng để tính toán yêu cầu để tích hợp các thành phần có thể sử dụng lại và/hoặc mã chương trình được tự động

tạo ra bằng thiết kế hoặc chương trình thay đổi công cụ. Nó thường được sử dụng trong liên kết với các mô hình kiến trúc.

4. *Mô hình kiến trúc* khi kiến trúc hệ thống được thiết kế, sự chính xác các ước lượng của kích thước phần mềm có thể được tạo ra. Đáp lại mô hình này sử dụng công thức chuẩn cho ước lượng chi phí thảo luận ở trên. Tuy nhiên, nó bao gồm nhiều mở rộng tập hợp 17 số nhân phản ánh khả năng nhân viên, sản phẩm và đặc điểm dự án.

Hình 8.7 Mô hình COCOMO II



Tất nhiên, trong các hệ thống lớn, các phần khác nhau có thể được phát triển sử dụng các kỹ thuật khác nhau, và bạn có thể không phải ước lượng tất cả các phần của hệ thống ở cùng cấp độ của sự chính xác. Trong trường hợp này, bạn có thể sử dụng các mô hình con thích hợp cho mỗi phần của hệ thống và kết hợp kết quả thành một ước lượng hỗn hợp.

#### 8.5.1. Mô hình kết cấu ứng dụng.

Mô hình kết cấu ứng dụng được giới thiệu trong vào COCOMO II để hỗ trợ sự ước lượng yêu cầu sự cố gắng cho việc tạo nguyên mẫu dự án và cho các dự án mà phần mềm được phát triển bởi các thành phần có sẵn. Đó là cơ sở của ước lượng điểm khả năng ứng dụng (điểm mục tiêu) chia cho ước lượng chuẩn của năng suất điểm ứng dụng. Sự ước lượng sau đó được điều chỉnh theo sự khó khăn của việc phát triển mỗi điểm mục tiêu (Boehm và cộng sự, 2000) năng suất người lập trình cũng phụ thuộc vào kinh nghiệm người phát triển và khả năng như là khả năng của công cụ CASE sử dụng hỗ trợ phát triển. Hình 8.8 thể hiện năng suất điểm mục tiêu đề xuất bởi người phát triển mô hình (Boehm và cộng sự, 1995).

Kết cấu ứng dụng thường bao gồm việc sử dụng lại phần mềm, và một số trong tổng số điểm ứng dụng trong hệ thống có thể trở thành công cụ với các thành phần có thể sử dụng lại. Do đó, bạn phải điều chỉnh cơ sở ước lượng trong tổng số điểm ứng dụng để đưa vào báo cáo tỷ lệ của thành phần sử dụng lại. Vì thế, công thức cuối cùng cho việc tính toán sự nỗ lực cho hệ thống nguyên mẫu là:

$$PM = (NAP * (1 - \% \text{sử dụng lại} / 100)) / PROD$$

PM là ước lượng nỗ lực của con người-tháng. NAP là tổng số của điểm ứng dụng trong hệ thống phân phối. % sử dụng lại là ước lượng của tổng số mã sử dụng lại trong sự phát triển. PROD là năng suất điểm mục tiêu được đưa ra trong hình 8.8. Mô hình giả thiết là không phải tốn công cho quá trình sử dụng lại tài nguyên.

### 8.5.2. Mô hình thiết kế sớm

Mô hình này được sử dụng trong yêu cầu người dùng đã đồng ý và giai đoạn khởi đầu của quá trình thiết kế hệ thống đang được thực hiện. Tuy nhiên, bạn không cần những thiết kế kiến trúc chi tiết để tạo sự ước lượng ban đầu. Mục tiêu của bạn trong giai đoạn này nên tạo các ước lượng gần đúng thích hợp. Do đó, bạn tạo những giả thiết đơn giản đa dạng, như sự cố gắng bao gồm trong việc tích hợp mã sử dụng lại là bằng không. Ước lượng thiết kế sớm là thường được sử dụng trong các khảo sát lựa chọn nơi bạn cần so sánh các cách khác nhau của các công cụ người dùng yêu cầu.

Ước lượng được đưa ra trong giai đoạn này lấy cơ sở trên công thức chuẩn cho mô hình thuật toán, cụ thể là:

$$\text{Sự cố gắng} = A * (\text{kích thước})^B * M$$

Trên cơ sở trong tập hợp dữ liệu lớn. Boehm đề xuất rằng hệ số A nên bằng 2.94. Kích thước của hệ thống được biểu diễn bằng KSLOC, mà nó là số lượng của hàng nghìn dòng trong mã nguồn. Bạn tính toán KSLOC bằng ước lượng số lượng điểm hàm trong phần mềm. Sau đó bạn sử dụng bảng chuẩn mà nó liên kết kích thước của phần mềm tới điểm hàm cho các ngôn ngữ lập trình khác nhau để tính toán ước lượng của kích thước hệ thống bằng KSLOC.

Số mũ B phản ánh yêu cầu nảy sinh thêm khi kích thước của dự án tăng. Nó không phải chung cho các kiểu khác nhau của hệ thống, như trong COCOMO 81, nhưng nó có thể thay đổi trong khoảng 1.1 tới 1.24 phụ thuộc vào tính khác thường của dự án, tính mềm dẻo của sự phát triển, sự liên kết trong nhóm phát triển, trình độ sử lý thành thạo của tổ chức. Tôi thảo luận về giá trị của số mũ trên được tính toán sử dụng các tham số được mô tả trong Mô hình kiến trúc COCOMO II.

Hệ số M trong mô hình COCOMO II lấy cơ sở trên tập hợp bảy dự án và đặc tính sử lý mà nó ảnh hưởng tới sự ước lượng. Nó có thể gia tăng hoặc giảm yêu cầu sự cố gắng. Đặc tính sử dụng trong mô hình thiết kế sớm là tích số của độ đáng tin cậy và phức tạp (RCPX), yêu cầu sử dụng lại (RUSE), nền tảng khó (PDIF), khả năng con người (PERS), kinh nghiệm con người (PREX), lịch trình (SCED), và tiện nghi trợ giúp (FCIL). Bạn ước lượng giá trị của các thuộc tính sử dụng sáu điểm mức độ mà 1 tương ứng với giá trị rất thấp cho các hệ số và 6 tương ứng với giá trị rất cao.

Kết quả của sự tính toán sự cố gắng là:

$$PM = 2.94 * (\text{kích thước})^B * M$$

Trong đó:

$$M = PERS * RCPX * RUSE * PDIF * PREX * FCIL * SCED$$

### 8.5.3. Mô hình sử dụng lại

Như đã thảo luận trong các chương trước, phần mềm sử dụng lại là chung bây giờ, và hầu hết các hệ thống lớn có tỷ lệ lớn của mã là được sử dụng lại từ trước khi phát triển. Mô hình sử dụng lại được sử dụng để ước lượng yêu cầu sự cố gắng để tích hợp thành phần có thể sử dụng lại hoặc mã tự phát.

COCOMO II quan tâm đến sử dụng lại mã trong hai loại. Mã hộp đen là mã có thể sử dụng lại mà không cần biết về mã hoặc sửa chữa nó. Sự cố gắng phát triển với mã hộp đen là bằng không. Mã mà có thể thay đổi để tích hợp nó với mã mới hoặc các phần khác sử dụng lại gọi là mã hộp trắng. Một số cố gắng phát triển là yêu cầu được sử dụng lại nó vì phải hiểu và sửa chữa lại trước thì nó mới có thể làm việc được chính xác trong hệ thống.

Thêm vào nữa, rất nhiều hệ thống bao gồm các mã tự phát từ các chương trình chuyển đổi nó sinh ra mã từ các hệ thống mẫu. Đó là dạng của sử dụng lại trong các mẫu chuẩn phức tạp của người viết. Các mẫu hệ thống được phân tích, và mã cơ sở của các mẫu chuẩn với các chi tiết thêm vào từ hệ thống mẫu được sinh ra. Mô hình sử dụng lại COCOMO II bao gồm các mô hình riêng biệt để ước lượng chi phí liên quan với mã tự phát.

Cho mã mà được tự sinh ra, mô hình ước lượng số lượng của yêu cầu con người trong các tháng để tích hợp mã này. Công thức cho sự ước lượng này là:

$$PM_{Auto} = (ASLOC * AT / 100) / ATPROD \quad // \text{ ước lượng cho mã tự phát.}$$

AT là tỷ lệ của mã thích nghi mà được tự động sinh ra và ATPROD là năng suất của các kỹ sư trong tích hợp mã. Boehm và cộng sự (Boehm, 2000) có phép đo ATPROD vào khoảng 2,400 báo cáo nguồn trên tháng. Vì thế, nếu có tổng số 20.000 dòng trong mã hộp trắng sử dụng lại trong hệ thống và 30% của chúng là mã tự động sinh ra, thì yêu cầu cố gắng để tích hợp mã tự phát là:

$$(20,000 * 30 / 100) / 2400 = 2.5 \text{ Người tháng.} \quad // \text{VD mã tự phát}$$

Các thành phần khác của mô hình sử dụng lại khi hệ thống bao gồm một số mã mới và một số sử dụng lại thành phần hộp trắng được tích hợp. Trong trường hợp này, mô hình sử dụng lại không được tính toán ngay lập tức. Hơn nữa, trên cơ sở của số dòng của mã được sử dụng lại, nó tính toán các đặc trưng được mô tả tương đương số dòng của mã mới.

Vì thế, nếu 30.000 dòng mã được sử dụng, ước lượng kích thước mới tương đương có thể là 6.000. Tất nhiên là, sử dụng lại 30000 dòng mã là tương đương với viết 6000 dòng mã mới. Đặc điểm tính toán là thêm số lượng của dòng mã mới vào sự phát triển trong mô hình kiến trúc COCOMO II.

Ước lượng trong mô hình sử dụng lại là:

ASLOC - số dòng của mã trong thành phần phải được thích nghi.

ESLOC - Số dòng tương đương của mã nguồn mới.

Công thức sử dụng để tính toán ESLOC đưa vào báo cáo yêu cầu sự cố gắng cho việc tìm hiểu phần mềm, để tạo sự thay đổi cho mã tái sử dụng và thay đổi hệ thống để tích hợp với mã đó. Nó cũng đưa ra báo cáo số lượng của mã mà nó tự động sinh ra khi cố gắng phát triển được tính toán, như được giải thích sớm trong mục này.

Công thức sau được sử dụng tính toán số dòng tương đương số dòng của mã nguồn:

$$ESLOC = ASLOC * (1 - AT / 100) * AAM$$

ASLOC được giảm tùy theo tỷ lệ của mã tự động sinh ra. AAM là Adaptation Adjustment Multiplier (Hệ số điều chỉnh thích nghi), nó đưa ra báo cáo yêu cầu cho mã tái sử dụng. Đơn giản, AAM là tổng của ba thành phần:

1. Thành phần thích nghi (AAF) nó mô tả chi phí cho tạo sự thay đổi của mã tái sử dụng. Nó bao gồm các thành phần thiết kế, thay đổi mã và tích hợp.

2. Thành phần tìm hiểu (SU) mô tả chi phí của việc tìm hiểu mã để sử dụng lại và sự hiểu rõ mã của kỹ sư với mã. SU sắp xếp từ 50 cho các mã phi cấu trúc phức tạp đến 10 cho các bản viết tốt, mã có định hướng mục tiêu.
3. Hệ số đánh giá (AA) mô tả chi phí của việc tạo quyết định sử dụng lại. Nghĩa là, một số phân tích luôn được yêu cầu để quyết định có nên sử dụng lại mã hay không, và nó bao gồm trong chi phí AA, AA thay đổi từ 0 đến 8 phụ thuộc vào số lượng phân tích yêu cầu.

Mô hình sử dụng lại là mô hình phi tuyến. Một số sự cố gắng được yêu cầu nếu sử dụng lại là xem xét để tạo đánh giá có nên sử dụng lại hay không. Hơn thế nữa, ngày càng nhiều phần tái sử dụng được xem xét, chi phí trên đơn vị mã sử dụng lại giảm như sự hiểu biết đầy đủ và đánh giá chi phí là đã qua nhiều dòng mã.

#### 8.5.4. Đẳng cấp kiến trúc

Đẳng cấp kiến trúc là chi tiết nhất của mô hình COCOMO II. Nó được sử dụng một lần khi thiết kế kiến trúc mở đầu cho hệ thống có thể sử dụng cho hệ thống con cấu trúc đã biết.

Sự ước lượng đưa ra trong đẳng cấp kiến trúc lấy cơ sở trên các công thức cơ bản ( $PM = A * (\text{kích thước})^B * M$ ) sử dụng trong ước lượng thiết kế sớm. Tuy nhiên, ước lượng kích thước cho phần mềm nền chính xác hơn trong giai đoạn này của quá trình ước lượng. Thêm nữa, có rất nhiều tập hợp thuộc tính được mở rộng của sản phẩm, quá trình và tổ chức (17 thay vì 7) được sử dụng để chọn lọc tính toán ban đầu. Có thể sử dụng nhiều hơn các thuộc tính trong giai đoạn này vì bạn có nhiều thông tin về phần mềm được phát triển và quá trình phát triển.

Ước lượng của kích thước mã trong đẳng cấp kiến trúc là sử dụng ba thành phần:

1. Ước lượng tổng số dòng mã mới được sử dụng để phát triển.
2. Ước lượng tương đương số dòng của mã nguồn (ESLOC) tính toán dựa trên mô hình sử dụng lại.
3. Ước lượng số dòng của mã phải sửa để phù hợp với yêu cầu.

Ba ước lượng trên được cộng vào để có tổng kích thước mã trong KSLOC mà bạn sử dụng trong công thức tính toán sự nỗ lực. Thành phần cuối cùng của sự ước lượng - tổng số dòng của mã đã thay đổi - phản ánh trên thực tế là yêu cầu phần mềm luôn thay đổi. Các chương trình hệ thống phải phản ánh các thay đổi đó để cộng thêm mã cho việc phát triển. Tất nhiên, ước lượng số dòng của mã mà nó sẽ thay đổi là không dễ, và nó sẽ thường trở nên không chắc chắn trong các đặc điểm hơn là trong ước lượng sự phát triển.

Số mũ giới hạn (B) trong công thức tính toán sự cố gắng có ba giá trị khả thi trong COCOMO I. Nó có liên hệ tới cấp của độ phức tạp dự án. Như dự án trở nên phức tạp hơn, hiệu quả của việc tăng kích thước hệ thống trở nên nhiều ý nghĩa. Tuy nhiên, tổ chức thực tiễn và sản xuất tốt có thể điều chỉnh sự lãng phí. Nó được công nhận trong COCOMO II, nơi mà dãy các giá trị của số mũ B là liên tục hơn là rời rạc. Số mũ trên lấy cơ sở trên năm hệ số phân mức, được đưa ra trong hình 26.9. Các nhân tố này là tỷ lệ với sáu mức điểm từ rất thấp đến rất cao (5 đến 0). Bạn nên thêm vào tỷ lệ, chia chúng cho 100 và cộng vào kết quả 1.01 để có số mũ có thể sử dụng.

Để minh họa điều này, tưởng tượng rằng một tổ chức đang nói về dự án trong miền mà có ít kinh nghiệm đi trước. Khách hàng dự án không xác định quá trình được sử dụng và không cho phép thời gian trong lịch trình dự án cho phân tích mạo hiểm. Nhóm phát triển mới phải đưa đồng thời các công cụ của hệ thống. Tổ chức đã gần đây đã đưa chương



trình phát triển quá trình vào và có tỷ lệ như trong tổ chức cấp 2 theo mô hình CMM ( xem Chương 28). Giá trị khả thi cho sử dụng tỷ lệ của số mũ tính toán bởi:

Hình 8.9 Hệ số mức sử dụng trong tính toán số mũ trong COCOMO II

Hệ số mức	Giải thích
Có tiền lệ	Phản ánh kinh nghiệm trước đó của tổ chức với loại dự án đó. Rất thấp là không có kinh nghiệm; Rất cao là tổ chức là một nhóm rất thành công với miền ứng dụng đó.
Sự mềm dẻo của dự án	Phản ánh mức độ của sự mềm dẻo trong quá trình phát triển. Rất thấp là các quá trình bắt buộc được sử dụng. Rất cao là tập hợp khách hàng chỉ là các mục tiêu chung.
Giải quyết kiến trúc/sự mạo hiểm	Phản ánh sự mở rộng của các phân tích mạo hiểm được thực hiện. Rất thấp là ít sự phân tích. Rất cao là các phân tích hoàn thành và vượt qua mạo hiểm.
Liên kết trong nhóm	Phản ánh tại sao một nhóm hiểu từng người và làm việc chung lại tốt. Rất thấp là rất khó tương tác; Rất cao là có sự kết hợp và hiệu quả và không có các vấn đề xấu trong giao tiếp.
Sử lý thuần thực	Phản ánh sự sử lý thuần thực của tổ chức. Các tính toán của giá trị này phụ thuộc trong Các câu hỏi về sự thành thực CMM, nhưng sự ước lượng có thể hoàn thành bằng cách trừ sử lý thuần thực CMM từ cấp 5.

- *Có tiền lệ* Đó là một dự án mới cho tổ chức - Tỷ lệ Thấp (4)
- *Sự mềm dẻo của dự án* không có sự bao gồm khách hàng - Tỷ lệ rất cao (1)
- *Giải quyết kiến trúc/sự mạo hiểm* không có các phân tích được thực hiện - Tỷ lệ rất thấp (5)
- *Liên kết trong nhóm* Nhóm mới nên không có thông tin – tỷ lệ trung bình (3)
- *Sử lý thuần thực* Một số điều khiển quá trình trong nhóm - tỷ lệ trung bình (3)

Tổng các giá trị là 16, vì thế tính toán số mũ bằng cách thêm 0.16 và 1.01, được 1.17.

Các thuộc tính ( Hình 8.10) được sử dụng trong điều chỉnh ước lượng ban đầu và tạo hệ số M trong mô hình kiến trúc được chia ra 4 lớp:

1. Thuộc tính sản phẩm liên quan đến đặc tính yêu cầu của sản phẩm được phát triển
2. Thuộc tính máy tính là bị áp đặt trong phần mềm bởi nền tảng phần cứng.
3. Thuộc tính con người là số nhân mà nó lấy kinh nghiệm và khả năng của con người trong công việc trong dự án để đưa ra báo cáo.
4. Thuộc tính dự án là liên quan với các đặc tính riêng biệt của dự án phát triển phần mềm

Thuộc tính	Phân loại	Mô tả
RELY	Sản phẩm	Yêu cầu độ tin tưởng hệ thống
CPLX	Sản phẩm	Độ phức tạp của các phần hệ thống
DOCU	Sản phẩm	Báo cáo mở rộng theo yêu cầu
DATA	Sản phẩm	Kích thước của cơ sở dữ liệu được sử dụng.
RUSE	Sản phẩm	Yêu cầu tỷ lệ thành phần sử dụng lại
TIME	Máy tính	Sự phụ thuộc thời gian thực thi.
PVOL	Máy tính	Sự không ổn định của nền tảng phát triển
STOR	Máy tính	Bộ nhớ phụ thuộc vào máy
ACAP	Máy tính	Khả năng phân tích dự án
PCON	Con người	Khả năng làm việc liên tục của con người
PCAP	Con người	Khả năng lập trình viên
PEXP	Con người	Kinh nghiệm lập trình viên trong miền dự án
AEXP	Con người	Kinh nghiệm phân tích trong miền dự án
LTEX	Con người	Kinh nghiệm ngôn ngữ và công cụ
TOOL	Dự án	Công cụ sử dụng trong phần mềm
SCED	Dự án	Nén lịch trình phát triển
SITE	Dự án	Mở rộng làm việc kết hợp và giao tiếp chung

Hình 8.11 thể hiện chi phí định hướng ảnh hưởng đến ước lượng sự cố gắng. Tôi đã đưa giá trị của số mũ là 1.17 như đã thảo luận ở trên và giả sử RELY, CPLX, STOR, TOOL và SCED là những chi phí định hướng chủ yếu trong dự án. Tất cả trong chi phí định hướng khác có giá trị là 1, vì thế chúng không ảnh hưởng đến sự tính toán sự cố gắng.

Hình 8.11 Sự ảnh hưởng của chi phí định hướng để ước lượng

Giá trị số mũ	1.17
Kích thước hệ thống (bao gồm các hệ số tái sử dụng và các yêu cầu dễ thay đổi)	128000 DSI
<b>Ước lượng ban đầu COCOMO không có chi phí định hướng</b>	<b>730 người-các tháng</b>
Khả năng tin tưởng	Rất cao, hệ số=1.39

Độ phức tạp	Rất cao, hệ số=1.3
Phụ thuộc bộ nhớ	Cao, hệ số=1.21
Công cụ sử dụng	Cao, hệ số=1.12
Lịch trình	Tăng nhanh, hệ số=1.29
<b>Điều chỉnh ước lượng COCOMO</b>	<b>2306 người-các tháng</b>
Khả năng tin tưởng	Rất thấp, hệ số=0.75
Độ phức tạp	Rất thấp, hệ số=0.75
Phụ thuộc bộ nhớ	Không, hệ số=1
Công cụ sử dụng	Rất cao, hệ số=0.72
Lịch trình	Bình thường, hệ số=1
<b>Điều chỉnh ước lượng COCOMO</b>	<b>295 người-các tháng</b>

Trong hình 8.11 có gán các giá trị lớn nhất và nhỏ nhất đến các chi phí định hướng chính để thể hiện làm thế nào chúng ảnh hưởng đến ước lượng sự cố gắng. Các giá trị được lấy từ COCOMO II (Boehm). Bạn có thể thấy là các giá trị cao cho chi phí định hướng dẫn việc ước lượng chi phí là hơn ba lần ước lượng ban đầu, ngược lại giá trị thấp giảm ước lượng chỉ còn một phần ba so với ban đầu. Sự khác biệt lớn của hai loại này là sự chuyển giao kinh nghiệm giữa các miền ứng dụng.

Công thức đề xuất bởi người phát triển mô hình COCOMO II phản ánh kinh nghiệm và dữ liệu của họ, nhưng nó là mô hình rất phức tạp để hiểu và sử dụng. Có rất nhiều thuộc tính và vấn đề không chắc chắn để ước lượng giá trị của nó. Cơ bản là mỗi người dùng của mô hình nên kiểm tra kích cỡ mô hình và giá trị thuộc tính theo dữ liệu dự án lịch sử của nó, như nó sẽ phản ánh tình huống ảnh hưởng đến mô hình.

Trong thực tiễn, tuy nhiên, một số ít các tổ chức có đủ dữ liệu từ các dự án quá khứ trong dạng mô hình hỗ trợ. Thực tế sử dụng COCOMO II vì thế phải bắt đầu với các giá trị chung cho các thuộc tính của mô hình, và nó là không thể cho người dùng để biết làm thế nào liên kết chúng. Điều đó có nghĩa là thực tế sử dụng COCOMO II là bị hạn chế. Các tổ chức rất lớn có thể có tài nguyên dư để thích nghi và sử dụng mô hình COCOMO II. Tuy nhiên, phần lớn các công ty, chi phí cho việc xác định và học tập để sử dụng mô hình thuật toán như COCOMO là rất cao mà nó không thể áp dụng.

### 8.6. Mô hình chi phí giải thuật trong kế hoạch dự án

Một trong các mô hình hoá chi phí giải thuật lớn nhất là so sánh các cách khác nhau của việc đầu tư cho việc giảm chi phí dự án. Đó là sự quan trọng riêng biệt nơi bạn có thể tạo chi phí phần cứng/phần mềm phi thương mại và nơi bạn có thể có các nhân viên mới với khả năng đặc biệt. Mô hình mã giải thuật giúp bạn đánh giá sự mạo hiểm của mỗi tùy chọn. Áp dụng mô hình chi phí để biểu thị tài chính mà nó kết hợp với các quyết định quản lý khác nhau.

Xem xét hệ thống phức tạp với các cuộc thí nghiệm. Các cuộc thí nghiệm phải đáng tin cậy và chặt chẽ. Số lượng của các chỗ sơ hở phải là nhỏ nhất. Trong giới hạn của mô hình COCOMO, các hệ số của sự phụ thuộc vào máy tính và độ tin cậy là lớn hơn 1.

Có 3 thành phần được đưa ra báo cáo trong việc chi phí dự án:

1. Chi phí của phần cứng mục tiêu để thực thi hệ thống.

2. Chi phí cho nền tảng (máy tính và phần mềm) để phát triển hệ thống.
3. Chi phí của yêu cầu cố gắng cho phát triển dự án.

Trong hình 8.13 thể hiện một số tùy chọn có thể cho dự án. Nó bao gồm việc thêm các phần cứng mục tiêu để giảm chi phí phần mềm để đầu tư tốt hơn các công cụ phát triển.

Thêm chi phí phần cứng có thể được chấp nhận bởi vì hệ thống là hệ thống chuyên dụng. Nếu phần cứng là phức tạp trong các sản phẩm tiêu dung, tuy nhiên, đầu tư trong phần cứng để giảm chi phí phần mềm sẽ làm tăng số đơn vị của sản phẩm, bất chấp số lượng được bán.

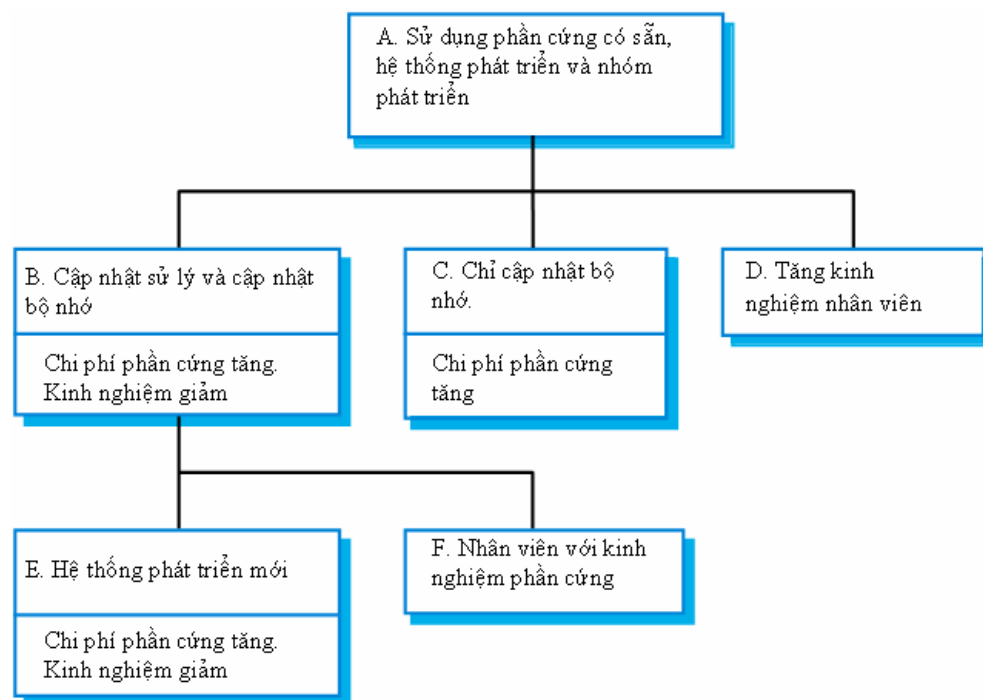
Hình 8.13 thể hiện phần cứng, phần mềm và tổng chi phí cho tùy chọn A-F thể hiện trong hình 8.12. Áp dụng mô hình COCOMO II chi phí định hướng báo trước sự cố gắng là 45 người- các tháng để phát triển hệ thống phần mềm phức tạp cho ứng dụng. Chi phí trung bình cho mỗi người-tháng là \$15000.

Các hệ số liên quan là cơ sở để lưu trữ và thực thi phụ thuộc thời gian (TIME và STOR), khả năng sử dụng của công cụ trợ giúp để phát triển hệ thống (TOOL), và kinh nghiệm của nhóm phát triển (LTEX). Trong tất cả các tùy chọn, hệ số tin tưởng được (RELY) là 1.39, nó cho biết ý nghĩa của việc phát triển các hệ thống tin cậy.

Chi phí phần mềm (SC) được tính như sau:

$$SC = \text{Ước lượng sự cố gắng} * RELY * TIME * STOR * TOOL * EXP * \$15000$$

Tùy chọn A mô tả chi phí của việc xây dựng hệ thống với các trợ giúp và nhân viên có sẵn. Nếu các mô tả là cơ sở cho so sánh. Tất cả các tùy chọn khác bao gồm phí tổn phần cứng hoặc tuyển thêm nhân viên. Tùy chọn B thể hiện sự cập nhật phần cứng là không cần thiết giảm chi phí. Các nhân viên thiếu kinh nghiệm với phần cứng mới sẽ làm tăng các hệ số STOR và TIME. Nó thực sự thêm chi phí hiệu quả để cập nhật bộ nhớ hơn là cấu hình máy tính.



Hình 8.12: Tùy chọn quản lý

Hình 8.13 Chi phí của tùy chọn quản lý

Bảng 8.13 Chi phí của các tùy chọn quản lý

Tùy chọn	Rely	Stor	Time	Tools	Ltex	Tổng công	Chi phí phần mềm	Chi phí phần cứng	Tổng chi phí
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	1.39	1.06	1.11	0.86	0.84	51	769008	100000	897490
EX	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

Tùy chọn D đưa ra chi phí thấp nhất cho tất cả các đánh giá cơ bản. Không phí tồn phần cứng nào được thêm vào nhưng cần thuê nhân viên mới cho dự án. Nếu công ty có sẵn điều này thì đây có thể là tùy chọn tốt nhất. Nếu không, họ phải được thuê từ bên ngoài, một việc cần nhiều chi phí và có nhiều rủi ro. Điều này có nghĩa là lợi thế về chi phí của tùy chọn này không lớn như được đưa ra bởi bảng 8.13. Tùy chọn C tiết kiệm \$50000 với việc gần như không có rủi ro kèm theo. Những nhà quản lý dự án bảo thủ có thể thích tùy chọn này hơn là tùy chọn nhiều rủi ro hơn D.

Các so sánh chỉ ra sự quan trọng của kinh nghiệm nhân viên như là một thừa số. Nếu nhân lực có chất lượng tốt với kinh nghiệm phù hợp được thuê thì có thể giảm đi nhiều chi phí dự án. Điều này nhất quán với thảo luận về hệ số năng suất trong phần 26.1. Nó cũng bộc lộ rằng đầu tư phần cứng mới và công cụ có thể không hiệu quả về chi phí. Một số kỹ sư có thể thích tùy chọn này vì nó cho họ cơ hội học hỏi hệ thống mới. Tuy nhiên, sự mất mát kinh nghiệm tác động nhiều lên chi phí hệ thống hơn sự tiết kiệm từ việc sử dụng hệ thống phần cứng mới.

### 8.7. Nhân viên và khoảng thời gian của dự án

Cũng như việc ước lượng công lao động cần thiết để phát triển hệ thống và chi phí chung của dự án., người quản lý dự án còn phải ước lượng thời gian dự án cần để phát triển và khi nào nhân viên cần làm việc cho dự án. Thời gian phát triển dự án được gọi là lịch trình dự án. Dần dần, các tổ chức sẽ yêu cầu thời gian phát triển dự án ngắn hơn để sản phẩm của họ đưa ra thị trường trước đối thủ.

Mối quan hệ giữa số nhân viên đang làm việc cho dự án, tổng công lao động cần và thời gian phát triển dự án là không tuyến tính. Khi số nhân viên tăng, có thể cần nhiều công lao động hơn. Nguyên nhân là họ sử dụng nhiều thời gian hơn để giao tiếp và xác định giao diện giữa các phần của hệ thống mà được người khác phát triển. Gấp đôi số nhân viên (ví dụ) do đó không có nghĩa là thời gian dự án còn một nửa.

Mô hình COCOMO bao gồm công thức để ước lượng lịch biểu thời gian(TDEV) cần để hoàn thành dự án. Công thức tính thời gian là giống nhau cho tất cả các cấp COCOMO:

$$TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$$

PM là sự ước tính công lao động và B là số mũ được tính, như đã thảo luận ở trên (B là 1 cho mô hình nguyên mẫu sớm) Sự ước tính này dự đoán lịch trình trung bình cho dự án.

Tuy nhiên, lịch trình dự án được dự đoán và lịch trình cần bởi kế hoạch dự án không nhất thiết giống nhau. Lịch trình được lên kế hoạch có thể ngắn hơn hoặc dài hơn kế hoạch dự án trung bình được dự đoán. Tuy nhiên, rõ ràng là có giới hạn cho việc mở rộng thay đổi lịch trình, và mô hình COCOMO II dự đoán điều này:

$$TDV = 3 \times (PM)^{(0.33+0.2*(B-1.01))} \times SCEDPercentage/100$$

SCEDPercentage là phần trăm tăng thêm hoặc giảm đi trong lịch trình trung bình. Nếu các con số được dự đoán thì khác nhiều so với lịch trình được lên kế hoạch, nó ám chỉ rằng có rủi ro cao trong vấn đề phát hành phần mềm khi được lên kế hoạch.

Để minh họa tính toán lịch trình phát triển COCOMO, giả sử rằng 60 tháng công được ước lượng để phát triển một hệ thống phần mềm. (tùy chọn C trong bảng 8.12). Giả sử rằng giá trị của số mũ B là 1.17. Từ phương trình lịch trình, thời gian cần để hoàn thành dự án là :

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ tháng.}$$

Trong trường hợp này, không có sự nén hay mở rộng lịch trình, vì số hạng cuối cùng trong công thức không ảnh hưởng đến việc tính toán.

Một hàm ý thú vị của mô hình COCOMO là thời gian cần để hoàn thành dự án là một hàm của tổng công lao động cần cho dự án. Nó không phụ thuộc vào số kỹ sư phần mềm đang làm việc. Điều này xác nhận quan điểm là khi thêm nhân lực cho dự án mà có lịch trình không chắc chắn sẽ giúp lịch trình được thu hồi lại. Myers(Myers, 1989) thảo luận vấn đề tăng tốc lịch trình. Ông đề xuất rằng dự án có thể rơi vào vấn đề lớn nếu họ có phát triển phần mềm mà không được cho đủ thời gian.

Việc chia công lao động cần thiết cho dự án bởi lịch trình phát triển không đưa ra chỉ dẫn có ích về số người cần cho đội dự án. Nói chung, chỉ một số nhỏ nhân lực ở lúc bắt đầu dự án để thực hiện thiết kế đầu. Đội sau đó xây dựng tới cao điểm trong suốt quá trình phát triển và kiểm thử hệ thống, và cuối cùng kích cỡ nhóm biến đổi khi bắt đầu triển khai. Việc xây dựng nhanh nhân viên dự án được đưa ra để tương quan với việc lịch trình dự án không đúng thời hạn. Người quản lý dự án do đó nên tránh sớm thêm quá nhiều nhân viên vào dự án trong chu kỳ sống của nó.

Việc xây dựng nguồn lực lao động có thể được mô hình hóa bằng cái gọi là đường cong Rayleigh (Londeix, 1987) và mô hình ước lượng của Putnam (Putnam, 1978) cái mà

kết hợp mô hình của bố trí nhân viên dự án dựa vào các đường cong này. Mô hình Putnam cũng bao gồm thời gian phát triển như là nhân tố chìa khóa. Khi thời gian dự án được giảm xuống, công lao động cần để phát triển hệ thống tăng theo hàm mũ.

#### **Những điểm chính:**

- Không có mối quan hệ tất yếu giữa giá cần trả cho một hệ thống và chi phí phát triển của nó. Nhân tố tổ chức có thể muốn giá phải trả tăng để bù cho rủi ro, hoặc tăng để có lợi thế cạnh tranh.
- Những nhân tố tác động lên năng suất phần mềm bao gồm thái độ cá nhân (nhân tố nổi trội), kinh nghiệm miền, quá trình phát triển, quy mô của dự án, công cụ hỗ trợ và môi trường làm việc.
- Phần mềm thường được định giá để thu được hợp đồng, và chức năng của hệ thống sau đó được điều chỉnh để phù hợp với giá được ước lượng.
- Có nhiều kỹ thuật ước lượng chi phí phần mềm khác nhau. Trong việc chuẩn bị đánh giá, một vài kỹ thuật khác nhau nên được sử dụng. Nếu các đánh giá chênh nhau nhiều, thì có nghĩa là có các thông tin đánh giá không tương xứng.
- Mô hình chi phí COCOMO II là mô hình chi phí thuật toán được phát triển tốt mà tính đến các thuộc tính dự án, sản phẩm, phần cứng và cá nhân khi công thức hóa một ước lượng chi phí. Nó cũng bao gồm phương tiện ước lượng lịch trình phát triển.
- Mô hình chi phí thuật toán có thể được sử dụng để hỗ trợ phân tích định lượng tùy chọn. Chúng cho phép các tùy chọn khác nhau của chi phí được tính toán và, thậm chí với lỗi, các tùy chọn có thể được so sánh dựa trên mục đích.
- Thời gian cần để hoàn thành một dự án không đơn giản chỉ là tỉ lệ với số người đang làm việc cho dự án. Việc thêm người vào dự án muộn có thể tăng thay vì giảm thời gian cần để kết thúc dự án.

#### **Đọc thêm:**

‘Ten unmyths of project estimation’ Một bài báo thực dụng thảo luận các khó khăn thực tế của ước lượng dự án và thách thức một vài giả thiết cơ bản trong lĩnh vực này. (P. Armour, Comm. ACM, 45(11), November)

*Software Cost Estimation with COCOMO II.* Đây là một cuốn sách rõ ràng về mô hình COCOMO II. Nó cung cấp mô tả đầy đủ về mô hình với nhiều ví dụ và bao gồm các phần mềm thực hiện mô hình. Nó thực sự chi tiết và không dễ đọc. Bài báo của Boehm ở dưới, theo quan điểm của tôi, là một sự giới thiệu dễ hơn. (B.Boehm, et al, 2000, Prentice Hall)

*Software Project Management: Reading and Cases.* Một bộ sưu tập các bài báo và trường hợp nghiên cứu về quản lý dự án mà nổi riêng là mạnh mẽ trong mô hình hóa chi phí giải thuật. (C.F.Kemerer (ed.) 1997, Irwin)

‘Cost models for future software life cycle processes : COCOMO II’. Giới thiệu mô hình ước lượng chi phí COCOMO II , bao gồm cơ sở cho các công thức được sử dụng. Để đọc hơn quyển sách cuối cùng.( B. Boehm et al. Annals of software Engineering, 1 Balzer Science Publisher, 1995)

### **Bài tập.**

3. Dưới hoàn cảnh nào một công ty có thể đòi giá cao hơn cho hệ thống phần mềm so với giá tính bởi ước lượng chi phí cộng thêm lãi suất bình thường.
4. Mô tả hai độ đo mà được sử dụng để đo năng suất lập trình viên. Bình luận ngắn gọn về thuận lợi và bất lợi của từng độ đo.
5. Trong phát sự phát triển của các hệ thống nhúng lớn, đề suất năm nhân tố mà có thể có tác động lớn đến năng suất của đội phát triển phần mềm.
6. Ước lượng chi phí vốn có sẵn rủi ro bất kể kĩ thuật ước lượng nào được sử dụng. Đề suất bốn cách làm giảm rủi ro trong ước lượng chi phí.
7. Tại sao nên sử dụng vài kĩ thuật ước lượng để ước lượng chi phí của hệ thống phần mềm lớn, phức tạp.
8. Đưa ra 3 nguyên nhân tại sao các ước lượng chi phí giải thuật được chuẩn bị trong các tổ chức khác nhau không thể so sánh trực tiếp.
9. Giải thích làm thế nào cách tiếp cận thuật toán với ước lượng chi phí có thể được sử dụng bởi những người quản lý dự án để phân tích tùy chọn. Đề suất một tình huống mà người quản lý có thể chọn cách tiếp cận mà không dựa trên chi phí dự án thấp nhất.
10. Một số dự án phần mềm rất lớn bao gồm việc viết hàng triệu dòng mã. Để suất các mô hình ước lượng chi phí hữu dụng cho các dự án như thế như thế nào. Tại sao các giả thiết chúng dựa trên có thể không đúng với các hệ thống phần mềm rất lớn.
11. Có hợp lý không việc công ty đưa ra giá thấp cho hợp đồng phần mềm biết rằng các yêu cầu là nhập nhằng và họ có thể tính giá cao cho những thay đổi về sau được đề nghị bởi khách hàng.

Những người quản lý có nên sử dụng năng suất được đo trong suốt tiến trình đánh giá nhân viên không? Những yếu tố an toàn nào cần thiết để đảm bảo chất lượng không bị ảnh hưởng bởi điều này ?



## CHƯƠNG 9: QUẢN LÝ CHẤT LƯỢNG PHẦN MỀM

*Mục tiêu:*

*Các mục đích của chương này là giới thiệu quản lý chất lượng phần mềm và độ đo phần mềm. Trong chương này bạn sẽ:*

- *Tìm hiểu về quá trình quản lý chất lượng và các hoạt động quá trình trung tâm của sự đảm bảo của chất lượng, lập kế hoạch chất lượng và kiểm soát chất lượng.*
- *Tìm hiểu sự quan trọng của các chuẩn mực trong quá trình quản lý chất lượng.*
- *Hiểu độ đo phần mềm là gì và sự khác biệt giữa độ đo tiên nghiệm và độ đo điều khiển.*
- *Tìm hiểu cách đo có thể hữu ích trong việc đánh giá các thuộc tính chất lượng sản phẩm.*
- *Có hiểu biết về các giới hạn hiện tại về độ đo phần mềm.*

### 9.1. Chất lượng quá trình và chất lượng sản phẩm:

Chất lượng của phần mềm phát triển mạnh mẽ từ 15 năm về trước. Một trong các lý do cho sự phát triển này là do các công ty đã áp dụng các kỹ thuật và công nghệ mới, ví dụ như việc sử dụng các phát triển hướng đối tượng và sự kết hợp cung cấp công nghệ phần mềm với sự hỗ trợ của máy tính. Thêm vào đó thì có những kiến thức nhiều hơn về tầm quan trọng của việc quản lý chất lượng sản phẩm và việc áp dụng các kỹ thuật quản lý chất lượng từ các nhà sản xuất trong công nghiệp phần mềm.

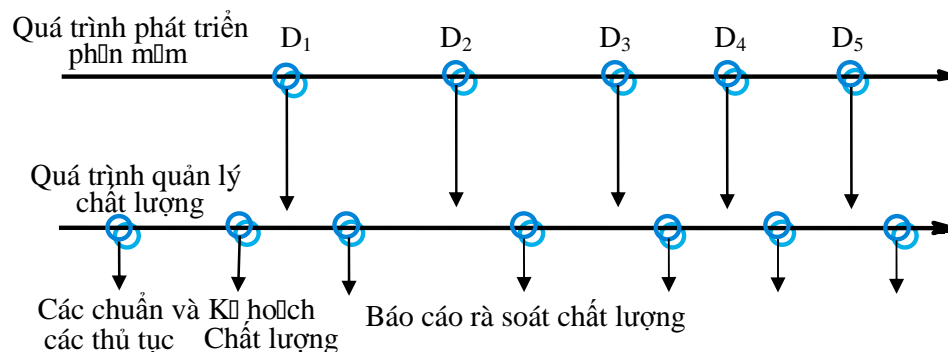
Tuy nhiên, chất lượng phần mềm là một khái niệm phức tạp, nó không thể so sánh một cách trực tiếp với chất lượng trong sản xuất. Trong sản xuất, khái niệm của chất lượng được đưa ra là: sản phẩm phát triển phải phù hợp với đặc tả của nó (Crosby, 1979). Nhìn chung, định nghĩa này được áp dụng cho tất cả các sản phẩm, tuy nhiên đối với hệ thống phần mềm, nảy sinh một số vấn đề với định nghĩa này:

1. Đặc tả phải được định hướng tới các đặc trưng của sản phẩm mà khách hàng mong muốn. Tuy nhiên, tổ chức phát triển có thể cũng có các yêu cầu (như các yêu cầu về tính bảo trì) mà không được kể đến trong các đặc tả.
2. Chúng ta không biết làm cách nào định rõ các đặc trưng chất lượng (ví dụ như tính bảo trì được) một cách rõ ràng.
3. Như ta đã nói ở phần I, nó bao gồm các yêu cầu về kỹ thuật, việc viết được đầy đủ các đặc tả phần mềm là một công việc rất khó khăn. Vì vậy, mặc dù sản phẩm phần mềm có thể phù hợp với các đặc tả của nó, nhưng người sử dụng có thể không coi đó là sản phẩm chất lượng cao bởi vì nó không phù hợp với những mong đợi của họ.

Bạn phải nhận ra những vấn đề do sự tồn tại các thông số phần mềm, vì vậy việc tạo ra các thiết kế chất lượng không chỉ phụ thuộc vào việc có một đặc tả hoàn hảo. Nói riêng, các thuộc tính phần mềm như khả năng bảo trì được, tính bảo mật hay tính hiệu quả không thể được định rõ. Tuy nhiên, chúng có tác động to lớn đến chất lượng của hệ thống. Ta sẽ bàn đến các thuộc tính này ở trong phần 9.3.

Một số người nghĩ rằng chất lượng có thể đạt được bằng cách định nghĩa các chuẩn, và các thủ tục chất lượng có tính tổ chức để kiểm tra những chuẩn này có được tuân theo bởi các đội phát triển phần mềm. Những tranh cãi của họ là việc các chuẩn được tóm lược thói quen tốt hay không, việc tuân theo các thói quen đó chắc chắn dẫn dắt đến các sản phẩm chất lượng cao. Trong thực tế, tuy nhiên, tôi nghĩ rằng cần nhiều sự quản lý chất lượng hơn là các chuẩn và được kết hợp với công việc phức tạp để đảm bảo rằng những chuẩn này được tuân theo.

Các nhà quản lý chất lượng tốt có mục tiêu là phát triển một “văn hoá chất lượng” nơi mà trách nhiệm của mọi cam kết cho sự phát triển sản phẩm để đạt tới mức độ cao của chất lượng sản phẩm. Họ khuyến khích các nhóm chịu trách nhiệm cho chất lượng công việc của mình và để phát triển các cách tiếp cận để cải tiến chất lượng. Khi mà các chuẩn và các thủ tục là phần cơ bản của quản lý chất lượng, kinh nghiệm của người quản lý chất lượng cho thấy có những mong đợi không thể nhìn thấy được về chất lượng phần mềm (tính bất mắt, tính dễ đọc, ...) mà không thể được thể hiện một cách rõ ràng theo các chuẩn. Chúng phục vụ cho những người quan tâm đến những khía cạnh không nhìn thấy được của chất lượng và khuyến khích cách cư xử chuyên nghiệp trong tất cả các thành viên của nhóm.



Hình 9.2: Quản lý chất lượng và quản lý phần mềm

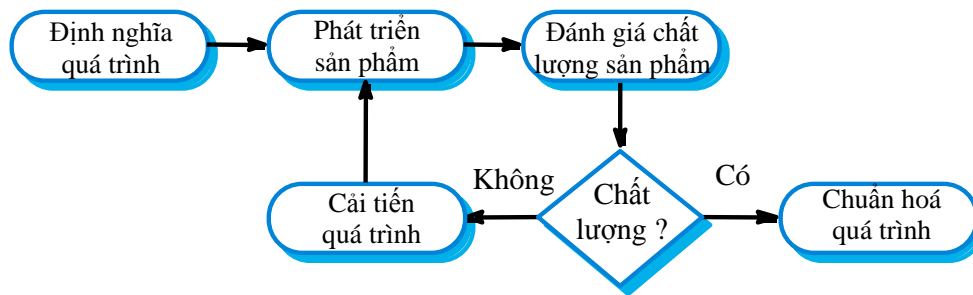
Việc quản lý chất lượng đã được chính thức hoá nói riêng là rất quan trọng đối với các nhóm có nhiệm vụ phát triển các hệ thống lớn và phức tạp. Tài liệu về chất lượng là một bản ghi về những việc đã làm bởi mỗi nhóm nhỏ trong một dự án. Nó trợ giúp con người kiểm tra những nhiệm vụ quan trọng không được phép quên, hay một phần của nhóm tạo ra các giả định về những gì mà các nhóm khác đã làm. Tài liệu ghi chất lượng cũng có nghĩa về việc trao đổi trong khoảng thời gian tồn tại của hệ thống. Nó cho phép các nhóm chịu trách nhiệm cho sự phát triển của hệ thống thể hiện những gì mà nhóm phát triển đã thực hiện.

Đối với những hệ thống nhỏ, quản lý chất lượng vẫn rất quan trọng, nhưng với một cách tiếp cận đơn giản hơn được áp dụng. Không cần thiết nhiều công việc giấy tờ bởi vì một nhóm phát triển nhỏ có thể trao đổi trực tiếp. Vấn đề then chốt chất lượng cho sự phát triển các hệ thống nhỏ là việc thiết lập một “văn hoá chất lượng” và bảo đảm rằng tất cả các thành viên nhóm có tiếp cận một cách tích cực với chất lượng phần mềm.

Quản lý chất lượng phần mềm cho các hệ thống lớn có thể được chia vào 3 hoạt động chính.

1. **Sự đảm bảo chất lượng:** sự thiết lập của một khung của tổ chức các thủ tục và các chuẩn để hướng đến sản phẩm chất lượng cao.
2. **Lập kế hoạch chất lượng:** Việc chọn lựa các thủ tục và các chuẩn thích hợp từ khung này, được sửa chữa cho các dự án phần mềm riêng biệt.
3. **Kiểm soát chất lượng:** Định nghĩa và đưa ra các quá trình để đảm bảo rằng đội phát triển phần mềm phải tuân theo các thủ tục và các chuẩn chất lượng dự án.

Quản lý chất lượng cung cấp một kiểm tra độc lập trong quá trình phát triển phần mềm. Quá trình quản lý chất lượng kiểm tra mức độ thực hiện dự án để đảm bảo rằng chúng phù hợp với các chuẩn và mục tiêu của tổ chức. (Hình 9.1). Đội đảm bảo chất lượng độc lập với đội phát triển, vì vậy họ có thể tạo ra các cách nhìn nhận khác nhau về phần mềm. Họ báo cáo các vấn đề và khó khăn tới người quản lý có thẩm quyền trong tổ chức.



Hình 9.2 Chất lượng được dựa trên quá trình.

Một đội độc lập chịu trách nhiệm đối với việc quản lý chất lượng và sẽ báo cáo tới người quản lý dự án ở cấp cao hơn. Đội quản lý chất lượng không được liên kết với bất cứ nhóm phát triển riêng biệt nào nhưng nhận trách nhiệm cho việc quản lý chất lượng. Lý do cho việc này là người quản lý dự án phải duy trì ngân sách dự án và lập lịch dự án. Nếu vấn đề xuất hiện, họ có thể bị lôi cuốn vào việc thỏa hiệp chất lượng sản phẩm, vì vậy họ phải lập lịch. Một đội quản lý chất lượng độc lập bảo đảm rằng mục tiêu tổ chức của chất lượng không bị thỏa hiệp bởi ngân sách ngắn hạn và chi phí lập lịch.

## 9.2. Chất lượng quá trình và chất lượng sản phẩm:

Một giả định cơ bản của quản lý chất lượng là chất lượng của quá trình phát triển ảnh hưởng trực tiếp đến chất lượng của các sản phẩm. Sự giả định này do hệ thống các nhà sản xuất, nơi mà chất lượng sản phẩm liên hệ một cách mật thiết tới quá trình sản xuất. Trong hệ thống sản xuất tự động, quá trình này bao gồm , cấu hình, cài đặt, và thao tác máy phát triển trong quá trình. Một khi các máy thao tác một cách chính xác, chất lượng sản phẩm tự nhiên sẽ tuân theo. Bạn có thể đo chất lượng của sản phẩm và thay đổi quá trình cho đến khi bạn đạt được mức độ chất lượng như mong muốn. Hình 9.2 cung cấp cách tiếp cận quá trình cơ bản để đạt được chất lượng sản phẩm.

Có sự liên kết rõ ràng giữa chất lượng của quá trình và chất lượng của sản phẩm trong sản xuất, bởi vì quá trình chuẩn hoá và giám sát tương đối dễ. Một khi hệ thống sản xuất được kiểm tra, chúng có thể được chạy lại nhiều lần cho đến ra là các sản phẩm có chất lượng cao. Tuy nhiên, phần mềm không phải là sản xuất, nhưng được thiết kế. Phát triển phần mềm là một sự sáng tạo hơn là một quá trình máy móc, vì vậy sự ảnh hưởng của các kỹ năng và kinh nghiệm riêng là rất đáng kể. Nhân tố bên ngoài, như tính mới lạ của ứng dụng hay sức ép của thương mại cho một sản phẩm mới, cũng ảnh hưởng đến chất lượng sản phẩm bất chấp quá trình được sử dụng.

Trong sự phát triển phần mềm, mối quan hệ giữa chất lượng quá trình và chất lượng sản phẩm là phức tạp hơn. Việc đo các thuộc tính chất lượng phần mềm là rất khó khăn, như khả năng bảo trì được, ngay cả sau khi sử dụng phần mềm trong một thời gian dài. Do đó, rất khó để nói xem các đặc trưng quá trình ảnh hưởng như thế nào đến các thuộc tính. Hơn thế nữa, bởi vì các quy định của thiết kế và sáng tạo trong quá trình phần mềm, ta không thể dự đoán quá trình thay đổi như thế nào sẽ ảnh hưởng đến chất của phẩm. Tuy nhiên, nhiều chuyên gia chỉ ra rằng quá trình chất lượng có ảnh hưởng đáng kể đến chất lượng của phần mềm. Quản lý và phát triển chất lượng quá trình và việc cải tiến thể tất yếu dẫn đến có ít khiếm khuyết hơn trong sản phẩm phần mềm được phát hành.

#### Quản lý quá trình chất lượng bao gồm:

1. Định nghĩa các chuẩn quá trình như bằng cách nào và khi nào những rà soát được chỉ đạo.
2. Giám sát quá trình phát triển để đảm bảo rằng các chuẩn được tuân theo.
3. Báo cáo quá trình phần mềm đến quản lý dự án và người mua phần mềm.

Một vấn đề với sự đảm bảo chất lượng dựa trên quá trình đo là đội đảm bảo chất lượng (QA- Quality Assurance) có thể bắt buộc yêu cầu quá trình chuẩn phải được sử dụng bất chấp kiểu loại phần mềm đang được phát triển. Ví dụ, các chuẩn của chất lượng quá trình cho hệ thống quan trọng có thể chỉ ra rằng các đặc tả phải được hoàn thành và được thông qua trước khi việc thực thi bắt đầu. Tuy nhiên đối với một số hệ thống quan trọng có thể yêu cầu nguyên mẫu, khi đó các chương trình được thực thi mà không cần đặc tả đầy đủ. Tôi đã có kinh nghiệm trong những trường hợp này khi mà đội quản lý chất lượng khuyên rằng mẫu này không được tiến hành bởi vì chất lượng bản mẫu không thể kiểm soát được. Trong một số trường hợp, người quản lý cấp cao phải can thiệp để bảo đảm rằng quá trình chất lượng trợ giúp nhiều hơn là gây cản trở cho sự phát triển sản phẩm.

### 9.3. Đảm bảo chất lượng và các chuẩn chất lượng.

Đảm bảo chất lượng là quá trình của việc định rõ làm cách nào để chất lượng sản phẩm có thể đạt được và làm thế nào để cho tổ chức phát triển biết phần mềm có yêu cầu chất lượng ở cấp độ nào. Đảm bảo chất lượng tiên tiến có liên quan đầu tiên đến việc định ra hoặc chọn lựa các chuẩn sẽ được áp dụng cho quá trình phát triển phần mềm hay sản phẩm phần mềm. Như là một phần của quá trình đảm bảo chất lượng, bạn có thể chọn lựa hoặc tạo ra các công cụ và các phương pháp để phục vụ cho các chuẩn này.

Có 2 loại chuẩn có thể được áp dụng như là một phần của quá trình đảm bảo chất lượng là:

1. *Các chuẩn sản phẩm*: Những chuẩn này áp dụng cho sản phẩm phần mềm phát triển. Chúng bao gồm các định nghĩa của đặc tả, như là cấu trúc của tài liệu yêu cầu; các chuẩn tài liệu, như các tiêu đề giải thích chuẩn cho định nghĩa lớp đối tượng; và các chuẩn mã để định rõ làm cách nào ngôn ngữ lập trình có thể được sử dụng.
2. *Các chuẩn quá trình*: Những chuẩn này định ra quá trình nên được tuân theo trong quá trình phát triển phần mềm. Chúng có thể bao gồm các việc xác định các đặc tả. Quá trình thiết kế và kiểm định quá trình và một bản mô tả các tài liệu nên được ghi lại trong giai đoạn của những quá trình này.

Như tôi đã nói ở mục 27.1, có một sự liên hệ rất gần giữa các chuẩn sản phẩm và chuẩn quá trình. Các chuẩn sản phẩm áp dụng cho đầu ra của quá trình phần mềm và trong nhiều trường hợp, các chuẩn quá trình bao gồm các hoạt động quá trình riêng biệt mà đảm bảo rằng các chuẩn sản phẩm được tuân theo.

Các chuẩn phần mềm là rất quan trọng vì những lý do sau:

1. Các chuẩn phần mềm dựa trên hiểu biết về những thực tiễn thích hợp nhất cho công ty. Kinh nghiệm này thường chỉ đạt được sau rất nhiều lần thử nghiệm và lỗi. Bỏ xung nó vào các chuẩn giúp cho công ty tránh sự lặp lại sai lầm trong quá khứ. Các chuẩn chứa đựng các kinh nghiệm từng trải này rất có giá trị cho tổ chức.
2. Các chuẩn phần mềm cung cấp một cái khung cho việc thực thi quá trình đảm bảo chất lượng. Đưa ra các chuẩn tổng kết thực tiễn, đảm bảo chất lượng bao gồm việc bảo đảm rằng các chuẩn được tuân theo một cách chặt chẽ.
3. Các chuẩn phần mềm trợ giúp tính liên tục khi mà một người tiếp tục công việc của người khác đã bỏ dở. Các chuẩn đảm bảo rằng tất các kỹ sư trong tổ chức chấp nhận cùng thói quen. Do vậy công sức nghiên cứu khi bắt đầu công việc mới sẽ giảm xuống.

Sự phát triển của các chuẩn dự án kỹ thuật phần mềm là quá trình rất khó khăn và tốn thời gian. Các tổ chức quốc gia, quốc tế như US DoD, ANSI, BSI, NATO và IEEE chủ động tạo ra các chuẩn. Những chuẩn này là chuẩn chung mà có thể được áp dụng ở phạm vi của các dự án. Các tổ chức như NATO và các các tổ chức bảo vệ có thể yêu cầu các chuẩn của họ được tuân theo trong các hợp đồng phần mềm.

Các chuẩn quốc gia và quốc tế đã phát triển bao gồm cả công nghệ kỹ thuật phần mềm, ngôn ngữ lập trình như Java, và C++, các ký hiệu như là biểu tượng bản đồ, các thủ tục cho các yêu cầu nhận và viết phần mềm, các thủ tục đảm bảo chất lượng, kiểm tra phần mềm và quá trình thông qua (IEEE, 2003).

Các nhóm đảm bảo chất lượng mà đang phát triển các chuẩn cho công ty thường dựa trên chuẩn quốc gia và quốc tế. Sử dụng những chuẩn này như là điểm bắt đầu, nhóm đảm bảo chất lượng phải thảo ra một tài liệu tóm tắt chuẩn. Tài liệu này phải định ra những tiêu chuẩn được yêu cầu bởi tổ chức của họ. Ví dụ về những tiêu chuẩn mà có thể kể đến trong tài liệu sách tóm tắt trong bảng 27.3.

Các chuẩn sản phẩm	Các chuẩn quá trình
Mẫu rà soát thiết kế	Sắp đặt rà soát thiết kế
Cấu trúc tư liệu yêu cầu	Sự đệ trình tư liệu đến CM (???)
Phương pháp định dạng tiêu đề	Quá trình phát hành phiên bản
Kiểu lập trình Java	Quá trình thông qua kế hoạch dự án
Định dạng kế hoạch dự án	Quá trình kiểm soát thay đổi
Mẫu yêu cầu thay đổi	Quá trình ghi nhận kiểm tra.

Bảng 9.3: Các chuẩn quá trình và chuẩn sản phẩm.

Các kỹ sư phần mềm đôi khi coi các chuẩn là phức tạp và không thích hợp đối với hoạt động công nghệ của việc phát triển phần mềm. Các chuẩn yêu cầu choán đầy các mẫu dài dòng và phải ghi lại công việc. Mặc dù các kỹ sư phần mềm thường đồng ý về các yêu cầu chung cho các tiêu chuẩn, các kỹ sư thường tìm nhiều lý do tại sao các chuẩn không thực sự thích hợp với dự án riêng của họ. Để tránh những vấn đề này, những người quản lý chất lượng thiết lập những tiêu chuẩn cần thiết là những tài nguyên tương xứng, và nên tuân theo các bước sau:

1. Bao gồm các kỹ thuật phần mềm trong việc chọn lựa các chuẩn sản phẩm. Họ nên hiểu tại sao các tiêu chuẩn được thiết kế và cam kết tuân theo chuẩn này. Tài liệu chuẩn không chỉ là đơn giản là nói rõ chuẩn được tuân theo mà nó phải bao gồm lý do căn bản tại sao các tiêu chuẩn riêng biệt được chọn.
2. Kiểm tra và thay đổi các tiêu chuẩn một cách đều nhau phản ánh các công nghệ thay đổi. Một khi các tiêu chuẩn được phát triển, chúng có xu hướng được lưu trữ trong tài liệu tóm tắt các tiêu chuẩn của công ty, và việc quản lý thường khó có thể thay đổi chúng. Một tài liệu tóm tắt tiêu chuẩn là rất cần thiết nhưng nó nên mở ra việc phản ánh các tình huống thay đổi và công nghệ thay đổi.
3. Cung cấp các công nghệ phần mềm để phục vụ các tiêu chuẩn bất kể khi nào có thể. Các tiêu chuẩn văn phòng là nguyên nhân của nhiều than phiền bởi vì công việc quá dài dòng để thực hiện chúng. Nếu công cụ phục vụ là có hiệu lực, bạn không cần cố gắng thêm để tuân theo các chuẩn phát triển phần mềm.

Các chuẩn quá trình có thể gây ra nhiều khó khăn nếu một quá trình không có tính thực tế được áp đặt cho nhóm phát triển. Các kiểu khác nhau của phần mềm cần các quá trình phát triển khác nhau. Không nhất thiết phải quy định cách làm việc nếu nó không thích hợp cho một dự án hay đội dự án. Mỗi người quản lý dự án phải có quyền thay đổi các chuẩn quá trình theo những trường hợp riêng. Tuy nhiên, các chuẩn mà liên quan đến chất lượng sản phẩm và quá trình gửi- phát phải chỉ được thay đổi sau khi có sự cân nhắc cẩn thận.



Người quản lý dự án và người quản lý chất lượng có thể tránh nhiều vấn đề về các chuẩn không hợp lý bằng cách lập kế hoạch chất lượng chu đáo sớm trong dự án. Họ phải quyết định những chuẩn đưa vào trong tài liệu, mà những chuẩn không thay đổi được ghi vào tài liệu, còn những chuẩn nào có thể được chỉnh sửa và những chuẩn nào có thể được bỏ qua. Những chuẩn mới có thể phải được tạo ra để đáp ứng những yêu cầu riêng biệt của từng dự án. Ví dụ, tiêu chuẩn cho các đặc tả hình thức có thể được yêu cầu nếu những đặc tả này không được sử dụng trong các dự án trước. Khi mà đội có thêm kinh nghiệm với chúng, bạn nên lập kế hoạch chỉnh sửa và đưa ra những chuẩn mới.

Trách nhiệm quản lý	Hệ thống chất lượng
Kiểm soát các sản phẩm không quy tắc	Kiểm soát thiết kế
Giải quyết, lưu trữ, đóng gói và phân phát	(?? Sức mua)
Các sản phẩm người mua sắm đầu vào	Dấu vết và căn cước của sản phẩm
Kiểm soát quá trình	Kiểm tra và thử nghiệm
Trang bị kiểm tra và thử nghiệm	Trạng thái kiểm tra và thử nghiệm
Rà soát lại hợp đồng	Hoạt động chỉnh sửa
Kiểm soát tư liệu	Bản ghi chất lượng.
Kiểm toán chất lượng nội bộ	Đào tạo
Dịch vụ	Kỹ thuật thống kê

### 9.3.1. ISO 9000

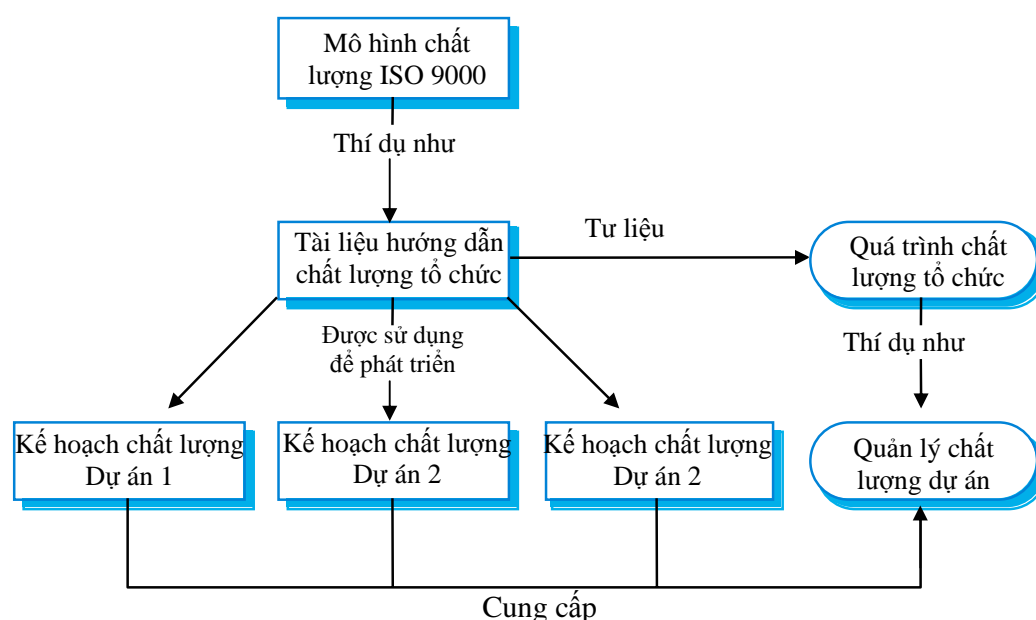
Một tập hợp các tiêu chuẩn quốc tế mà có thể được sử dụng trong việc phát triển của hệ thống quản lý chất lượng trong tất cả ngành công nghiệp được gọi là ISO 9000. Các chuẩn ISO 9000 có thể được áp dụng cho nhiều tổ chức từ sản xuất cho đến công nghiệp dịch vụ. ISO 9001 là những cái chung nhất của những chuẩn này và áp dụng cho những tổ chức trong các quá trình chất lượng dùng để thiết kế, phát triển và bảo trì sản phẩm. Một tài liệu phục vụ (ISO 9000-3) hiểu là ISO 9001 cho phát triển phần mềm. Một vài quyển sách mô tả chuẩn ISO 9000 là có giá trị (Johnson, 1993; Oskarsson và Glass, 1995; Peach, 1996; Bamford và Deibler, 2003).

Chuẩn ISO không tập trung cụ thể vào việc phát triển phần mềm nhưng nó thiết lập nguyên lý chung mà có thể ứng dụng vào phần mềm. Chuẩn ISO 9000 mô tả nhiều dạng bên ngoài khác nhau của quá trình chất lượng và bố cục các chuẩn tổ chức và các thủ tục tổ chức mà một công ty phải định ra. Những chuẩn này được ghi lại trong một tài liệu chỉ dẫn chất lượng của tổ chức. Định nghĩa quá trình phải bao gồm việc mô tả các tài liệu được yêu cầu để thể hiện những quá trình được định ra được tuân theo trong quá trình phát triển sản phẩm.

Chuẩn ISO 9001 không định nghĩa quá trình chất lượng nào nên được sử dụng. Trên thực tế, nó không ràng buộc các quá trình sử dụng vào trong bất kỳ tổ chức theo bất

kỳ cách nào. Điều nào cho phép sự mềm dẻo trong các bộ phận công nghiệp và điều này có ý nghĩa rằng các công ty nhỏ có thể có những quá trình không phức tạp và vẫn tuân theo chuẩn ISO 9000. Tuy nhiên, sự linh hoạt này có nghĩa là bạn không thể tạo ra bất kỳ giả định nào về sự tương tự hay khác nhau giữa quá trình trong các công ty ứng dụng ISO 9000.

Bảng 9.4 thể hiện các lĩnh vực bao trùm trong ISO 9001. Tôi không đủ không gian ở đây để thảo luận chuẩn này sâu hơn nữa được. Ince (Ince, 1994) và Oskarrson và Glass (Oskarrson and Glass, 1995) đưa bản mô tả chi tiết hơn về việc làm thế nào chuẩn có thể được sử dụng để phát triển các quá trình quản lý chất lượng phần mềm. Các mối liên quan giữa ISO 9000, tài liệu chỉ dẫn về chất lượng và các kế hoạch chất lượng dự án riêng biệt được thể hiện trong hình 27.5.1. Tôi đã lấy hình vẽ này từ một mô hình được đưa ra trong quyển sách của Ince (Ince, 1994).



Hình 9.5: ISO 9000 và quản lý chất lượng

Các thủ tục đảm bảo chất lượng trong tổ chức được tài liệu hoá trong tài liệu chỉ dẫn chất lượng, tài liệu này định ra quá trình chất lượng. Trong một số quốc gia, giấy chứng nhận quyền sở hữu chứng nhận rằng quá trình chất lượng tuân theo chuẩn ISO 9001. Ngày càng nhiều khách hàng xem giấy chứng nhận ISO 9000 ở nhà phục vụ như là cách chứng minh tính đúng đắn chất lượng sản phẩm của nhà cung cấp.

Một số người nghĩ rằng giấy chứng nhận ISO 9000 có nghĩa rằng chất lượng của phần mềm được tạo ra bởi các công ty đã được chứng thực sẽ tốt hơn là các công ty mà chưa được chứng thực. Điều này không phải thực sự như vậy. Chuẩn ISO 9000 thường là liên quan đến định nghĩa của các quá trình được sử dụng trong công ty và tài liệu được liên kết lại như các quá trình kiểm soát mà có thể thể hiện một cách dễ dàng những quá trình được tuân theo. Nó không liên quan đến việc đảm bảo rằng những quá trình này phản ánh thực tế tốt nhất, hay chất lượng sản phẩm.



Vì vậy, một công ty có thể định ra các thủ tục kiểm tra sản phẩm mà dẫn đến việc kiểm tra phần mềm chưa hoàn thành. Trong một thời gian dài những thủ tục này được tuân theo và được tài liệu hoá, công ty nên tuân theo chuẩn ISO 9001. Trong khi, trường hợp này là chưa chắc chắn, có một số chuẩn công ty là không thuyết phục và đóng góp ít vào chất lượng phần mềm thực.

### 9.3.2. Các chuẩn tài liệu:

Các chuẩn tài liệu trong một dự án phần mềm là quan trọng bởi vì các tài liệu là cách xác thực để thể hiện phần mềm và quá trình phần mềm. Các tài liệu tiêu chuẩn hoá có bề ngoài, cấu trúc và chất lượng không thay đổi, bởi vậy nó dễ đọc và dễ hiểu hơn.

Có ba kiểu chuẩn tài liệu:

1. *Các chuẩn quá trình tài liệu* Những chuẩn này định ra quá trình mà sẽ được tuân theo đối với việc tạo ra tài liệu.
2. *Các chuẩn tài liệu* Những chuẩn này chi phối cấu trúc và cách thể hiện của các tài liệu.
3. *Các chuẩn trao đổi tài liệu* Những chuẩn này đảm bảo rằng tất cả các bản sao điện tử của các tài liệu là tương thích.

Các chuẩn tài liệu quá trình định ra quá trình mà được sử dụng cho việc tạo ra các tài liệu. Điều này có nghĩa rằng bạn sắp đặt các thủ tục, bao gồm việc phát triển tài liệu và các thiết bị phần mềm được sử dụng cho việc tạo ra tài liệu. Bạn cũng có thể định ra việc kiểm tra và cải tiến các thủ tục để đảm bảo rằng các tài liệu có chất lượng cao được tạo ra.

Các chuẩn tài liệu chất lượng quá trình phải linh hoạt và có thể thích ứng với nhiều loại tài liệu. Đối với công việc trên giấy tờ hay sổ ghi nhớ điện tử, không cần thiết phải kiểm tra chất lượng một cách rõ ràng. Tuy nhiên, đối với các tài liệu chính thức sẽ được sử dụng cho việc phát triển sau này hay chuyển giao cho khách hàng, thì bạn nên sử dụng quá trình chất lượng chính qui. Hình 27.6 là một mô hình của quá trình tài liệu có thể thực hiện được.

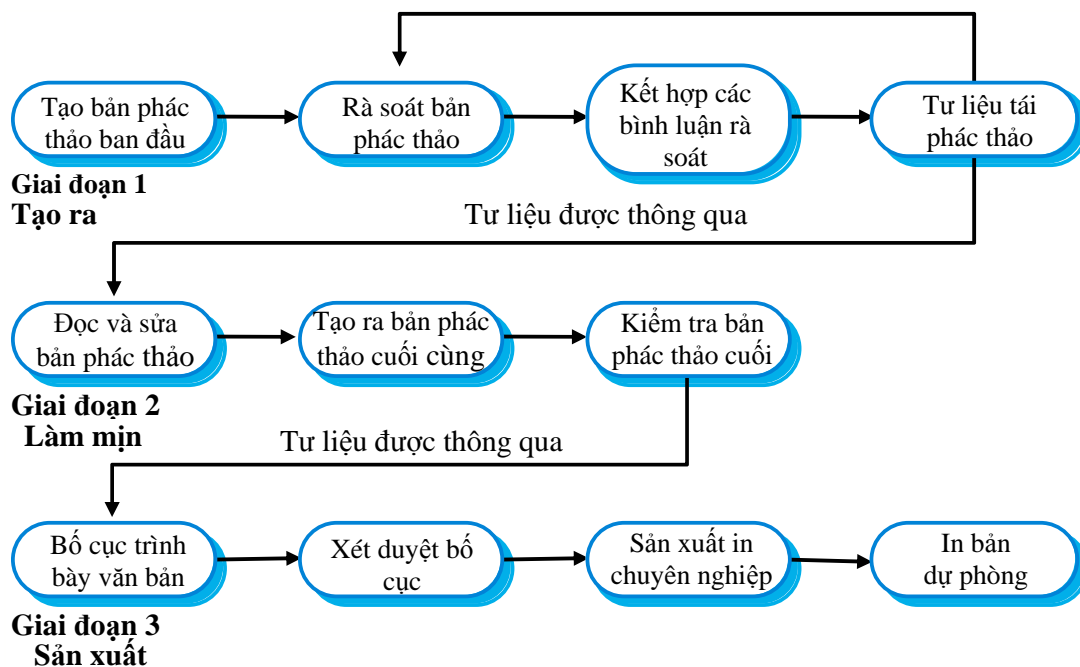
Việc phác thảo, kiểm tra, phác thảo lại và biên tập lại là một quá trình lặp đi lặp lại. Nó nên tiếp diễn cho đến khi một tài liệu chất lượng có thể chấp nhận được được tạo ra. Mức độ chất lượng yêu cầu còn tùy thuộc vào kiểu của tài liệu và khả năng của người đọc tài liệu.

Các chuẩn tài liệu phải áp dụng cho tất cả các tài liệu được tạo ra trong một dự án phát triển phần mềm. Các tài liệu phải có một kiểu cách và bề ngoài ổn định. Các tài liệu của cùng một kiểu cũng phải có cấu trúc như nhau. Mặc dù các chuẩn tài liệu có thể thích ứng với các yêu cầu của dự án riêng biệt, trong thực tế đó là quy luật như nhau được sử dụng trong tất cả các tài liệu được sản xuất bởi tổ chức.

Các ví dụ của các chuẩn tài liệu có thể được phát triển là:

1. *Các chuẩn nhận dạng tài liệu* Khi mà những dự án phát triển hệ thống lớn có thể tạo ra hàng nghìn các tài liệu, mỗi tài liệu phải được nhận biết duy nhất. Đối với các tài liệu chính quy, nhận dạng này có thể là nhận dạng chính quy được định ra bởi người quản lý cấu hình. Đối với các tài liệu không chính quy, người quản lý dự án có thể định ra mẫu của tài liệu.

2. *Các chuẩn cấu trúc tài liệu* Mỗi lớp của tài liệu được tạo ra trong dự án phần mềm phải theo một số cấu trúc chuẩn. Các chuẩn cấu trúc phải định ra các mục được thêm vào và phải định rõ các quy ước được sử dụng cho đánh số trang, thông tin tiêu đề trang, và đánh số mục và mục con.
3. *Các chuẩn trình diễn tài liệu* Các chuẩn trình diễn tài liệu định ra quy luật cho các tài liệu và đóng góp đáng kể cho tính kiên định của tài liệu. Chúng bao gồm việc định rõ của các cỡ chữ và kiểu chữ được sử dụng trong tài liệu, cách sử dụng logo và các tên công ty, cách sử dụng màu sắc làm cho nổi bật cấu trúc tài liệu, ...



Hình 9.6 Một quá trình sản xuất tài liệu bao gồm việc kiểm tra chất lượng

4. *Các chuẩn cập nhật tài liệu* Khi mà một tài liệu phát triển phản ánh các thay đổi trong hệ thống, chỉ thị phù hợp cho những thay đổi tài liệu sẽ được sử dụng. Bạn có thể sử dụng màu sắc để biểu thị phiên bản của văn bản và các thanh thay đổi trong mục canh lề để biểu thị một số đoạn được thay đổi hay được thêm vào. Tuy nhiên tôi khuyên không nên sử dụng sự thay đổi tự hiệu chỉnh khi được phục vụ trong một số bộ xử lý văn bản thường được sử dụng. Nếu có nhiều tác giả, sự tự hiệu chỉnh đem lại nhiều sự khó khăn hơn là hữu dụng.

Các chuẩn trao đổi tài liệu là rất quan trọng khi mà các bản sao điện tử của các tài liệu được trao đổi. Sử dụng các chuẩn trao đổi cho phép các tài liệu được truyền đi bằng tín hiệu điện và được tái tạo lại dưới dạng nguyên gốc.

Thừa nhận rằng sử dụng các công cụ chuẩn được đặt trong các quá trình chuẩn, các chuẩn trao đổi định ra các quy ước cho việc sử dụng những công cụ này. Các ví dụ về các chuẩn trao đổi bao gồm cách sử dụng một bảng tính kiểu chuẩn nếu một bộ xử lý văn bản được sử dụng hay các giới hạn trong việc sử dụng các macro tài liệu để tránh nhiễm virus.

Các chuẩn trao đổi cũng có thể giới hạn các cỡ chữ và các kiểu văn bản được sử dụng bởi vì máy in khác nhau và khả năng hiển thị cũng khác nhau.

#### 9.4. Lập kế hoạch chất lượng.

Lập kế hoạch chất lượng là quá trình của sự phát triển một kế hoạch chất lượng cho một dự án. Kế hoạch chất lượng phải thiết lập các chất lượng phần mềm được yêu cầu và mô tả làm cách nào những chất lượng này có thể được quyết định. Bởi vậy nó định ra phần mềm chất lượng cao thực sự có ý nghĩa như thế nào. Nếu không có sự định trước này các kỹ sư có thể tạo ra các giả định khác nhau và đôi khi là xung đột với nhau về các thuộc tính sản phẩm sẽ được tối ưu hoá.

Kế hoạch chất lượng sẽ chọn những chuẩn tổ chức mà nó thích hợp với một sản phẩm riêng biệt và quá trình phát triển. Những chuẩn mới có thể phải được định nghĩa nếu dự án sử dụng các phương pháp và công cụ mới. Humphrey (Humphrey, 1989) trong cuốn sách kinh điển về quản lý phần mềm, gợi ý rằng một cấu trúc phân cấp cho kế hoạch chất lượng. Điều này bao gồm:

1. *Sự giới thiệu sản phẩm* Một mô tả về sản phẩm, mô tả định hướng thị trường dự định và các mong đợi chất lượng cho sản phẩm.
2. *Các kế hoạch sản phẩm* Kì hạn phát hành và các trách nhiệm sản phẩm cùng với các dự án cho việc phân phối và dịch vụ sản phẩm.
3. *Các mô tả quá trình* Các quá trình phát triển và dịch vụ sẽ được sử dụng cho quản lý và phát triển sản phẩm.
4. *Các mục tiêu chất lượng* Các mục tiêu và kế hoạch chất lượng cho sản phẩm bao gồm việc xác định và điều chỉnh các thuộc tính chất lượng quan trọng của sản phẩm.
5. *Rủi ro và quản lý rủi ro* Các rủi ro chính mà có thể ảnh hưởng đến chất lượng và các hoạt động sản phẩm.

Các kế hoạch chất lượng thực sự khác biệt trong chi tiết phụ thuộc vào kích thước và kiểu của hệ thống mà đang được phát triển. Tuy nhiên, khi viết các kế hoạch chất lượng, bạn nên cố gắng giữ cho chúng ngắn nhất có thể. Nếu như tài liệu quá dài, mọi người sẽ không thể đọc nó, điều này sẽ phá huỷ mục đích của việc tạo ra kế hoạch chất lượng.

Có một phạm vi rộng của các thuộc tính chất lượng phần mềm tiềm năng (Bảng 9.7) mà bạn nên xem xét trong quá trình lập kế hoạch chất lượng. Nhìn chúng ta không thể tối ưu hoá cho tất các thuộc tính đối với bất kỳ hệ thống nào. Vì vậy trong kế hoạch chất lượng, bạn phải định ra những thuộc tính chất lượng quan trọng nhất cho phần mềm đang được phát triển. Điều này có thể là có hiệu quả là quan trọng và các nhân tố khác được bỏ qua để đạt được chất lượng. Nếu bạn đã phát biểu điều này trong kế hoạch chất lượng, các kỹ sư phát triển có thể hợp tác để đạt được điều này. Kế hoạch phải bao gồm việc định rõ quá trình đánh giá chất lượng. Điều này nên là một cách chuẩn của việc đánh giá một số chất lượng, như khả năng bảo trì hay tính bền vững được hiện diện trong sản phẩm.

Tính an toàn	Tính có thể hiểu được	Tính di động
Tính bảo mật	Tính có thể kiểm tra	Tính tiện dụng
Tính tin cậy	Tính thích khi	Tính tái sử dụng
??Tính mềm dẻo	Tính mô đun	Tính hiệu quả
Tính bền vững	Tính phức tạp	Tính dễ học.

Bảng 9.7 Các thuộc tính chất lượng phần mềm.

### 9.5. Kiểm soát chất lượng

Kiểm soát chất lượng bao gồm việc kiểm tra quá trình phát triển phần mềm để đảm bảo rằng các thủ tục và các chuẩn đảm bảo chất lượng được tuân theo. Như tôi đã thảo luận ở chương trước (hình 27.1), mức độ thực hiện quá trình phần mềm được kiểm tra lại các chuẩn dự án đã được định ra trong quá trình kiểm soát chất lượng.

Có hai cách tiếp cận bổ xung cho nhau mà có thể được sử dụng để kiểm tra chất lượng của mức độ thực hiện của dự án.

1. Việc rà soát lại chất lượng nơi mà phần mềm, tài liệu của nó và các quá trình đã sử dụng để tạo ra mà phần mềm được rà soát bởi một nhóm người. Việc rà soát chịu trách nhiệm việc kiểm tra các chuẩn dự án được tuân theo và phần mềm và các tài liệu làm cho phù hợp với những chuẩn này. Sự lệch khỏi các chuẩn này đã chú ý và người quản lý dự án được cảnh báo tới chúng.
2. Đánh giá phần mềm tự động là nơi phần mềm và các tài liệu được sẽ được tạo ra xử lý bởi một số chương trình và được so sánh với các chuẩn áp dụng cho dự án phát triển riêng biệt. Đánh giá tự động này có thể bao gồm việc đo một số thuộc tính phần mềm và so sánh những độ đo với một số mức độ mong muốn. Tôi sẽ thảo luận độ đo phần mềm trong mục 27.5.

#### 9.5.1. Rà soát chất lượng

Rà soát là phương thức được sử dụng rộng rãi nhất trong việc rà soát chất lượng của một quá trình hay sản phẩm. Việc rà soát bao gồm một nhóm người kiểm tra một phần hay tất cả một quá trình phần mềm, hệ thống hay các tài liệu liên quan với các vấn đề tiềm tàng phát hiện. Các kết luận của việc rà soát được ghi lại và thông qua một cách chính thức tới tác giả hay bất kỳ người nào chịu trách nhiệm việc sửa lại những vấn đề được phát hiện.

Hình 9.8 mô tả ngắn gọn một vài loại rà soát, bao gồm các rà soát đối với quản lý chất lượng.

Kiểu rà soát	Mục đích chủ yếu
Kiểm tra thiết kế hay chương trình	Phát hiện các lỗi chi tiết trong các yêu cầu, thiết kế hay mã. Danh sách kiểm tra các lỗi có thể sẽ dẫn dắt việc rà soát.
Rà soát tiến độ.	Cung cấp thông tin cho việc quản lý tiến độ của dự án. Đây cũng vừa là quá trình và vừa là rà soát sản phẩm nó có liên quan đến chi phí, kế hoạch, lập lịch.
Rà soát chất lượng	Tiến hành các phân tích công nghệ của các thành phần sản phẩm hay tư liệu để tìm ra chỗ không tương xứng giữa đặc tả và thiết kế thành phần, mã hay tư liệu và đảm bảo rằng các chuẩn chất lượng đã được đưa ra được tuân theo.

Vấn đề của đội rà soát là để tìm ra các lỗi và các mâu thuẫn và chuyển giao chúng cho người thiết kế hay tác giả của tài liệu. Các việc rà soát được dựa trên tài liệu nhưng nó không giới hạn tới các đặc tả, các thiết kế hay mã. Các tài liệu như các mô hình quá trình, kế hoạch kiểm tra, các thủ tục quản lý cấu hình, các chuẩn quá trình và tài liệu chỉ dẫn người dùng có thể tất cả được rà soát lại.

Đội rà soát nên có hạt nhân là ba hay bốn người mà được chọn như là người rà soát chủ yếu. Một thành viên nên là người thiết kế lâu năm người mà có thể chịu trách nhiệm cho việc ra quyết định công nghệ quan trọng. Những người xét duyệt quan trọng có thể mời các thành viên dự án khác. Họ có thể không phải xét duyệt toàn bộ tài liệu. Hơn nữa, họ tập trung vào một số phần mà ảnh hưởng đến công việc của họ. Đội rà soát có thể chuyển tài liệu đã được rà soát và yêu cầu cho các lời chú giải từ hình ảnh rộng của các thành viên dự án.

Các tài liệu được rà soát phải được phân phối tốt trước khi xét duyệt để cho phép những người rà soát có thể đọc và hiểu chúng. Mặc dù sự trễ này có thể phá vỡ quá trình phát triển, việc rà soát là không hiệu quả nếu đội xét duyệt không hiểu một cách đúng đắn các tài liệu trước khi việc rà soát diễn ra.

Việc tự rà soát nên là tương đối ngắn (hầu hết là hai giờ). Tác giả của tài liệu được rà soát nên làm chủ toạ việc rà soát và một số khác ghi lại tất cả các quyết định rà soát các hành động được xảy ra. Trong suốt quá trình rà soát, người chủ toạ chịu trách nhiệm việc đảm bảo rằng tất cả các lời phê bình được ghi chép đều được xem xét. Vị chủ toạ rà soát nên ghi vào bản ghi nhận các lời phê bình và các hoạt động được đồng ý trong quá trình rà soát. Bản ghi nhận này sao đó được xem như là một phần của tài liệu dự án chính thức. Nếu mà các vấn đề thứ yếu được phát hiện, một cuộc xét duyệt sau đó có thể là không cần thiết. Người chủ toạ chịu trách nhiệm đối với việc đảm bảo rằng các thay đổi được yêu cầu được quyết định. Nếu những thay đổi chính yếu là cần thiết, một cuộc rà soát sau đó có thể được sắp xếp.

## 9.6. CMM/CMMi

### 9.6.1. CMM và CMMi là gì?

CMM và CMMi là chuẩn quản lý quy trình chất lượng của các sản phẩm phần mềm được áp dụng cho từng loại hình công ty khác nhau. Hay nói cách khác đây là các phương pháp phát triển hay sản xuất ra các sản phẩm phần mềm.

Tháng 8/ 2006, SEI (Software Engineering Institute – Viện Công Nghệ Phần Mềm Mỹ) - tổ chức phát triển mô hình CMM/CMMI đã chính thức thông báo về phiên bản mới CMMI 1.2. Như vậy là sau gần 6 năm ban hành và sử dụng thay thế cho CMM (từ tháng 12/2001), CMMI phiên bản 1.1 (CMMI 1.1) đã được chính thức thông báo với lộ trình thời gian chuyển tiếp lên phiên bản mới CMMI 1.2.

Mặc dù số lượng các công ty phần mềm tại Việt Nam đạt được CMM/CMMI đến nay vẫn chưa nhiều, nhưng với sự khởi sắc trong lĩnh vực gia công và sản xuất phần mềm vài năm trở lại đây, sự cạnh tranh cũng như yêu cầu ngày càng cao của khách hàng, đã thúc đẩy các công ty xây dựng hệ thống quản lý chất lượng theo các mô hình quốc tế.

Có những khác biệt đáng kể giữa CMMI 1.1 và CMMI 1.2, trong khuôn khổ một bài viết chúng tôi cố gắng nêu những nét cơ bản nhất, nhằm giúp bạn đọc có cái nhìn tổng quát, từ đó dễ dàng định hướng cho việc nghiên cứu chi tiết hơn về CMMI 1.2.

**CMM và CMMi là một bộ khung (framework) những chuẩn đề ra cho một tiến trình sản xuất phần mềm hiệu quả, mà nếu như các tổ chức áp dụng nó sẽ mang lại sự khả dụng về mặt chi phí, thời gian biểu, chức năng và chất lượng sản phẩm phần mềm.**

**Mô hình CMM và mô tả các nguyên tắc và các thực tiễn** nằm bên trong tính “thành thực” quá trình phần mềm và chủ ý giúp đỡ các công ty phần mềm hoàn thiện khả năng thuần thục quá trình sản xuất phần mềm, đi từ tự phát, hỗn độn tới các quá trình phần mềm thành thực, có kỷ luật.

Bằng việc thực hiện CMM các công ty thu được những lợi ích xác thực, giảm được rủi ro trong phát triển phần mềm và tăng được tính khả báo - do đó trở thành đối tác hay một nhà cung ứng hấp dẫn hơn đối với các khách hàng trên toàn thế giới. Tuy nhiên, CMM không phải không đòi hỏi chi phí. Những nguồn lực đáng kể của công ty phải được dành cho việc hướng tới các vùng tiến trình then chốt, cần thiết để lên từng bậc thang của chứng nhận CMM. CMM đưa ra một loạt các mức độ để biểu thị mức độ thành thực đã đạt được. Mức 1 ứng với mức độ thành thực thấp nhất và mức 5 ứng với mức độ thành thực cao nhất. Gần đây, SEI đã xúc tiến CMMi, một mô hình kế thừa CMM và CMMi hiện nay các công ty cũng đang bắt đầu triển khai việc sử dụng mô hình này

## 9.6.2. Cấu trúc của CMM

### 9.6.2.1. Các level của CMM

CMM bao gồm 5 levels và 18 KPAs(Key Process Area)

5 levels của CMM như sau:

- 1: Initial, 2: Repeatable, 3: Defined, 4: Managed, 5: Optimising

Nói cách khác mỗi một level đều tuân theo một chuẩn ở mức độ cao hơn. Muốn đạt được chuẩn cao hơn thì các chuẩn của các level trước phải thoả mãn. Mỗi level đều có đặc điểm chú ý quan trọng của nó cần các doanh nghiệp phải đáp ứng được

Level 1 thì không có KPAs nào cả

Level 2 : có 6 KPAs

Level 3: có 7 KPAs

**Level 4: có 2 KPAs**

**Level 5: có 3 KPAs**

**18 KPAs của CMM được đều có 5 thuộc tính(chức năng) chung trong đó có các qui định về key practice là những hướng dẫn về các thủ tục(procedure), qui tắc(polities), và hoạt động (activites)của từng KPA.**

**Đầu tiên ta có cấu trúc của một KPA với 5 điểm đặc chung(common feature) như sau:**

-----

**KPA Goals**

-----

||

||

**1 2 3 4 5**

**Trong đó để thực hiện KPA này ta cần phải thực hiện theo những qui tắc sau để bảo đảm đạt được KPA đó:**

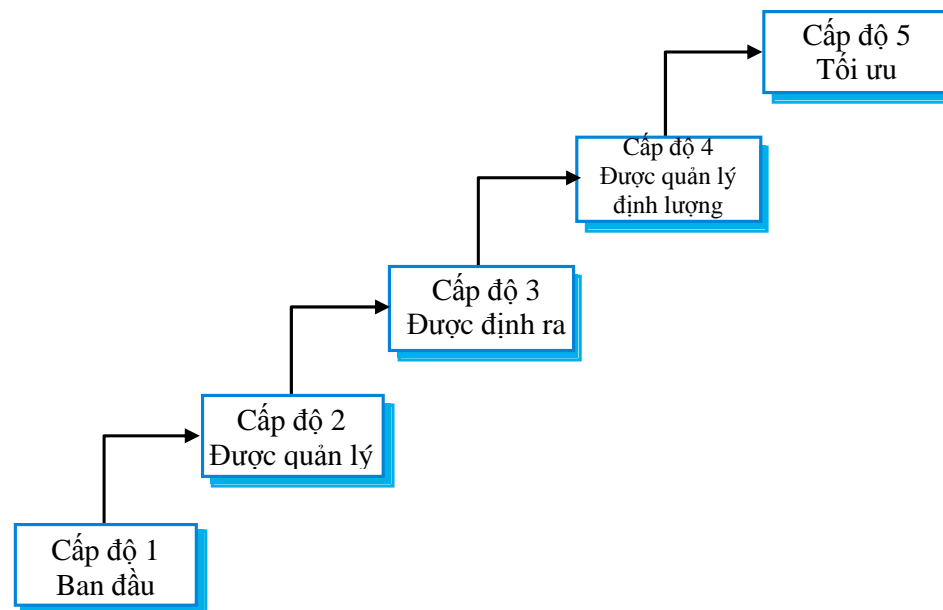
**(1) Commitment to Perform ( Tạm dịch là cam kết thực hiện)**

**(2) Ability to Perform (Khả năng thực hiện)**

**(3) Activities Peformed (Các hoạt động lâu dài)**

**(4) Measurement and Analysis (Khuôn khổ và phân tích)**

**(5) Verifying and Implementation**



### 9.6.2.2. Các level của CMM

#### ◆ Level 1

Level 1 là bước khởi đầu của CMM, mọi doanh nghiệp, công ty phần mềm, cá nhóm, cá nhân đều có thể đạt được. Ở lever này CMM chưa yêu cầu bất kỳ tính năng nào. Ví dụ: không yêu cầu quy trình, không yêu cầu con người, miễn là cá nhân, nhóm, doanh nghiệp... đều làm về phần mềm đều có thể đạt tới CMM này.

**Đặc điểm của mức 1:**

**Hành chính:** Các hoạt động của lực lượng lao động được quan tâm hàng đầu nhưng được thực hiện một cách vội vã hấp tấp

**Không thống nhất:** Đào tạo quản lý nhân lực nhỏ lẻ chủ yếu dựa vào kinh nghiệm cá nhân

**Quy trách nhiệm:** Người quản lý mong bộ phận nhân sự điều hành và kiểm soát các hoạt động của lực lượng lao động

**Quan liêu:** Các hoạt động của lực lượng lao động được đáp ứng ngay mà không cần phân tích ảnh hưởng

**Doanh số thường xuyên thay đổi:** Nhân viên không trung thành với tổ chức

#### ◆ Level 2

**Có 6 KPA nó bao gồm như sau**

- Requirement Management ( Lấy yêu cầu khách hàng, quản lý các yêu cầu đó)
- Software Project Planning ( Lập các kế hoạch cho dự án)
- Software Project Tracking (Theo dõi kiểm tra tiến độ dự án)
- Software SubContract Managent ( Quản trị hợp đồng phụ phần mềm)
- Software Quality Assurance (Đảm bảo chất lượng sản phẩm)
- Software Configuration Management (Quản trị cấu hình sản phẩm=> đúng yêu cầu của khách hàng không)

**Khi ta áp dụng Level 2, KPA 2(Software Project Planning), ta sẽ có những common feature(đặc điểm đặc trưng) như sau:**

**Mục tiêu(Goal):** các hoạt động và những đề xuất của một dự án phần mềm phải được lên kế hoạch và viết tài liệu đầy đủ

**Đề xuất/ Xem xét (Commitment):** dự án phải tuân thủ theo các qui tắc của tổ chức khi hoạch định

**Khả năng(Ability):** Việc thực hiện lập kế hoạch cho dự án phần mềm phải là bước thực hiện từ rất sớm khi dự án được bắt đầu

**Đo lường(Measument):** Sự đo lường luôn được thực thi và sử dụng chúng ta luôn có thể xác định và kiểm soát được tình trạng các hoạt động trong tiến trình thực hiện dự án



**Kiểm chứng(Verification):** Các hoạt động khi lập kế hoạch dự án phải được sự reviewed của cấp senior manager

Câu hỏi được đặt ra ở đây là vậy thì vai trò của QA sẽ như thế nào trong việc thực hiện project plan, tôi không đề cập đến vai trò Project Manager vì anh ta phải biết về các nguyên tắc về estimates project theo Function Point, Line of Code v.v...

Nói đến software quality, ắt hẳn các bạn QA không thể không biết đến tam giác quỷ "Quality Triangle" (Cost,Functionality,Schedule) => đúng chức năng , đúng chi phí, và đúng thời hạn. Để ra cái gọi là chất lượng phần mềm, thì phải có người review và kiểm chứng nó. Chúng tôi đặt ra các tiêu chuẩn của qui trình mà từ đó chúng tôi sản xuất được phần mềm tốt nhất, chúng tôi thực hiện và chúng tôi có những người kiểm chứng và theo sát các hoạt động của nhóm thực hiện project sao cho đúng qui trình. Vâng đó là QA Engineer, lâu nay vai trò của người QA trong các công ty phần mềm bị đồng hoá với vai trò của một tester hay còn gọi là QC - Quality Control.

Để đạt được Level 2 thì người quản lý phải thiết lập được các nguyên tắc cơ bản và quản lý các hoạt động diễn ra. Họ có trách nhiệm quản lý đội ngũ của mình

Các KPA( Key Process Areas) của nó chú trọng tới các thành phần sau :

- + Chế độ đãi ngộ
- + Đào tạo
- + Quản lý thành tích
- + Phân công lao động
- + Thông tin giao tiếp
- + Môi trường làm việc

*Sẽ có người hỏi để từ level1 tiến tới level 2 cần có những gì:*

**Trả lời:**

Trước tiên nó phải thỏa mãn các điều kiện ở level1

Tiếp theo là phải chú trọng tới các phần sau

1. Môi trường làm việc:

- Đảm bảo điều kiện làm việc
- Tạo hứng thú trong công việc
- Không bị ảnh hưởng, mất tập trung bởi các nhân tố khác

2. Thông tin:

Xây dựng cơ chế truyền tin thông suốt từ trên xuống dưới và ngược lại nhằm giúp cá nhân trong tổ chức chia sẻ thông tin, kiến thức, kinh nghiệm, các kỹ năng giao tiếp phối hợp và làm việc hiệu quả

3. Xây dựng đội ngũ nhân viên:

Ngay từ khâu tuyển dụng, lựa chọn kỹ càng và định hướng, thể chế hóa quy trình tuyển dụng

4. Quản lý thành tích:

Đẩy mạnh thành tích, công nhận năng lực, thành tích bằng cách thiết lập các tiêu chí khách quan để đánh giá và liên tục khuyến khích khả năng làm việc, tập trung phát triển sự nghiệp, xây dựng các mục tiêu tiếp theo.

5. Đào tạo:

Không chỉ đào tạo các kiến thức chuyên môn phục vụ cho dự án mà còn mở rộng đào tạo các kỹ năng then chốt, cần thiết như kỹ năng làm việc đội, nhóm, kỹ năng quản lý... nhằm tạo cơ hội cho người lao động phát huy khả năng, cơ hội học hỏi và phát triển bản thân.

#### 6. Chế độ đãi ngộ:

Hoạch định chiến lược đãi ngộ, thu thập ý kiến lực lượng lao động và công bố công khai. Chế độ đãi ngộ cần tập trung vào việc trả lương cho công nhân viên dựa vào vai trò, vị trí của họ (Position), Con người (Person) – thái độ và tác phong làm việc và Thành tích (Performance) mà họ đạt được, công hiến cho tổ chức. Đưa ra được chính sách lương, thưởng, phụ cấp các quyền lợi khác để khuyến khích các cá nhân dựa trên sự đóng góp của họ và cấp độ phát triển của toàn tổ chức.

### ◆ Level 3

Các vùng tiến trình chủ chốt ở mức 3 nhằm vào cả hai vấn đề về dự án và tổ chức, vì một tổ chức (công ty) tạo nên cấu trúc hạ tầng thể chế các quá trình quản lý và sản xuất phần mềm hiệu quả qua tất cả các dự án. Chúng gồm có Tập trung Tiến trình Tổ chức (Organization Process Focus), Phân định Tiến trình Tổ chức (Organization Process Definition), Chương trình Đào tạo (Training Program), Quản trị Phần mềm Tích hợp (Integrated Software Management), Sản xuất Sản phẩm Phần mềm (Software Product Engineering), Phối hợp nhóm (Intergroup Coordination), và Xét duyệt ngang hàng (Peer Reviews).

Để đạt được level 3 thì người quản lý phải biến đổi cải tiến các hoạt động đang diễn ra, cải tiến môi trường làm việc.

Lực lượng lao động sở hữu những kiến thức, kỹ năng cốt lõi  
KPA chú trọng tới các yếu tố sau :

- + Văn hóa cá thể
- + Công việc dựa vào kỹ năng
- + Phát triển sự nghiệp
- + Hoạch định nhân sự
- + Phân tích kiến thức và kỹ năng

#### *Từ Level 2 lên Level 3: Các KPA cần thực hiện*

##### 1. Phân tích kiến thức và kỹ năng:

Xác định những kỹ năng và kiến thức cần thiết để làm nền tảng cho hoạt động nhân sự. Lĩnh vực phân tích này bao gồm: xác định quy trình cần thiết để duy trì năng lực tổ chức, phát triển và duy trì các kỹ năng và kiến thức phục vụ công việc, dự báo nhu cầu kiến thức và kỹ năng trong tương lai.

##### 2. Hoạch định nguồn nhân lực:

Đây là lĩnh vực phối hợp hoạt động nhân sự với nhu cầu hiện tại và trong tương lai ở cả các cấp và toàn tổ chức. Hoạch định nguồn nhân lực có tính chiến lược cùng với quy trình theo dõi chặt chẽ việc tuyển dụng và các hoạt động phát triển kỹ năng sẽ tạo nên thành công trong việc hình thành đội ngũ.

##### 3. Phát triển sự nghiệp:

Tạo điều kiện cho mỗi cá nhân phát triển nghề nghiệp và có cơ hội thăng tiến trong nghề nghiệp, nó bao gồm: thảo luận về lựa chọn nghề nghiệp với mỗi cá nhân, xác định các cơ hội, theo dõi sự tiến bộ trong công việc, được động viên, khuyến khích đạt

mục tiêu công việc, giao quyền và khuyến khích thực hiện những mục tiêu trong công việc.

4. Các hoạt động dựa trên năng lực:

Ngoài các kỹ năng, kiến thức cốt lõi còn có hoạch định nhân lực, tuyển dụng dựa vào khả năng làm việc, đánh giá hiệu quả qua mỗi công việc và vị trí, xây dựng chế độ phúc lợi, đãi ngộ dựa trên hiệu quả... giúp bảo đảm rằng mọi hoạt động của tổ chức đều xuất phát từ mục đích phục vụ cho phát triển nguồn nhân lực

5. Văn hóa cá thể:

Tạo lập được cơ chế liên lạc thông suốt, kênh thông tin hiệu quả ở mọi cấp độ trong tổ chức, phối hợp được kinh nghiệm, kiến thức của mỗi người để hỗ trợ lẫn nhau, giúp nhau cùng tiến bộ. Trao quyền thúc đẩy nhân viên tham gia ý kiến, ra quyết định

◆ **Level 4**

Các vùng tiến trình chủ yếu ở mức 4 tập trung vào thiết lập hiểu biết định lượng của cả quá trình sản xuất phần mềm và các sản phẩm phần mềm đang được xây dựng. Đó là Quản lý quá trình định lượng (Quantitative Process Management) và Quản lý chất lượng phần mềm (Software Quality Management)

Lực lượng lao động làm việc theo đội, nhóm và được quản lý một cách định lượng.

Các KPA của level 4 chú trọng tới:

- + Chuẩn hóa thành tích trong tổ chức
- + Quản lý năng lực tổ chức
- + Công việc dựa vào cách làm việc theo nhóm
- + Xây dựng đội ngũ chuyên nghiệp
- + Cố vấn

Để đạt được level 4 thì phải đo lường và chuẩn hóa. Đo lường hiệu quả đáp ứng công việc, chuẩn hóa phát triển các kỹ năng, năng lực cốt lõi

Level 4 này sẽ chú trọng vào những người đứng đầu của một công ty, họ có khả năng quản lý các công việc như thế nào

◆ **Level 5**

Các vùng tiến trình chủ yếu ở mức 5 bao trùm các vấn đề mà cả tổ chức và dự án phải nhắm tới để thực hiện hoàn thiện quá trình sản xuất phần mềm liên tục, đo đếm được. Đó là Phòng ngừa lỗi (Defect Prevention), Quản trị thay đổi công nghệ (Technology Change Management), và Quản trị thay đổi quá trình (Process Change Management) Để đạt được level 4 thì phải đo lường và chuẩn hóa. Đo lường hiệu quả đáp ứng công việc, chuẩn hóa phát triển các kỹ năng, năng lực cốt lõi

Để đạt được Level 5 thì doanh nghiệp đó phải liên tục cải tiến hoạt động tổ chức, tìm kiếm các phương pháp đổi mới để nâng cao năng lực làm việc của lực lượng lao động trong tổ chức, hỗ trợ các nhân phát triển sở trường chuyên môn.

Chú trọng vào việc quản lý, phát triển năng lực của nhân viên

Huấn luyện nhân viên trở thành các chuyên gia

### 9.6.3. So sánh giữa CMM và CMMi

Hẳn những ai quan tâm tới CMM, cũng đã từng nghe hoặc biết qua CMMi . Và khi đã nghe qua, tôi đoán không ít người thầm thắc mắc, hoặc tự hỏi rằng không biết CMM với CMMi nó khác nhau ra sao nhỉ ???

Nếu đặc tả một các trực quan thì ta thấy CMM và CMMi chỉ khác nhau có một chữ “I” (Integration). Nhưng một chữ “I” đó thôi cũng tạo ra sự khác biệt đáng kể giữa CMM và CMMi rồi.

Trước tiên hãy xem qua nguồn gốc của hai thứ này một tí . Nếu nói rằng CMM ra đời trước CMMi thì cũng đúng nhưng mà nói CMMi có trước từ khi CMM ra đời cũng chẳng sai. Thật ra khi CMM được chính thức công bố vào cuối năm 1990 thì CMMi đã được manh mẽ được nhắc đến từ nhiều năm trước đó (chính xác hơn là từ 1979- Crosby’s maturity grid (Quality is Free)) thông qua cấu trúc Continuous & Staged. Có thể nói CMMi là một phiên bản cải thiện tất yếu của CMM.

Trong khi CMM được hoàn thiện và phát triển bởi viện SEI của Mỹ, thì CMMi là sản phẩm của sự cộng tác giữa viện này và chính phủ Mỹ. Từ khi CMM được công nhận và áp dụng trên thế giới thì tầm quan trọng của nó đã vượt qua giới hạn của một viện khoa học. Với tốc độ phát triển không ngừng và đòi hỏi sự cải thiện liên tục trong ngành công nghệ thông tin, việc chính phủ Mỹ cùng với viện SEI kết hợp để hoàn thiện CMM và cho ra đời phiên CMMi là một hệ quả tất yếu.

Mô hình CMM trước đây gồm có 5 mức: khởi đầu, lặp lại được, được định nghĩa, được quản lý và tối ưu. Một điểm đặc biệt là mỗi doanh nghiệp có thể áp dụng mô hình CMM ở bất kỳ mức nào mà không cần tuân theo bất kỳ một qui định nào, không cần phải đạt mức thấp trước rồi mới có thể đạt mức cao (có thể đi thẳng lên mức cao, hoặc cũng có thể tự hạ xuống mức thấp hơn). Về nguyên tắc, SEI không chính thức đứng ra công nhận CMM mà thông qua các tổ chức tư vấn, các đánh giá trưởng được SEI ủy quyền và thừa nhận.

Từ cuối 2005, SEI không tổ chức huấn luyện SW-CMM và chỉ thừa nhận các đánh giá theo mô hình CMMi mới từ tháng 12/2005. CMMi được tích hợp từ nhiều mô hình khác nhau, phù hợp cho cả những doanh nghiệp phần cứng và tích hợp hệ thống, chứ không chỉ đơn thuần áp dụng cho doanh nghiệp sản xuất phần mềm như CMM trước đây. Có 4 mô hình áp dụng CMMi là CMMi-SW (dành cho công nghệ phần mềm), CMMi-SE/SW (dành cho công nghệ hệ thống và phần mềm), CMMi-SE/SW/IPPD (dành cho công nghệ hệ thống + công nghệ phần mềm với việc phát triển sản phẩm và quy trình tích hợp), CMMi-SE/SW/IPPD/SS (dành cho công nghệ hệ thống + công nghệ phần mềm với việc phát triển sản phẩm và quy trình tích hợp có sử dụng thầu phụ). Có 2 cách diễn đạt và sử dụng CMMi: Staged (phù hợp cho tổ chức có trên 100 người) và Continuous (phù hợp cho tổ chức dưới 40 người). CMMi cũng bao gồm 5 mức như CMM: khởi đầu, lặp lại được, được định nghĩa, được quản lý và tối ưu.

CMMi đưa ra cụ thể các mô hình khác nhau cho từng mục đích sử dụng có đặc riêng bao gồm :

- CMMi-SW mô hình chỉ dành riêng cho phần mềm.

- CMMI-SE/SW mô hình tích hợp dành cho các hệ thống và kỹ sư phần mềm.
- CMMI-SE/SW/IPPD mô hình dành cho các hệ thống, kỹ sư phần mềm và việc tích hợp sản phẩm cùng quá trình phát triển nó.

**Như vậy các đặc điểm khác nhau quan trọng nhất giữa CMM và CMMI là gì?**

**Thử nghĩa qua xem trong CMMI hiện nay có gì khác với thằng anh sinh ra trước nó (CMM):**

- CMMI đưa ra cụ thể hơn 2 khái niệm đối ứng stageds VS continuous
- Chu kỳ phát triển của CMMI được phát triển sớm hơn.
- Nhiều khả năng tích hợp (Integration) hơn
- Sự đo lường, định lượng được định nghĩa hẳn là một Process area (vùng tiến trình). Chứ không hẳn chỉ là một đặc trưng cơ bản.
- Bao hàm nhiều Process Areas hơn.
- Đạt được thành quả dễ dàng hơn với mỗi Process area.

#### **9.6.4. Lợi ích của CMM đem lại cho doanh nghiệp**

##### **1. Viễn cảnh mà CMM mang lại**

- Ý nghĩa của việc áp dụng những nguyên tắc:
  - + Quản lý chất lượng tổng thể
  - + Quản lý nguồn nhân lực
  - + Phát triển tổ chức
    - + Tính cộng đồng
  - + Phạm vi ảnh hưởng rộng: từ các ngành công nghiệp đến chính phủ
  - + Hoàn toàn có thể xem xét và mở rộng tầm ảnh hưởng với bên ngoài
  - + Chương trình làm việc nhằm cải tiến, nâng cao hoạt động của đội ngũ lao động
    - + Đánh giá nội bộ
  - + Các hoạt động của đội ngũ lao động được cải tiến
- + Các chương trình nhằm nâng cao năng lực, hiệu quả công việc luôn được tổ chức

##### **2. Mục tiêu chiến lược**

- Cải tiến năng lực của các tổ chức phần mềm bằng cách nâng cao kiến thức và kỹ năng của lực lượng lao động
- Đảm bảo rằng năng lực phát triển phần mềm là thuộc tính của tổ chức không phải của một vài cá thể
- Hướng các động lực của cá nhân với mục tiêu tổ chức
- Duy trì tài sản con người, duy trì nguồn nhân lực chủ chốt trong tổ chức

##### **3. Lợi ích CMM mang lại cho Doanh nghiệp gói gọn trong 4 từ: Attract, Develop, Motivate và Organize**

##### **4. Lợi ích CMM mang lại cho người lao động:**

- Môi trường làm việc, văn hóa làm việc tốt hơn
- Vạch rõ vai trò và trách nhiệm của từng vị trí công việc
- Đánh giá đúng năng lực, công nhận thành tích
- Chiến lược, chính sách đãi ngộ luôn được quan tâm
- Có cơ hội thăng tiến
  - Liên tục phát triển các kỹ năng cốt yếu.

## CHƯƠNG 10: QUẢN LÝ CẤU HÌNH

### *Mục đích*

*Mục đích của chương này là giới thiệu về quá trình quản lý mã và tài liệu của một hệ thống phần mềm mở rộng. Khi bạn đọc xong chương này bạn sẽ:*

- *hiểu tại sao quản trị cấu hình phần mềm lại được đòi hỏi cho những hệ thống phần mềm phức tạp;*
- *được giới thiệu tới bốn hoạt động quản trị cấu hình cơ bản- kế hoạch quản lý, quản lý thay đổi, quản lý phiên bản và bản phát hành, và xây dựng hệ thống.*
- *hiểu công cụ CASE được sử dụng như thế nào để hỗ trợ cho quá trình quản lý.*

### 10.1. Giới thiệu

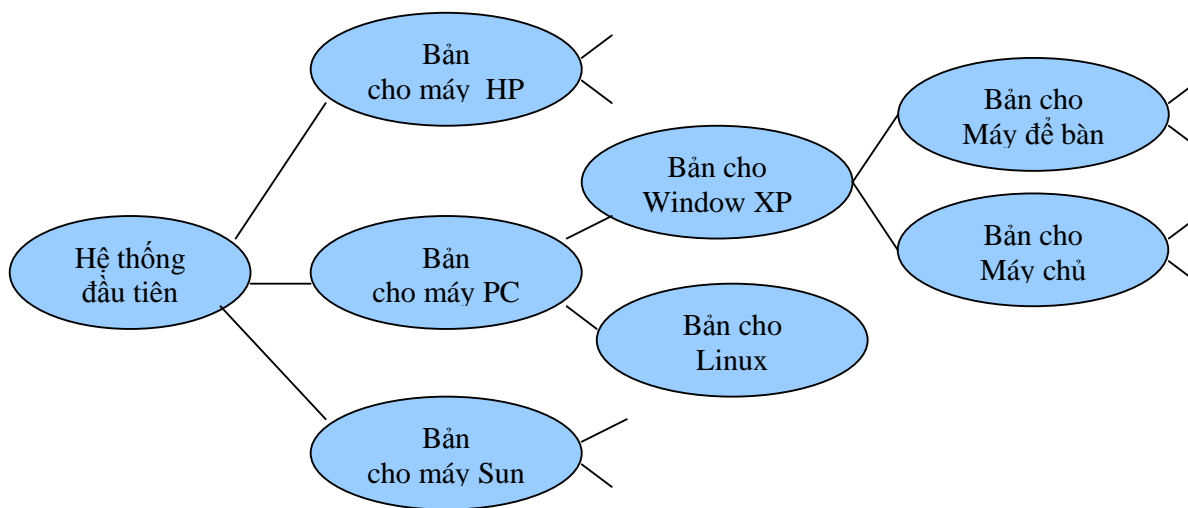
Quản trị cấu hình(CM ) là sự mở rộng và sử dụng của những tiêu chuẩn và thủ tục cho quản lý một hệ thống phần mềm mở rộng. Như đề cập ở chương 7, những đòi hỏi của hệ thống luôn luôn thay đổi trong suốt quá trình phát triển và sử dụng, và bạn phải tổ chức sát nhập những yêu cầu này vào trong những phiên bản mới của hệ thống. Bạn cần quản lý việc mở rộng hệ thống bởi vì rất dễ sai lầm khi theo dõi những thay đổi nào đã được đưa vào phiên bản nào của hệ thống. Những phiên bản hợp những đề xuất để thay đổi, sự chính xác và tương thích cho những phần cứng và hệ điều hành khác nhau. Có thể một vài phiên bản được mở rộng và sử dụng trong cùng một thời gian. Nếu bạn không có một cách thức quản lý cấu hình hiệu quả trong trường hợp này bạn sẽ phải tốn rất nhiều nỗ lực cho việc sửa chữa những phiên bản lỗi của hệ thống, hay phân phối những phiên bản này tới người dùng hoặc mất kiểm soát nơi mà mã phần mềm được lưu trữ.

Thủ tục quản trị cấu hình cho biết làm thế nào để ghi chép và giám sát những thay đổi của hệ thống được đề xuất, làm sao để liên hệ những điều này tới những thành phần của hệ thống và những phương pháp được sử dụng để chỉ ra những phiên bản khác nhau của hệ thống. Những công cụ quản trị cấu hình được sử dụng để lưu trữ những phiên bản của thành phần hệ thống, xây dựng hệ thống từ những thành phần này và theo dõi việc phân phối các phiên bản tới khách hàng.

Quản trị cấu hình đôi khi được xem như là một phần của quá trình quản lý chất lượng phần mềm (chương 27), vì cùng việc quản lý chất lượng và trách nhiệm quản lý. Phần mềm ban đầu được đưa ra bởi người phát triển với sự bảo đảm về chất lượng. Còn QA thì kiểm tra việc chất lượng của hệ thống có chấp nhận được hay không. Nó trở thành một hệ thống điều khiển, điều đó có nghĩa rằng sự thay đổi của hệ thống phải được chấp nhận và ghi lại trước khi nó được thực thi. Điều khiển hệ thống đôi khi được gọi là *đường cơ bản(baselines)* vì nó là điểm bắt đầu cho một cuộc cách mạng xa hơn.

Có rất nhiều nguyên nhân giải thích tại sao những hệ thống lại tồn tại trong những cấu hình khác nhau. Các cấu hình có thể được đưa ra cho những máy tính khác nhau, cho những hệ điều hành khác nhau, hay cho việc kết hợp những chức năng riêng của khách hàng...(hình 9.1) Những nhà quản trị cấu hình có trách nhiệm trong việc giám sát sự khác nhau giữa các

phiên bản phần mềm, đảm bảo rằng những phiên bản mới được phát hành trong tầm kiểm soát và đưa tới đúng khách hàng tại đúng thời điểm.



Hình 29.1-Quan hệ các hệ thống

Việc định nghĩa và sử dụng những tiêu chuẩn quản trị cấu hình là cần thiết với việc chứng nhận chất lượng trong cả hai tiêu chuẩn **ISO9000** và **CMM** và **CMMI**. Một ví dụ về một tiêu chuẩn là **IEEE 828-1998**, là một chuẩn cho chiến lược quản trị cấu hình. Trong một công ty, những tiêu chuẩn này có thể được đưa vào sổ tay hướng dẫn chất lượng hay sách chỉ dẫn quản trị cấu hình. Tất nhiên, những tiêu chuẩn mở rộng nói chung có thể được sử dụng như một nền tảng cho những chuẩn được tổ chức chi tiết hơn.

Một quá trình phát triển phần mềm thông thường dựa trên mô hình *thác nước* (waterfall), phần mềm được đưa tới đội ngũ nhà quản trị cấu hình sau khi việc mở rộng hoàn tất và những thành phần khác nhau đã được kiểm tra. Đội ngũ này sau đó sẽ tiếp quản trách nhiệm cho việc xây dựng hệ thống hoàn thiện và quản lý kiểm tra hệ thống. Những lỗi được phát hiện quá trình kiểm tra hệ thống sẽ được gửi lại đội ngũ nhà mở rộng để sửa chữa. Sau khi lỗi đã được sửa một phiên bản mới của thành phần được sửa chữa được gửi tới nhà bảo đảm chất lượng. Nếu chất lượng là chấp nhận được thì nó có thể trở thành một con đường có bản mới đi tới việc phát triển hệ thống xa hơn.

Trong mô hình này, việc những **CM** điều khiển mở rộng hệ thống và kiểm tra các quá trình đã có ảnh hưởng tới việc phát triển các tiêu chuẩn quản trị cấu hình. Hầu hết các chuẩn **CM** đều có những tiền đề được nhúng vào cái mà mô hình *thác nước* sẽ được sử dụng cho việc phát triển hệ thống. Điều đó có nghĩa là những tiêu chuẩn phải được cập nhật thường xuyên để tương thích với những thay đổi và những tiến bộ mới của hệ thống. Hass (Hass,2003) cho rằng những sự tương thích cho quá trình mở rộng hệ thống phần mềm diễn ra rất nhanh chóng.

Để tăng trưởng lợi nhuận, một số tổ chức đã mở rộng cách tiếp cận tới quản trị cấu hình để hỗ trợ việc phát triển và kiểm tra hệ thống hiện hành. Cách tiếp cận này dựa trên tần suất xây dựng rất cao toàn bộ hệ thống từ những thành phần của nó:

1. Các tổ chức phát triển thường sắp xếp phân phối thời gian cho các thành phần của hệ thống nhằm theo dõi hoạt động của nó. Nếu những người phát triển có những phiên bản mới của các thành phần này thì họ sẽ thử nghiệm ngay trong khoảng thời gian này. Các thành phần có thể không được hoàn thành nhưng nó có thể cung cấp những chức năng cơ bản để có thể được kiểm tra.
2. Một phiên bản mới của hệ thống được xây dựng từ những thành phần này bằng cách biên tập và liên kết chúng lại.
3. Hệ thống này sẽ được đưa tới những người kiểm tra (*tester*), nơi mà sẽ thi hành một loạt các cuộc kiểm tra được xác định trước. Trong cùng thời gian này những người phát triển vẫn làm việc với những phần việc của họ, như là thêm các tính năng mới và sửa lỗi được tìm ra ở kỳ kiểm tra trước.
4. Những lỗi được tìm thấy khi kiểm tra sẽ là những tư liệu và được đưa trở lại người phát triển. Họ sửa những lỗi này ở những phiên bản sau của các thành phần.

Những tiện lợi của việc xây dựng phần mềm hàng ngày theo kiểu này cho phép tìm ra các vấn đề nảy sinh trong tương tác giữa các thành phần một cách dễ dàng khi mà mức độ xử lý ngày càng tăng. Hơn thế nữa, Việc này còn khuyến khích việc kiểm tra tỉ mỉ các thành phần. Về mặt tâm lý, các nhà mở rộng còn bị đè nặng dưới áp lực của việc “*không được phá vỡ hệ thống*” vì mỗi thành phần có thể làm hỏng toàn bộ hệ thống.

Sự sử dụng thành công việc xây dựng hệ thống liên tục đòi hỏi một tiến trình quản lý nghiêm ngặt để theo dõi các vấn đề đã được phát hiện và sửa chữa. Quản lý cấu hình tốt là cần thiết để đi tới thành công.

Quản lý cấu hình với những cách thức tiếp cận linh hoạt và mềm dẻo không thể dựa trên những nhà sản xuất cứng nhắc và trên công việc giấy tờ. Trong khi điều này là cần thiết cho những dự án lớn và phức tạp, nó lại làm chậm tiến trình mở rộng. Việc theo dõi sát sao là cần thiết cho những hệ thống lớn và phức tạp được mở rộng thông qua các thông số, nhưng không cần thiết cho những dự án nhỏ. Trong những dự án này toàn bộ thành viên của nhóm làm việc với nhau trong cùng một phòng, và việc nâng cấp có liên quan đến việc theo dõi làm chậm tiến trình mở rộng. Tuy nhiên, điều đó không hoàn toàn có nghĩa là **CM** nên được từ bỏ khi mà việc mở rộng linh hoạt được yêu cầu. Đúng hơn là, những quá trình linh hoạt sử dụng những công cụ **CM** đơn giản, giống như là một mô hình quản lý và công cụ xây dựng hệ thống. Tất cả các thành viên phải học sử dụng các công cụ này và thích nghi với những kỉ luật mà họ phải tuân hành.

## 10.2. Kế hoạch quản trị cấu hình

Một kế hoạch quản trị cấu hình mô tả những tiêu chuẩn và thủ tục là những cái có thể được sử dụng cho quản trị cấu hình. Điểm bắt đầu cho việc mở rộng kế hoạch có thể là một tập các tiêu chuẩn quản trị cấu hình, và những cái này có thể tương thích cho phù hợp với những yêu cầu và sức ép của mỗi dự án. Kế hoạch quản lý có thể được phân theo các mục sau:

1. Định nghĩa những cái gì được quản lý và chiến lược bạn sử dụng để chỉ ra các thực thể này.



2. Đưa ra người nào có trách nhiệm cho những thủ tục quản trị cấu hình và cho việc đưa ra những thực thể tới những người quản trị cấu hình.
3. Định nghĩa những đường lối quản lý cái mà toàn đội phải sử dụng để thay đổi việc điều khiển và các mô hình quản lý.
4. Chỉ ra những công cụ bạn có thể sử dụng cho quản trị cấu hình và những tiến trình sử dụng những công cụ này.
5. Mô tả kiến trúc của cấu hình cơ sở dữ liệu được dùng để thông báo thông tin về cấu hình những thông tin có thể được lưu trữ trong cơ sở dữ liệu.

Bạn cũng có thể tính đến những vấn đề khác trong kế hoạch quản trị như là việc quản lý phần mềm từ những nguồn mở rộng và thủ tục kiểm toán cho tiến trình quản lý.

Một phần quan trọng của kế hoạch quản trị (KHQT) là việc định nghĩa trách nhiệm. Kế hoạch có thể đưa ra người có trách nhiệm cho việc phân phối các tài liệu hoặc phần mềm để bảo đảm chất lượng. Cá nhân có trách nhiệm cho việc phân phối tài liệu cần không trùng với người có trách nhiệm cho sản xuất tài liệu. Để làm đơn giản mặt này, người quản lý dự án hoặc đội ngũ đứng đầu phải có trách nhiệm cho tất cả tài liệu được họ đưa ra.

#### **11.1.1. Xác minh các cấu hình**

Trong một hệ thống phần mềm lớn, có thể có hàng nghìn môđun mã nguồn, tập lệnh kiểm tra, tài liệu thiết kế và nhiều thứ khác. Chúng được đưa ra bởi những người khác nhau và khi tạo ra, có thể được gán giống nhau hoặc là trùng tên. Để theo dõi tất cả những thông tin này thì các file cần phải được tìm thấy khi cần thiết, bạn cần có một chiến lược xác minh nhất quán cho mọi chỉ mục trong hệ thống quản trị cấu hình.

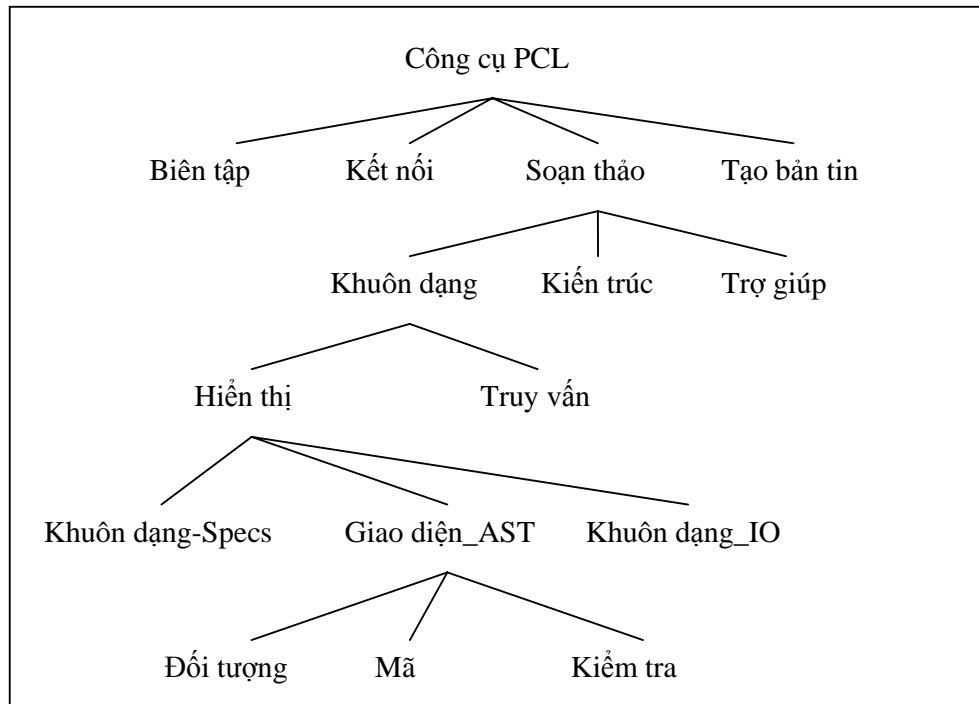
Trong suốt tiến trình lập kế hoạch quản trị cấu hình, bạn quyết định chính xác mục nào(hoặc lớp mục nào) được điều khiển. Những tài liệu hoặc các nhóm tài liệu gần nhau khi được xác lập là những tài liệu trang trọng rõ ràng. Những kế hoạch của dự án, bản mô tả, bản thiết kế, chương trình và kiểm tra dữ liệu thích hợp được duy trì thường xuyên như là một loại của cấu hình. Tất cả các tài liệu mà có thể có ích cho cuộc cách mạng hệ thống trong tương lai đều có thể được điều hành bởi hệ thống quản trị cấu hình.

Tuy nhiên, điều đó không có nghĩa là mọi tài liệu hay tập lệnh được đưa ra đều chịu sự giám sát của điều hành cấu hình. Những tài liệu kỹ thuật, những phút gặp gỡ trao đổi, những bản phác thảo, những đề xuất... có thể không có liên quan hoặc không cần thiết cho sự duy trì hệ thống trong tương lai.

Kế hoạch xác minh các chỉ mục cho cấu hình cần được gắn liền với một tên duy nhất cho mọi tài liệu dưới sự giám sát của việc điều hành cấu hình. Cái tên này có thể phản ánh loại mục, phần của hệ thống mà nó ám chỉ tới. Trong việc đánh tên kế hoạch của bạn, bạn có thể mong rằng phản ánh mối quan hệ giữa các chỉ mục bằng việc đảm bảo rằng những tài liệu có liên quan với nhau có một gốc chung với tên của chúng. Từ đó bạn có thể định nghĩa một lớp các tên được phân cấp như:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE

PCL-TOOLS/EDIT/HELP/QUERY/HELPPFRAMES/FR-1



Hình 10.2-Phân cấp cấu hình

Phần đầu của tên là tên dự án, PCL-TOOLS. Trong dự án này, có một số lượng các công cụ đang được mở rộng, do đó tên công cụ (EDIT) được sử dụng như phần tiếp theo của tên. Mỗi công cụ bao gồm nhiều môđun tên khác nhau mà tên của nó tạo nên thành phần tiếp theo của chỉ mục xác minh (FORMS, HELP). Quá trình phân tích này còn tiếp tục cho tới tài liệu chính thức cấp cơ bản là những tham chiếu (hình 29.2). Việc phân cấp tài liệu sẽ đưa ra những chỉ mục cấu hình chính thức. Hình 9.2 chỉ ra ba chỉ mục được yêu cầu cho mỗi thành phần: một đối tượng mô tả (*OBJECTS*), mã nguồn của phần tử (*CODE*) và một tập các cuộc kiểm tra cho các phần tử (*TESTS*). Các chỉ mục như những frame trợ giúp cũng được quản lý và có nhiều tên khác nhau (*FR-1*)

Chiến lược phân cấp đánh dấu tên rất đơn giản và dễ hiểu, và đôi khi nó còn chỉ ra cấu trúc thư mục thường là nơi chứa file của dự án. Tuy nhiên nó phản ánh cấu trúc của dự án mà phần mềm đã được phát triển. Tên của các chỉ mục cấu hình liên kết các thành phần với dự án thực tế và có thể làm giảm việc sử dụng trở lại. Có thể rất khó tìm thấy các thành phần có liên quan tới nhau (chẳng hạn, mọi thành phần đều được mở rộng cùng một lập trình viên) do đó mối quan hệ không được phản ánh trong chiến lược đánh tên các chỉ mục.

### 11.1.2. Cơ sở dữ liệu của cấu hình

Cơ sở dữ liệu cấu hình được sử dụng để thông báo các thông tin có liên quan về cấu hình hệ thống và các chỉ mục. Bạn sử dụng cơ sở dữ liệu **CM** để giúp truy cập vào những thay đổi của hệ thống và đưa ra những thông báo về tiến trình quản lý. Như một phần của tiến trình kế hoạch hóa **CM**, bạn nên đưa ra giản đồ cơ sở dữ liệu quản lý, định hình việc thu thập thông tin đưa vào trong cơ sở dữ liệu, từ đó giúp thông báo và truy xuất thông tin về dự án.

Một cơ sở dữ liệu cấu hình không bao gồm thông tin về các chỉ mục cấu hình. Nó có thể thông báo các thông tin về những người sử dụng của các thành phần, hệ thống khách hàng, nền tảng thực thi, những đề xuất cho thay đổi. Nó có thể cung cấp những giải đáp cho rất nhiều câu hỏi khác nhau về cấu hình hệ thống. Những thắc mắc điển hình như là:

1. Khách hàng nào vừa mới được phân phối một phiên bản riêng biệt của hệ thống?
2. Cấu hình phần cứng và hệ điều hành cần đòi hỏi như thế nào để chạy phiên bản đó của hệ thống?
3. Có bao nhiêu phiên bản đã được tạo ra và ngày tạo ra chúng?
4. Phiên bản nào có thể có hiệu quả nếu một thành phần riêng biệt bị thay đổi?
5. Có bao nhiêu đòi hỏi thay đổi là nổi bật trên một phiên bản thực tế?
6. Có bao nhiêu trách lỗi được thông báo tồn tại trên một phiên bản thực tế?

Nếu là lý tưởng, cơ sở dữ liệu cấu hình nên được tích hợp cùng với phiên bản của hệ thống quản lý nơi mà được sử dụng để lưu trữ và quản lý các tài liệu chính thức của dự án. Cách làm này, được hỗ trợ bởi một số công cụ **CASE**, tạo cho chúng có khả năng liên kết trực tiếp các tài liệu và thành phần có hiệu quả. Liên kết giữa các tài liệu (như tài liệu thiết kế) và mã chương trình có thể được duy trì để bạn có thể tìm thấy được mọi thứ mà bạn cần phải chỉnh sửa lại khi mà một sự thay đổi được đề xuất.

Tuy nhiên, các công cụ tích hợp **CASE** cho quản trị cấu hình là rất đắt. Rất nhiều công ty không sử dụng chúng nhưng lại duy trì cơ sở dữ liệu cấu hình tách rời với hệ thống quản lý cấu hình của họ. Họ lưu trữ các chỉ mục cấu hình như là những file trong cấu trúc thư mục hoặc trong hệ thống quản lý phiên bản như **CVS**.

Cơ sở dữ liệu cấu hình lưu trữ thông tin về các chỉ mục cấu hình và tham chiếu đến tên của chúng trong hệ thống quản lý phiên bản hoặc file lưu trữ. Trong khi đó là một cách làm rẻ tiền và mèm dẻo thì vấn đề đặt ra với nó là các chỉ mục cấu hình có thể bị thay đổi mà không thông qua cơ sở dữ liệu cấu hình. Từ đó, bạn không thể chắc chắn rằng cơ sở dữ liệu cấu hình là một hệ được cập nhật hàng ngày các trạng thái của hệ thống.

## 11.2. Quản lý việc thay đổi

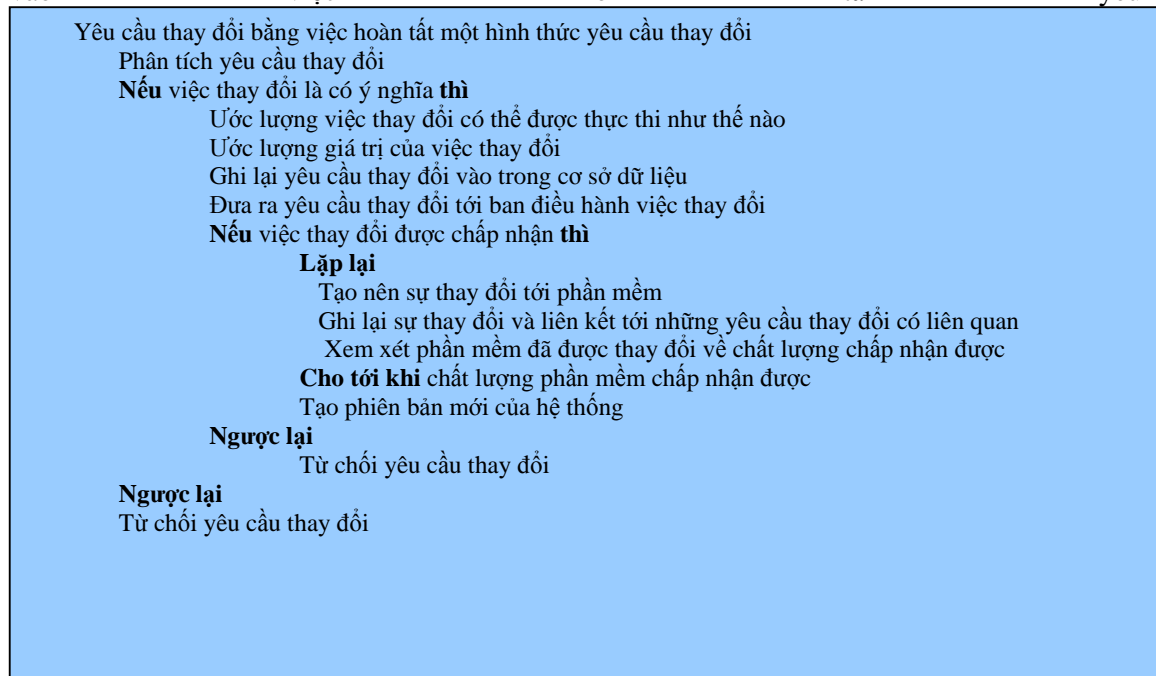
Thay đổi là một thực tế của đời sống của những hệ thống phần mềm lớn. Như đã đề cập đến ở các chương trước, việc tổ chức cần và đòi hỏi thay đổi trong suốt quá trình sống của một hệ thống.

Điều đó có nghĩa là bạn cần tạo ra những sự thay đổi cho phù hợp với hệ thống phần mềm. Để chắc chắn rằng sự thay đổi là được cập nhật vào hệ thống theo đúng đường lối có kiểm soát, bạn cần có một tập các công cụ hỗ trợ, các thủ tục quản lý thay đổi.

Các thủ tục quản lý thay đổi được liên hệ với việc phân tích giá trị và lợi ích của việc thay đổi theo đề xuất. Tiến trình quản lý thay đổi (*hình 29.3*) có thể được hình thành có hiệu quả khi phần mềm hay tài liệu liên quan được vạch ra bởi đội ngũ những người quản trị cấu hình.

Chặng đầu tiên trong tiến trình quản lý là hoàn tất hình thành yêu cầu việc mô tả (Change Request Form-**CRF**) sự thay đổi mà hệ thống đòi hỏi. **CRF** đưa ra thông báo đến lưu ý đến sự thay đổi, giá trị ước lượng của sự thay đổi và ngày khi mà sự thay đổi được yêu cầu, tán thành, thực thi và có hiệu lực. **CRF** có thể bao gồm một phần phân tích bên ngoài việc thay đổi được thực thi thế nào.

Một ví dụ của việc hình thành yêu cầu thay đổi được chỉ ra trong *hình 29.4*. **CRF** được xác định trong suốt quá trình lập kế hoạch quản trị cấu hình. Một ví dụ **CRF** là có thể được sử dụng trong một dự án cho một hệ thống lớn và phức tạp. Đối với những dự án nhỏ, chúng tôi cho rằng yêu cầu thay đổi có thể được thông báo chính thức, nhưng **CRF** nên tập trung vào việc mô tả yêu cầu



Hình 10.3 Tiến trình quản lý sự thay đổi

cầu thay đổi. Kỹ sư thiết kế sự thay đổi quyết định việc làm cách nào để thực thi sự thay đổi đó trong một tình huống cụ thể.

Một **CRF** đã được xem xét, thì nó nên được đăng nhập vào trong cơ sở dữ liệu cấu hình. Sau đó việc yêu cầu thay đổi được phân tích để kiểm tra rằng yêu cầu thay đổi là cần thiết. Một số yêu cầu có thể bị hiểu lầm, với lỗi hệ thống và sự thay đổi của hệ thống là không cần thiết. Một số khác có thể tham chiếu tới những lỗi đã được biết đến. Nếu người phân tích khám phá ra rằng yêu cầu thay đổi là không có ý nghĩa, được mô phỏng hay là đã được xem xét, thì sự thay đổi sẽ bị bác bỏ. Bạn nên nói với người đệ trình việc yêu cầu thay đổi rằng tại sao nó bị từ chối.

Đối với sự thay đổi có ý nghĩa, chặng tiếp theo của quá trình là đánh giá và suy xét giá trị sự thay đổi. Tác động của sự thay đổi lên hệ thống phải được kiểm tra. Điều đó liên quan đến việc chỉ ra mọi thành phần chịu sự tác động của sự thay đổi có sử dụng thông tin ở cơ sở dữ liệu cấu hình và mã nguồn của phần mềm. Nếu việc tạo nên sự thay đổi lại xa hơn những gì hệ thống cần thì rõ ràng cái giá của việc thực thi sự thay đổi phải tăng lên. Tiếp theo, yêu cầu thay đổi tới hệ thống sẽ được đánh giá. Cuối cùng, cái giá của việc thay đổi sẽ được ước lượng, gửi tới tài khoản giá trị của việc thay đổi các thành phần liên quan.

Một ban quản lý thay đổi (*Change Control Board\_CCB*) nên xem xét và phê chuẩn mọi yêu cầu thay đổi nếu những sự thay đổi không chỉ đơn giản liên quan tới việc chính xác những lỗi nhỏ trên những màn hình hiển thị, những trang Web hay trong những tài liệu. Ban **CCB** xem xét kĩ tác động của một sự thay đổi có chiến lược và tổ chức hơn là so với một quan điểm về kĩ thuật. Những ban này nên quyết định xem sự thay đổi là xác đáng về mặt kinh tế hay không và nên dành ưu tiên cho những sự thay đổi đã được chấp nhận.

Ban **CCB** (*Change Control Board*) bao hàm ý nghĩ về một đội ngũ lớn những nhà đưa ra những quyết định thay đổi. Như cấu trúc thông thường của **CCB**, bao gồm những khách hàng lâu

Hình thức yêu cầu thay đổi	
<b>Dự án:</b> Proteus/Công cụ_PCL	<b>Số:</b> 23/02
<b>Người yêu cầu thay đổi:</b> I.Sommerville	<b>Ngày:</b> 1/12/02
<b>Yêu cầu thay đổi:</b> Khi một thành phần được chọn từ cấu trúc, hiển thị tên của file chứa thành phần đó	
<b>Người phân tích thay đổi:</b> G.Dean	<b>Ngày phân tích:</b> 10/12/02
<b>Thành phần bị tác động:</b> Hiển thị Icon.Select, Hiển thị Icon.Display	
<b>Thành phần liên kết:</b> FileTable	
<b>Đánh giá sự thay đổi:</b> Tương đối đơn giản để thực thi như là với một file có tên Table có hiệu lực. Yêu cầu có sự thiết kế và thực thi đối với trường hiển thị. Không có sự thay đổi nào tới các thành phần liên quan được yêu cầu.	
<b>Mức ưu tiên thay đổi:</b> Thấp	
<b>Thực thi sự thay đổi:</b>	
<b>Đánh giá nỗ lực:</b> 0.5 ngày	
<b>Ngày tới CCB:</b> 15/12/02	<b>Ngày quyết định của CCB:</b> 1/2/03
<b>Quyết định của CCB:</b> Chấp nhận thay đổi, Thay đổi sẽ được thực thi trong phiên bản 2.1	
<b>Người thực thi:</b>	<b>Ngày thay đổi:</b>
<b>Ngày đệ trình tới QA:</b>	<b>Quyết định của QA:</b>
<b>Ngày đệ trình tới CM:</b>	
<b>Chú giải</b>	

Hình 10.4 Hình thức yêu cầu thay đổi được hoàn thành

năm, những người quản lý trực tiếp (*contractor staff*) là những yêu cầu cho một dự án nghiêm ngặt. Tuy nhiên, với những dự án có kích cỡ nhỏ và vừa, **CCB** có thể đơn giản chỉ bao gồm một người quản lý dự án cộng với một hoặc hai kĩ sư những người không trực tiếp liên quan đến việc phát triển phần mềm. Trong một số trường hợp, **CCB** có thể là một người theo dõi thay đổi đơn lẻ người đưa ra lời khuyên về sự thay đổi là xác đáng hay không.

Quản lý thay đổi cho những sản phẩm phần mềm chung được đóng gói so với những hệ thống được làm riêng cho những khách hàng cụ thể phải được điều khiển trong những phương thức khác nhau. Trong những hệ thống này, khách hàng không liên quan trực tiếp tới do đó sự thay đổi tương ứng tới việc kinh doanh của khách hàng không phải là một sản phẩm. Yêu cầu thay đổi trong những sản phẩm này thường được gán với những lỗi trong hệ thống đã được khám phá trong suốt quá trình kiểm tra hay bởi những khách hàng sau khi phần mềm đã được phát hành. Các khách hàng có thể sử dụng một trang Web hoặc e\_mail để thông báo lỗi. Một đội ngũ những người quản lý lỗi sẽ kiểm tra thông báo lỗi là có hiệu lực và chuyển chúng tới hệ thống yêu cầu thay đổi chính thức. Như với các loại hệ thống khác, thay đổi phải được ưu tiên cho việc thực thi và lỗi có thể không được khắc phục nếu như giá thành khắc phục quá cao.

Trong suốt quá trình phát triển, khi những phiên bản mới của hệ thống được tạo ra thông qua việc xây dựng hệ thống hàng ngày, một tiến trình quản lý thay đổi đơn giản hơn sẽ được sử dụng. Những vấn đề và thay đổi vẫn phải được thông báo, nhưng những thay đổi mà chỉ ảnh hưởng tới các thành phần và module khác nhau không cần thiết phải được đánh giá độc lập. Chúng được đưa trực tiếp tới hệ thống những người phát triển. Hệ thống những người phát triển hoặc là chấp nhận chúng hoặc là đưa ra lý do tại sao chúng không được yêu cầu. Những thay đổi ảnh hưởng những module hệ thống được đưa ra bởi những đội ngũ người phát triển khác nhau, tuy nhiên, nên được đánh giá bởi một chuyên gia quản lý thay đổi, người đặt quyền ưu tiên về thực thi cho chúng.

Trong một số phương pháp mềm dẻo, như lập trình xa, các khách hàng được liên quan trực tiếp tới việc quyết định việc thay đổi có nên được thực thi hay không. Khi họ đề xuất một sự thay đổi tới những yêu cầu của hệ thống, họ sẽ làm việc với đội ngũ những người làm việc để đánh giá tác động của sự thay đổi đó và quyết định sự thay đổi có nên được giữ ở những thành phần đã được lập kế hoạch cho sự phát triển tiếp theo của hệ thống. Tuy nhiên, những thay đổi liên quan tới phần mềm được nâng cấp được xem xét thận trọng ở những lập trình viên đang làm việc với hệ thống. Đưa trở lại nhà cung cấp, nơi phần mềm tiếp tục được cải thiện, không được xem như là một lần nâng cấp nhưng lại được xem như một phần cần thiết của quá trình phát triển.

Với những thành phần của phần mềm được thay đổi, một bản ghi của những sự thay đổi được tạo ra tới mỗi thành phần nên được lưu trữ. Nó đôi khi được gọi là *quá trình hình thành* (*derivation history*) của một thành phần. Một con đường tốt để lưu trữ *quá trình hình thành* là ở trong một chú giải đã được chuẩn hóa tại điểm bắt đầu của mã nguồn thành phần (*hình 10.5*). Chú giải này nên tham chiếu tới yêu cầu thay đổi cái đã tác động đến sự thay đổi của phần mềm. Sau đó bạn có thể viết một văn bản bao quát toàn bộ các thành phần và tiến trình để đưa ra những bản ghi về sự thay đổi của thành phần. Một cách tiếp cận tương tự có thể được sử dụng cho những trang Web. Với những tài liệu đã được xuất bản, những bản ghi về thay đổi được liên kết trong mỗi phiên bản thường được lưu trữ trong một trang riêng tại mặt trước của các tài liệu.

```
// Dự án BANKSEC (IST 6087)
//
// Công cụ BANKSEC/AUTH/RBAC/USER_ROLE
//
// Đối tượng: Role hiện thời
// Tác giả: N.Perwaiz
// Ngày tạo ra: 10th November 2002
//
// (c) Lancaster University 2002
//
// Quá trình chỉnh sửa
// Phiên bản      Người chỉnh sửa      Ngày      Thay đổi      Lý do
// 1.0            J.Jones            1/12/2002  Add header     Đề trình tới CM
// 1.1            N.Perwaiz          9/4/2003   New field      Thay đổi req. R07/02
```

Hình 10.5 Thông tin các thành phần đầu tiên

### 11.3. Quản lý phiên bản và bản phát hành

Những tiến trình được bao hàm trong quản lý phiên bản và bản phát hành được liên kết với việc chỉ định và theo dõi các phiên bản của hệ thống. Những người quản lý phiên bản đưa ra các thủ tục để chắc chắn rằng phiên bản của một hệ thống có thể được truy xuất khi được yêu cầu và không phải được thay đổi ngẫu nhiên bởi những người phát triển. Với những sản phẩm, những người quản lý phiên bản làm việc với nhân viên marketing và, với những hệ thống quen thuộc với các khách hàng, để lập kế hoạch khi những bản phát hành mới của hệ thống nên được tạo ra và phân phối cho sự triển khai.

Một phiên bản của hệ thống là một mặt của hệ thống, và theo một cách nào đó, để phân biệt với các bộ mặt khác. Những phiên bản của hệ thống có thể có nhiều chức năng khác nhau, tính thực thi được nâng cao hay những lỗi phần mềm đã được khắc phục. Một số phiên bản có thể có chức năng tương đương nhưng mà lại được thiết kế cho những cấu hình phần mềm hay phần cứng khác nhau. Những phiên bản mà sự khác nhau là nhỏ thì đôi khi được gọi là những *biến thể* (*variant*).

Một bản phát hành của hệ thống là một phiên bản được phân phối tới khách hàng. Mỗi bản phát hành của hệ thống nên chứa đựng một tính năng mới hoặc nên được hướng tới cho một nền tảng phần cứng khác. Thông thường thì một hệ thống có nhiều phiên bản hơn là những bản phát hành. Các phiên bản được tạo ra bên trong tổ chức để hướng tới mở rộng hoặc là kiểm tra nội bộ và không được hướng tới việc phân phối tới khách hàng.

Như sẽ thảo luận trong mục 10.5, công cụ **CASE** giờ đây luôn luôn được sử dụng để hỗ trợ cho việc quản lý phiên bản. Những công cụ này quản lý việc lưu trữ mỗi phiên bản của phần mềm và điều khiển truy cập tới các thành phần của hệ thống. Các thành phần phải được kiểm tra từ hệ thống để hiệu chỉnh. Ghi lại thành phần tạo ra một phiên bản mới, và một sự xác minh được chỉ định với hệ thống quản lý phiên bản. Trong khi rõ ràng các công cụ là khác nhau đáng kể trong các nét riêng và trong các giao diện người sử dụng của chúng, nguyên lý chung của quản lý phiên bản ở đây bao gồm những nét cơ bản cho mọi công cụ hỗ trợ.

#### 11.3.1. Xác minh phiên bản

Để tạo ra một phiên bản thực tế của một hệ thống, bạn phải định ra những phiên bản của các thành phần của hệ thống được bao gồm bên trong nó. Trong một hệ thống phần mềm lớn thì có hàng trăm các thành phần của phần mềm, mỗi một trong chúng có thể tồn tại trong một vài các phiên bản khác nhau. Do đó có thể có một cách rõ ràng để xác minh mỗi phiên bản thành phần để chắc chắn rằng thành phần chính xác được bao hàm bên trong hệ thống. Tuy nhiên, bạn không thể sử dụng tên của chỉ mục cấu hình cho việc xác minh phiên bản bởi vì có thể có vài phiên bản của mỗi chỉ mục cấu hình được xác minh.

Thay vì đó, ba kỹ thuật cơ bản được sử dụng cho việc xác minh phiên bản thành phần là:

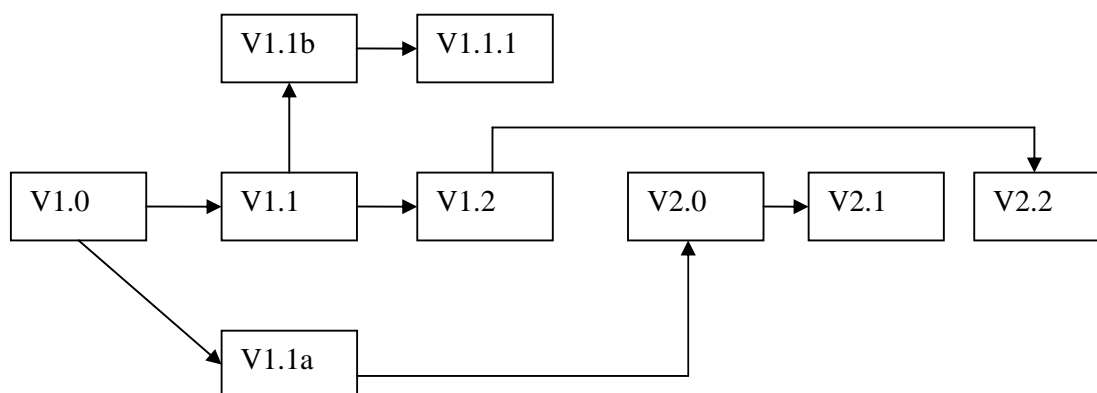
1. *Đánh số phiên bản* Thành phần được đánh dấu bởi một số phiên bản rõ ràng và đơn nhất. Đó là chiến lược xác minh được sử dụng chung nhất.
2. *Xác minh thuộc tính cơ bản* Mỗi thành phần có một tên ( như tên chỉ mục cấu hình, những cái không đơn nhất qua các phiên bản ) và một tập các thuộc tính có liên quan với nhau cho mỗi phiên bản (Estublier và Casallas, 1994). Từ đó các thành phần được xác minh bằng việc chỉ ra tên của chúng và các giá trị thuộc tính.

3. *Xác minh hướng thay đổi* Mỗi thành phần được đánh tên như trong xác minh các thuộc tính cơ bản nhưng điều đó cũng liên quan với một hoặc nhiều yêu cầu thay đổi (Munch, et al., 1993). Do đó nó được thừa nhận rằng mỗi phiên bản của thành phần đã được tạo ra trong sự tác động từ một hoặc nhiều hơn những yêu cầu thay đổi. Phiên bản của thành phần được xác minh bởi tập các yêu cầu thay đổi được đưa tới thành phần.

### 11.3.2. Đánh số phiên bản

Trong một chiến lược đánh số phiên bản đơn giản, một số của phiên bản được gắn với tên của thành phần hoặc là hệ thống. Từ đó, bạn có thể tham khảo tới Solaris 4.3 (phiên bản 4.3 của Solaris system) và phiên bản 1.4 của thành phần **getToken**. Nếu phiên bản đầu tiên được gọi là 1.0 thì theo trình tự sau các phiên bản sẽ là 1.1, 1.2, và vân vân. Tại một số chặng, một bản phát hành mới được tạo ra (bản 2.0) và tiến trình lại bắt đầu từ phiên bản 2.1. Chiến lược này là tuyến tính, cơ bản dựa trên giả định rằng các phiên bản của hệ thống được tạo ra tuần tự. Phần lớn các công cụ quản lý phiên bản (trong mục 9.5) như **RCS** (Tichy, 1985) và **CVS** (Berliner, 1990) hỗ trợ các tiếp cận này cho việc xác minh phiên bản.

Tôi minh họa cách tiếp cận này và sự hình thành của một số các phiên bản hệ của thống ở trên *hình 10.6*. Mũi tên ngang trong sơ đồ này bắt nguồn từ phiên bản gốc tới phiên bản được tạo ra từ phiên bản gốc. Chú ý rằng sự hình thành của các phiên bản không nhất thiết phải là tuyến tính và các phiên bản với các số phiên bản liên tiếp nhau có thể được đưa ra từ những đường nối khác nhau. Chẳng hạn, trong *hình 29.6* phiên bản 2.2 được tạo ra từ phiên bản 1.2 chứ không phải từ phiên bản 2.1. Về mặt nguyên tắc, bất cứ phiên bản đang tồn tại nào có thể được sử dụng như một điểm bắt đầu cho một phiên bản mới của hệ thống.



Hình 29.6 Cấu trúc sự hình thành các phiên bản

Chiến lược này là đơn giản, nhưng bạn cần lưu trữ lưu trữ một lượng lớn các thông tin mở rộng để theo dõi sự khác nhau giữa các phiên bản và mối quan hệ giữa các đề xuất thay đổi hệ thống và các phiên bản. Chẳng hạn đối với phiên bản 1.1 và 1.2 của một hệ thống có thể khác nhau bởi vì phiên bản 1.2 đã được sản xuất bởi việc sử dụng một thư viện đồ họa khác. Cái tên không nói cho bạn biết về phiên bản đó hoặc tại sao nó được tạo ra. Do đó bạn cần theo dõi các bản ghi trong cơ sở dữ liệu cấu hình nơi đã mô tả mỗi phiên bản tại sao nó được



tạo ra. Bạn cũng có thể cần liên kết rõ ràng những yêu cầu thay đổi tới những phiên bản khác nhau của mỗi thành phần.

### 11.3.3. Xác minh thuộc tính cơ bản

Một vấn đề cơ bản đối với chiến lược đánh tên rõ ràng các phiên bản là nó không phản ánh rất nhiều thuộc tính mà nó có thể được sử dụng để xác minh các phiên bản. Chẳng hạn những thuộc tính được xác minh là:

- Khách hàng
- Ngôn ngữ phát triển
- Tình trạng phát triển
- Nền tảng phần cứng
- Ngày tạo ra

Nếu mỗi phiên bản được xác minh bởi một tập các thuộc tính đơn nhất, thì dễ dàng thêm phiên bản mới được tìm thấy ở bất cứ phiên bản nào đang tồn tại. Chúng được xác minh có sử dụng một tập các giá trị thuộc tính đơn nhất. Chúng chia sẻ phần lớn những giá trị này với phiên bản cha mẹ của chúng từ đó các mối quan hệ giữa các phiên bản sẽ được duy trì. Bạn có thể nhận được những phiên bản được chỉ định bằng việc chỉ ra giá trị thuộc tính được yêu cầu. Những chức năng của thuộc tính hỗ trợ các thắc mắc như là *‘phiên bản gần đây nhất được tạo ra’* hoặc là *‘phiên bản được tạo ra giữa các ngày đã cho’*.

Chẳng hạn, phiên bản của hệ thống phần mềm **AC3D** được phát triển trên Java cho Window XP trong tháng giêng năm 2003 có thể được xác minh:

**AC3D** (ngôn ngữ = Java, nền = XP, ngày = Jan2003)

Để sử dụng một cách chỉ định tổng quan các thành phần trong **AC3D**, công cụ quản lý phiên bản sẽ lựa chọn những phiên bản của những thành phần có thuộc tính ‘Java’, ‘XP’ và ‘Jan2003’.

Xác minh thuộc tính cơ bản có thể được thực thi trực tiếp bởi hệ thống quản lý phiên bản, với các thuộc tính của các thành phần được lưu trữ trong cơ sở dữ liệu hệ thống. Như một sự lựa chọn, hệ thống xác minh thuộc tính có thể được xây dựng như một lớp hàng đầu của một chiến lược đánh số phiên bản được ấn dấu. Cơ sở dữ liệu cấu hình sẽ lưu trữ các mối liên kết giữa các thuộc tính xác minh và hệ thống bên trong và các phiên bản thành phần.

### 11.3.4. Xác minh hướng thay đổi

Việc xác minh các thuộc tính cơ bản của các phiên bản hệ thống đã giải tỏa một số vấn đề rắc rối của phương thức đánh số phiên bản đơn giản. Tuy nhiên, để truy xuất được một phiên bản bạn vẫn phải biết các thuộc tính liên quan của nó. Hơn thế nữa, bạn vẫn cần sử dụng một hệ thống quản lý thay đổi riêng để tìm ra mối quan hệ giữa các phiên bản và sự thay đổi.

Xác minh hướng thay đổi thường được sử dụng để xác định các phiên bản của hệ thống hơn là các thành phần. Những công cụ xác minh phiên bản của những thành phần riêng lẻ được ẩn dấu đối với người sử dụng của hệ thống CM. Mỗi sự thay đổi của hệ thống đã được thực thi có một tập các sự kiện thay đổi, nó mô tả những thay đổi được yêu cầu tới những thành phần hệ thống khác nhau. Tập thay đổi này có thể được đặt theo trình tự hay là ít nhất trong nguyên lý, phiên bản có thể kết hợp với một tập các thay đổi bất kỳ. Chẳng hạn, tập các thay đổi của hệ thống được tạo nên để làm cho nó thích hợp với Linux hơn là so với Solaris có thể được ứng dụng, tiếp bước bởi những thay đổi được yêu cầu về việc kết hợp một cơ sở dữ liệu mới. Cũng tương tự như vậy, sự thay đổi của Linux/Solaris có thể được làm tiếp theo bởi những thay đổi đòi hỏi chuyển giao diện người sử dụng từ tiếng Anh sang tiếng Italia.

Trên thực tế, tất nhiên, không phải là có thể yêu cầu một tập các thay đổi tùy ý tới hệ thống. Những tập thay đổi có thể là không thích hợp đến nỗi chẳng hạn tập thay đổi A tiếp theo tập thay đổi D có thể tạo ra hệ thống không có giá trị. Hơn thế nữa, những tập thay đổi có thể xung đột với nhau trong đó những thay đổi khác nhau tác động tới cùng một mã của hệ thống. Nếu mã đã bị thay đổi bởi tập thay đổi A, thì tập thay đổi D có thể không làm việc nữa. Để đánh dấu những khó khăn này, các công cụ quản lý phiên bản hỗ trợ việc xác minh hướng thay đổi cho phép thực hiện những nguyên tắc chặt chẽ của hệ thống nhằm xác minh. Những điều này hạn chế các cách kết hợp các tập thay đổi.

#### 11.4. Quản lý bản phát hành

Một bản phát hành của hệ thống là một phiên bản của hệ thống đã được phân phối tới các khách hàng. Những người quản lý bản phát hành của hệ thống có trách nhiệm trong việc quyết định khi nào hệ thống có thể được phân phối tới khách hàng, trong quản lý quá trình tạo ra bản phát hành và phương tiện truyền thông cho phân phối, và việc tài liệu hóa các bản phát hành để chắc chắn rằng nó có thể được tạo lại chính xác như đã được phân phối nếu điều đó là cần thiết.

Một bản phát hành không phải chính là những mã có thể thực hiện của hệ thống. Những bản này có thể bao gồm:

1. *Các file cấu hình* định nghĩa bản phát hành nên được định dạng như thế nào cho việc cài đặt thực tế.
2. *Các file dữ liệu* cần thiết cho hệ điều hành thành công
3. *Một chương trình cài đặt* được sử dụng nhằm cài đặt hệ thống lên phần cứng chỉ ra
4. *Tài liệu giấy và tài liệu điện tử* mô tả hệ thống
5. *Đóng gói và liên kết quảng cáo* đã được thiết kế cho việc phát hành.

Những người quản lý bản phát hành không thừa nhận rằng khách hàng luôn luôn cài đặt những bản mới. Một số người sử dụng hệ thống có thể thích với những hệ thống hiện hành. Họ có thể xem việc thay đổi ở một hệ thống mới là không có giá trị. Những bản phát hành mới của hệ thống không thể dựa trên việc cài đặt của bản trước. Để minh họa vấn đề này ta xem xét các mục sau:

1. Bản 1 của hệ thống được phân phối và đưa vào sử dụng
2. Bản 2 đòi hỏi việc cài đặt của những file dữ liệu mới, nhưng một số khách hàng không cần những tiện ích của bản 2 khi trải qua với bản 1.
3. Bản 3 đòi hỏi những file dữ liệu đã được cài đặt ở bản 2 và nó lại không có file dữ liệu mới

Người phát hành phần mềm không thể thừa nhận rằng những file được yêu cầu cho bản 3 đã được cài đặt trong mọi trường hợp. Một số trường hợp có thể mang trực tiếp từ bản 1 tới bản 3, bỏ qua bản 2. Một số có thể được chỉnh sửa file dữ liệu liên kết với bản 2 nhằm phản ánh các sự kiện riêng. Từ đó, các file dữ liệu phải được phát hành và cài đặt với bản 3 của hệ thống.

### **Ra quyết định phát hành**

Chuẩn bị và phân phối một bản phát hành của hệ thống là một quá trình đắt, thực tế là cho những sản phẩm phần mềm mang tính thị trường. Nếu những bản phát hành được đưa ra quá thường xuyên, các khách hàng có thể không nâng cấp lên phiên bản mới, đặc biệt là nếu nó không free. Nếu các bản phát hành hệ thống không được đưa ra thường xuyên, thì phần có thể bị mất khi mà khách hàng chuyển hướng tới những hệ thống khác. Tất nhiên, điều đó không đúng cho phần mềm quen thuộc được mở rộng đặc biệt cho một tổ chức. Đối với những phần mềm truyền thống, những bản phát hành không thường xuyên có thể làm tăng sự khác biệt giữa phần mềm và các tiến trình thương mại mà nó được thiết kế để hỗ trợ.

Những nhân tố kĩ thuật và tổ chức khác nhau mà bạn đưa vào trong danh mục khi quyết định tạo một bản phát hành mới của hệ thống được chỉ ra trên *hình 10.7*.

### **Tạo bản phát hành**

Tạo bản phát hành là quá trình thu thập các file và những tài liệu bao gồm tất cả các thành phần của bản phát hành của hệ thống. Mã thực hiện hoặc là những chương trình và các file dữ liệu liên quan phải được thu thập và xác minh. Việc mô tả cấu hình có thể phải được viết cho những phần cứng và hệ điều hành khác nhau đồng thời mô tả thông qua các chỉ dẫn đã được chuẩn bị cho những khách hàng cần thiết lập cấu hình cho chính hệ thống của họ. Nếu những máy đọc được thu

Nhân tố	Mô tả
<b>Chất lượng của hệ thống</b>	Nếu những lỗi nghiêm trọng của hệ thống được thông báo có tác động tới cách sử dụng của nhiều khách hàng, nó có thể cần thiết để đưa ra bản đã được sửa chữa lỗi. Tuy nhiên, với những lỗi hệ thống nhỏ có thể được sửa chữa bằng việc đưa ra những cách chấp vá (thường được cung cấp qua Internet) có thể được áp dụng cho bản hiện hành của hệ thống.
<b>Những thay đổi về nền</b>	Bạn có thể phải tạo ra một bản mới của phần mềm ứng dụng khi một phiên bản mới của hệ điều hành nền được phát hành.
<b>Định luật Ledman thứ năm</b>	Điều này gợi ý rằng việc tăng cường chức năng ở mỗi bản phát hành ước chừng là một hằng số. Từ đó, một bản phát hành mới của hệ thống với chức năng quan trọng có thể được tạo ra tiếp theo việc khắc phục một bản trước.
<b>Cạnh tranh</b>	Một bản phát hành mới của hệ thống có thể là cần thiết vì một sản phẩm cạnh tranh
<b>Yêu cầu marketing</b>	Bộ phận marketing của một tổ chức có thể tận tụy với các bản phát hành tại một thời điểm cụ thể.
<b>Đề xuất thay đổi của khách hàng</b>	Với những hệ thống theo yêu cầu, khách hàng có thể đưa ra một tập những đề xuất cho thay đổi hệ thống, và họ mong đợi một bản phát hành của hệ thống ngay sau khi điều đó được thực thi.

Hình 10.7 Những nhân tố ảnh hưởng tới bản phát hành của hệ thống

Phương tiện phân phối thông thường cho những bản phát hành của hệ thống ngày nay thường là đĩa quang (CD-ROM hoặc là DVD) là những đĩa có khả năng chứa từ 600 Mbyte đến 6Gbyte dữ liệu. Trong trường hợp khác phần mềm có thể được phân phối trực tuyến, cho phép khách hàng download nó từ Internet, mặc dù nhiều người thấy nó quá lâu đối với những file lớn và họ thích cách phân phối bằng đĩa CD hơn.

Có những chi phí rất cao cho việc marketing và đóng gói liên quan đến việc phân phối các sản phẩm phần mềm mới, do đó nhà cung cấp sản phẩm thường tạo ra những bản mới chỉ khi nào dành cho những phần mềm nền mới hay là để thêm vào một số chức năng đặc biệt. Rồi họ tính giá tới người sử dụng. Khi có những vấn đề được khám phá ở một bản hiện hành, nhà sản xuất thường đưa ra những cách chấp vá trên website và khách hàng có thể download để sửa chữa phần mềm.

Một phần từ chi phí cho việc tìm kiếm và download các bản phát hành mới, vấn đề là nhiều khách hàng có thể không bao giờ khám phá ra sự tồn tại của những cách sửa chữa này hoặc là không có sự hiểu biết kỹ thuật để cài đặt chúng. Từ đó họ có thể tiếp tục sử dụng hệ thống lỗi mà họ đang có nhưng với một mức độ rủi ro rất yếu đối với công việc của họ. Trong một số trường hợp, nơi mà miếng vá được thiết kế để sửa chữa những lỗ hổng an ninh, sự mạo hiểm khi cài đặt miếng vá có thể có nghĩa là công việc rất dễ bị tấn công từ bên ngoài.

### Tài liệu bản phát hành

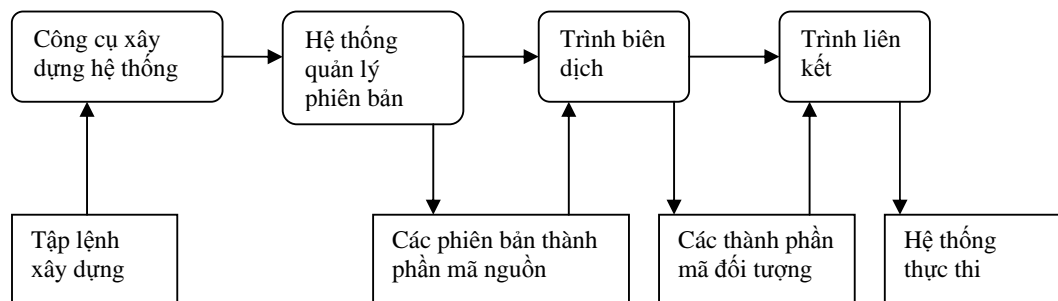
Khi một bản phát hành của hệ thống được sản xuất, nó phải được tài liệu hóa để chắc chắn rằng nó có thể được tạo lại chính xác như thế trong tương lai. Đó là điều rất quan trọng cho những hệ thống những có thời gian sống lâu và được tạo ra theo yêu cầu của khách hàng như việc điều khiển những máy móc phức tạp. Khách hàng có thể sử dụng một bản phát hành riêng lẻ của những hệ thống này lâu dài sau khi bản gốc được phát hành.

Để tài liệu hóa một bản phát hành, bạn phải ghi lại phiên bản chỉ ra của thành phần mã nguồn được sử dụng để tạo ra mã thực thi. Bạn phải giữ bản sao của nguồn và mã thực thi, mọi dữ liệu và các file cấu hình. Bạn cũng nên ghi lại các phiên bản của hệ điều hành, các thư viện, chương trình dịch, và các công cụ khác được sử dụng để xây dựng hệ thống. Những điều này có thể được yêu cầu để xây dựng chính xác hệ thống như thế này sau một thời gian. Điều đó có nghĩa là bạn phải lưu trữ những bản sao của phần mềm nền và các công cụ được sử dụng để tạo ra hệ thống trong hệ thống quản lý phiên bản cùng với mã nguồn của hệ thống.

### 11.5. Xây dựng hệ thống

Xây dựng hệ thống là một quá trình biên tập và liên kết các thành phần phần mềm vào một chương trình để thực thi trên một cấu hình mục tiêu thực tế đã được xác định. Khi bạn xây dựng một hệ thống từ các thành phần của nó, bạn phải lưu ý những câu hỏi sau đây:

1. Tất cả các thành phần tạo nên hệ thống đã được bao hàm trong các chỉ thị xây dựng hay chưa?
2. Các phiên bản tương thích của mỗi thành phần yêu cầu đã được bao hàm trong các chỉ thị xây dựng hay chưa?
3. Tất cả các file dữ liệu yêu cầu là đã sẵn sàng hay không?
4. Nếu các file dữ liệu là được thao chiếu bên trong một thành phần, thì tên đã được sử dụng có giống với tên của file dữ liệu bên trên chiếc máy mục tiêu hay không.
5. Các phiên bản tương thích của trình biên dịch và các công cụ yêu cầu khác là có hiệu lực hay không? Phiên bản hiện hành của các công cụ phần mềm có thể không tương thích với những phiên bản cũ được sử dụng để phát triển hệ thống.



Hình 10.8 Xây dựng hệ thống

Ngày nay, những công cụ quản trị cấu hình phần mềm hoặc là môi trường lập trình được sử dụng để tự động hóa các tiên trình xây dựng hệ thống. Nhóm **CM** viết một tập lệnh xây

dựng để định nghĩa sự phụ thuộc giữa các thành phần hệ thống. Tập lệnh này cũng định nghĩa các công cụ để biên tập và liên kết các thành phần hệ thống. Công cụ xây dựng hệ thống thông dịch những tập lệnh xây dựng này và gọi những chương trình khác được yêu cầu để xây dựng hệ thống thực thi từ các thành phần của nó. Điều này được minh họa trên *hình 10.8*. Trong một số môi trường lập trình (như là môi trường Java mở rộng), các tập lệnh xây dựng được tạo ra tự động bằng việc phân tích mã nguồn và nhận ra các thành phần được gọi. Tất nhiên, trong trường hợp này, tên của thành phần được lưu trữ phải trùng với tên của thành phần chương trình.

Sự phụ thuộc giữa các thành phần được xác định trong các tập lệnh xây dựng. Nó cung cấp thông tin mà công cụ xây dựng hệ thống có thể quyết định khi nào mã nguồn của các thành phần phải được biên dịch lại và khi mã đối tượng hiện hành có thể được tái sử dụng. Trong rất nhiều công cụ, những tập lệnh phụ thuộc này thường được xem như sự phụ thuộc giữa các file vật lý trong đó mã nguồn và các thành phần mã đối tượng được lưu trữ. Tuy nhiên, khi đây là những file đa mã nguồn biểu diễn nhiều thành phần của phiên bản, có thể rất khó để nói file nguồn nào đã được sử dụng để nhận lấy các thành phần mã đối tượng. Những rắc rối này giống tương đương như khi giữa nguồn và những file mã đối tượng có cùng một tên nhưng khác đuôi. Vấn đề này có thể chỉ được giải quyết khi việc quản lý phiên bản và các công cụ xây dựng hệ thống được tích hợp.

### 11.6. Các công cụ CASE cho quản trị cấu hình

Những tiến trình quản trị cấu hình thường được chuẩn hóa và liên quan đến ứng dụng của những thủ tục đã được xác định trước. Chúng đòi hỏi sự quản trị cẩn mật một số lượng rất lớn dữ liệu và hướng tới những chi tiết chủ yếu. Khi một hệ thống đang được xây dựng từ những phiên bản thành phần, một lỗi đơn giản trong quản trị cấu hình có thể có nghĩa là phần mềm sẽ không hoạt động đúng. Từ đó, việc hỗ trợ công cụ **CASE** là thiết yếu cho quản trị cấu hình, và từ những năm 70 nhiều công cụ phần mềm bao gồm nhiều phạm vi khác nhau của quản trị cấu hình đã được đưa ra.

Những công cụ này có thể được kết hợp để tạo một mô hình làm việc của quản trị cấu hình để hỗ trợ mọi hoạt động quản trị cấu hình. Có hai loại của mô hình làm việc **CM**:

1. *Mô hình làm việc mở* Các công cụ cho mỗi chặng của quá trình quản trị cấu hình được tích hợp qua những thủ tục được tổ chức chuẩn cho việc sử dụng những công cụ này. Có rất nhiều công cụ **CM** nguồn mở và mang tính thương mại có giá trị cho những mục đích được chỉ ra. Việc quản lý thay đổi có thể được hỗ trợ những công cụ theo dõi lỗi như là Bugzilla, quản lý phiên bản bằng việc sử dụng những công cụ như là **make** (Feldman, 1979; Oram và Talbott, 1991) hoặc **imake** (Dubois, 1996). Chúng đều là những công cụ nguồn mở và đều có giá trị miễn phí.
2. *Mô hình làm việc tích hợp* Những mô hình này cung cấp những khả năng tích hợp cho quản lý phiên bản, xây dựng hệ thống và kiểm tra thay đổi. Chẳng hạn, quá trình Ration's Unified Change Management dựa trên một mô hình làm việc **CM** tích hợp kết hợp với ClearCASE (White, 2000) cho xây dựng hệ thống và quản lý phiên bản và ClearQuest cho theo dõi thay đổi. Sự tiện lợi của mô hình làm việc **CM** tích hợp là những thay đổi dữ liệu đơn giản, và mô hình này bao gồm cơ sở dữ liệu **CM** tích hợp. Mô hình **SCM** tích hợp đã được nhận lấy từ những hệ thống sớm hơn như

Lifespan (Whitgift, 1991) cho quản lý thay đổi và DSEE (Leblang và Chase, 1987) cho quản lý phiên bản và xây dựng hệ thống. Tuy nhiên, mô hình **CM** tích hợp rất phức tạp và đắt, nên nhiều tổ chức vẫn thích sử dụng những công cụ hỗ trợ rẻ tiền và đơn giản khác hơn.

Rất nhiều hệ thống lớn được phát triển ở những nơi khác nhau, và chúng cần những công cụ **SCM** hỗ trợ cho làm việc đa vị trí với nhiều nơi lưu trữ dữ liệu cho các chỉ mục cấu hình. Trong khi phần lớn các công cụ **SCM** được thiết kế cho làm việc đơn vị trí, một số công cụ, như **CVS**, có những chức năng cho hỗ trợ đa vị trí (Vesperman, 2003).

#### 11.6.1. Hỗ trợ cho quản lý thay đổi

Mỗi cá nhân liên quan trong quá trình quản lý thay đổi có trách nhiệm cho một số hoạt động. Họ hoàn thành hoạt động này, và đưa những mô hình chỉ mục cấu hình liên quan tới một số người khác. Thủ tục tự nhiên của quá trình này có nghĩa là một mô hình về quá trình thay đổi có thể được thiết kế và tích hợp với một hệ thống quản lý phiên bản. Mô hình này có thể được thông dịch ra những tài liệu để đưa tới đúng người tại đúng thời điểm.

Có một vài công cụ quản lý thay đổi có hiệu lực, từ những công cụ nguồn mở tương đối đơn giản như Bugzilla đến những hệ thống tích hợp toàn diện như Rational ClearQuest. Những công cụ này cung cấp một số hoặc là tất cả những chức năng sau để hỗ trợ tiến trình.

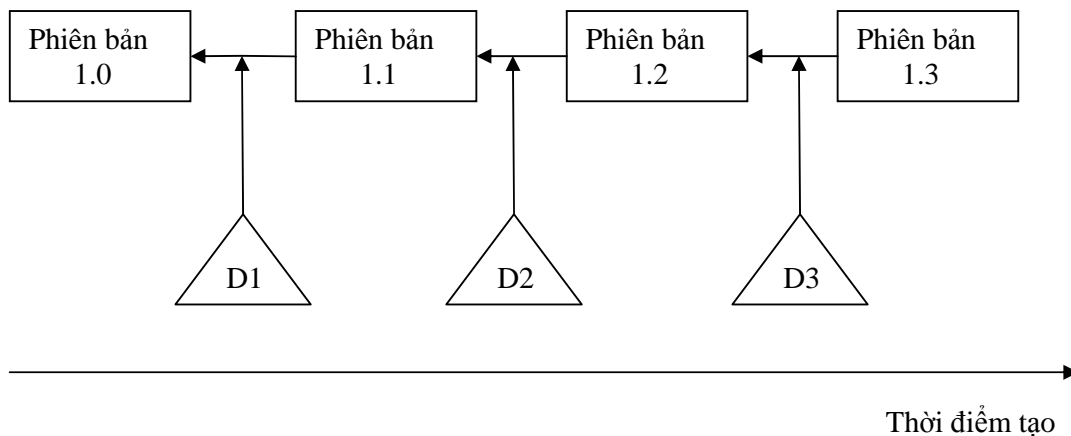
1. *Một dạng biên tập* cho phép những dạng đề xuất thay đổi được tạo ra và hoàn thành bởi những người đưa ra yêu cầu thay đổi.
2. *Một hệ thống dòng làm việc* cho phép nhóm **CM** định nghĩa người phải soạn thảo dạng yêu cầu thay đổi và trình tự của tiến trình. Hệ thống này sẽ tự động đưa khuôn dạng này tới đúng người tại đúng thời điểm và thông báo tới những nhóm thành viên của quá trình đi lên của thay đổi. Thư điện tử được sử dụng để cung cấp những sự nâng cấp cho những điều liên quan này bên trong quá trình.
3. *Một cơ sở dữ liệu thay đổi* được sử dụng để quản lý mọi đề xuất thay đổi và có thể được liên kết tới một hệ thống quản lý phiên bản. Những chức năng truy vấn dữ liệu cho phép nhóm **CM** tìm ra những đề xuất thay đổi được chỉ định.
4. *Một hệ thống thông báo thay đổi* điều hành những ghi chép quản lý trên trạng thái của những yêu cầu thay đổi đã được đệ trình.

#### 11.6.2. Hỗ trợ cho quản lý phiên bản

Quản lý phiên bản liên quan tới quản lý một lượng lớn thông tin và chắc chắn rằng những thay đổi của hệ thống là được ghi lại và theo dõi. Những công cụ quản lý phiên bản điều hành một nơi chứa những chỉ mục cấu hình nơi mà nội dung của nó là không thay đổi. Để làm việc trên một chỉ mục cấu hình bạn phải lấy nó ra từ nơi chứa và để nó vào một thư mục làm việc. Sau khi tạo sự thay đổi tới phần mềm, bạn đưa nó trở lại vào trong mục chứa và một phiên bản mới được tự động tạo thành.

Mọi hệ thống quản lý phiên bản đều cung cấp một tập các khả năng cơ bản có thể so sánh mặc dù một số có những chức năng phức tạp hơn những thứ khác. Ví dụ về những chức năng này như là:

1. *Xác minh phiên bản và bản phát hành* Phiên bản được quản lý được gắn với việc xác minh khi chúng được xem xét tới hệ thống. Những hệ thống khác nhau hỗ trợ những cách xác minh phiên bản khác nhau như đã được xem xét ở mục 29.3.1
2. *Quản lý lưu trữ* Để giảm không gian lưu trữ đòi hỏi bởi nhiều phiên bản, hệ thống quản lý cung cấp những chức năng quản lý không gian mà ở đó các phiên bản được mô tả bởi sự khác nhau của chúng với một số phiên bản chính. Sự khác nhau giữa các phiên bản được biểu diễn như *delta*, cái được tóm lược trong những chỉ thị được yêu cầu để tái tạo phiên bản của hệ thống liên quan. Điều này được minh họa trên *hình 10.9*, nó chỉ ra làm thế nào việc khôi phục *delta* có thể được xem xét phiên bản gần nhất của tới việc tái tạo những phiên bản sớm hơn của hệ thống. Phiên bản gần nhất là 1.3. Để tạo phiên bản 1.2 bạn xem xét việc thay đổi *delta* để tái tạo lại phiên bản đó.



Hình 10.9 Đánh dấu phiên bản

3. *Thông báo quá trình thay đổi* Mọi thay đổi tác động tới mã của hệ thống hoặc là các thành phần được ghi lại và đưa vào sổ sách. Trong một số hệ thống, những thay đổi này có thể được sử dụng để chọn một phiên bản thực tế của hệ thống.
4. *Phát triển độc lập* Nhiều phiên bản của hệ thống có thể được phát triển song song và mỗi phiên bản có thể được thay đổi độc lập. Chẳng hạn, bản phát hành 1 có thể được chỉnh sửa sau khi phát triển bản 2 bằng việc cộng thêm một cấp độ *deltas*. Hệ thống quản lý phiên bản theo dõi các thành phần đã được kiểm tra cho việc điều chỉnh và chắc chắn rằng những thay đổi tác động tới cùng một thành phần bởi những người phát triển khác nhau không xung đột. Một số hệ thống cho phép chỉ một mẫu của một thành phần được kiểm tra cho việc chỉnh sửa; một số khác giải quyết những xung đột tiềm ẩn khi những thành phần đã chỉnh sửa được đưa trở lại hệ thống.

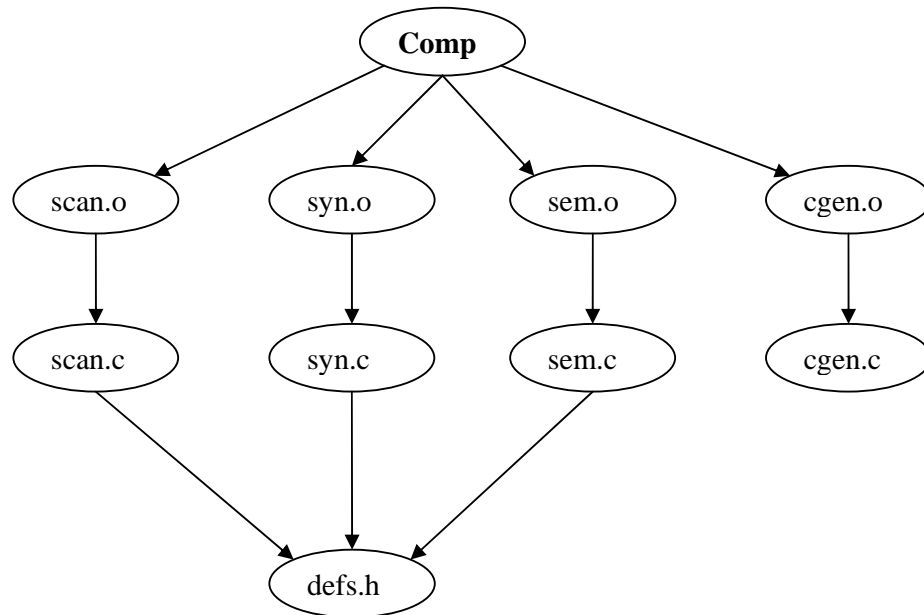


5. *Hỗ trợ dự án* Hệ thống có thể hỗ trợ nhiều dự án như là nhiều file. Trong hệ thống hỗ trợ dự án, như **CVS**, có thể kiểm tra gỡ rối mọi file liên quan với một dự án hơn là một file tại một thời điểm.

### 11.6.3. Hỗ trợ xây dựng hệ thống

Xây dựng hệ thống là một tiến trình chuyên sâu máy tính. Việc biên tập và liên kết tất cả mọi thành phần của một hệ thống lớn có thể mất vài giờ đồng hồ. Có thể có hàng trăm file liên quan, với với những khả năng theo đó là lỗi của con người nếu chúng được biên tập và liên kết thủ công. Những công cụ xây dựng hệ thống tự động xây dựng tiến trình giảm những lỗi tiềm tàng do con người gây ra, và tối thiểu hóa thời gian đòi hỏi xây dựng hệ thống.

Các công cụ xây dựng hệ thống có thể đứng độc lập, như việc hình thành Unix đưa ra những tiện ích (Oram và Talbott, 1991), hoặc có thể được kết hợp với những công cụ quản lý phiên bản. Các chức năng cung cấp bởi những công cụ xây dựng hệ thống **CASE** có thể là :



Hình 10.10 Các thành phần phụ thuộc

1. *Một ngôn ngữ có sự phụ thuộc có hiệu quả và trình thông dịch tương ứng.* Những thành phần phụ thuộc vào nhau có thể được mô tả và tối thiểu việc biên dịch lại. Tôi sẽ giải thích điều này chi tiết hơn sau đây.
2. *Công cụ hỗ trợ việc cài đặt và lựa chọn* Trình biên dịch và những công cụ soạn thảo khác được sử dụng để xử lý những file mã nguồn có thể được chỉ định và khởi tạo như yêu cầu.
3. *Sự biên dịch được phân bổ* Một số công cụ xây dựng hệ thống, đặc biệt chúng là một phần của hệ thống **CM** tích hợp, hỗ trợ biên dịch được phân phối trong mạng máy tính. So với mọi sự biên dịch dựa trên việc thực thi trên một máy đơn, những công cụ xây dựng hệ thống tìm kiếm những bộ xử lý nhàn rỗi trong mạng và truyền sao chép một số các trình biên dịch

song song. Điều này rất quan trọng trong việc giảm thời gian đòi hỏi để xây dựng hệ thống.

4. *Quản lý đối tượng dẫn xuất* Những đối tượng dẫn xuất là những đối tượng được tạo ra từ những đối tượng nguồn khác. Quản lý đối tượng dẫn xuất sẽ liên kết mã nguồn và đối tượng dẫn xuất và dẫn xuất lại chỉ một đối tượng khi nó được yêu cầu bởi việc thay đổi mã nguồn.

Quản lý đối tượng dẫn xuất và tối thiểu hóa việc biên dịch lại được giải thích tốt nhất bằng việc sử dụng một ví dụ đơn giản. Xem xét một trình biên dịch được gọi là **Comp** được tạo ra hơn bốn module đối tượng có tên là scan.o, syn.o, sem.o, và cgen.o. Mỗi module đối tượng được tạo ra từ một module mã nguồn với tên tương ứng (scan.c, syn.c, sem.c và cgen.c). Một file của biến và những khai báo hằng được gọi là defs.h được chia sẻ bởi scan.c, syn.c, sem.c (hình 10.10). Trong hình 10.10, mũi tên có nghĩa là “phụ thuộc vào”\_thực thể tại đuôi mũi tên phụ thuộc và thực thể tại đầu mũi tên. Từ đó, **Comp** phụ thuộc vào scan.o, syn.o, sem.o và cgen.o, scan.o phụ thuộc vào scan.c ... vân vân.

Nếu scan.c thay đổi, công cụ xây dựng hệ thống, công cụ xây dựng hệ thống có thể phát hiện thấy rằng đối tượng dẫn xuất scan.o cần được tạo lại. Nó làm điều này bằng việc so sánh thời điểm hiệu chỉnh của scan.o và scan.c và nhận thấy rằng scan.c được hiệu chỉnh sau scan.o. Sau đó nó sẽ gọi trình biên dịch C để biên dịch scan.c để tạo một đối tượng dẫn xuất mới là scan.o.

Công cụ xây dựng sau đó sẽ sử dụng những liên kết phụ thuộc giữa **Comp** và scan.o để nhận thấy rằng **Comp** cũng cần được tái tạo bằng việc liên kết scan.o, syn.o, sem.o và cgen.o. Hệ thống có thể nhận ra rằng những thành phần mã đối tượng khác là không thay đổi, do đó việc biên dịch lại mã nguồn của chúng là không cần thiết.

Phần lớn các công cụ xây dựng hệ thống sử dụng file ngày chỉnh sửa như một khóa thuộc tính trong việc quyết định khi nào việc biên dịch lại được yêu cầu. Nếu một file mã nguồn được chỉnh sửa sau file mã đối tượng tương ứng, thì file mã đối tượng đó phải được tái tạo. Về cơ bản, có thể chỉ có một phiên bản của mã đối tượng tương ứng với thành phần mã nguồn được thay đổi gần đây nhất. Khi một phiên bản mới của thành phần mã nguồn được tái tạo, mã đối tượng của phiên bản trước sẽ bị mất.

Tuy nhiên, một số công cụ sử dụng một các tiếp cận tinh vi hơn cho việc quản lý đối tượng dẫn xuất. Họ đánh thẻ địa chỉ đối tượng dẫn xuất với phiên bản được xác minh của mã nguồn được sử dụng để tạo ra đối tượng. Trong giới hạn của dung lượng lưu trữ, họ lưu giữ mọi đối tượng dẫn xuất. Bởi thế, nó thường có khả năng khám phá ra mã đối tượng của mọi phiên bản của các thành phần mã nguồn mà không cần biên dịch lại.

### Những điểm quan trọng

- Quản trị cấu hình là việc quản lý những thay đổi của hệ thống. Khi một hệ thống được lưu giữ, vai trò của nhóm **CM** là đảm bảo rằng những thay đổi là được phối hợp trong phạm vi có kiểm soát.
- Trong những dự án lớn, một kế hoạch đánh tên tài liệu chính thức nên được thiết lập và sử dụng như một điều cơ bản cho việc theo dõi những phiên bản của mọi tài liệu dự án.

- Nhóm **CM** nên được hỗ trợ bởi một cơ sở dữ liệu cấu hình có ghi chép thông tin về những thay đổi của hệ thống và những yêu cầu thay đổi. Dự án nên có một số điều kiện chính thức của yêu cầu thay đổi hệ thống.
- Khi sắp đặt một chiến lược quản trị cấu hình, một chiến lược xác minh phiên bản thích ứng nên được thiết lập. Các phiên bản có thể được xác định bởi số, bởi một tập các thuộc tính liên quan hoặc bởi những thay đổi hệ thống được đề xuất mà nó thực thi.
- Những bản phát hành của hệ thống bao gồm mã thực thi, file dữ liệu, file cấu hình và các tài liệu. Quản lý bản phát hành bao gồm ra quyết định ngày phát hành, chuẩn bị mọi thông tin cho phân phối và tài liệu hóa mỗi bản phát hành của hệ thống.
- Xây dựng hệ thống là một quá trình kết hợp các thành phần của hệ thống vào trong chương trình thực thi để chạy trên một số hệ thống máy tính mục tiêu.
- Những công cụ **CASE** có giá trị cho việc hỗ trợ mọi hoạt động quản trị cấu hình. Chúng bao gồm những công cụ như **CVS** để quản lý phiên bản hệ thống, những công cụ cho quản lý thay đổi và những công cụ cho việc xây dựng hệ thống.
- Những công cụ **CASE** cho **CM** có thể là những công cụ độc lập hỗ trợ quản lý thay đổi, quản lý phiên bản và xây dựng hệ thống, hoặc có thể là những mô hình làm việc được tích hợp nhằm cung cấp một giao diện chung cho mọi hỗ trợ cho **CM**.

## Bài tập

1. Giải thích tại sao bạn không nên sử dụng tiêu đề của một tài liệu để xác định các tài liệu trong hệ thống quản trị cấu hình. Gợi ý rằng một tiêu chuẩn cho chiến lược xác định tài liệu có thể được sử dụng cho mọi dự án trong một tổ chức.
2. Sử dụng một cách tiếp cận hướng đối tượng (trong chương 8), thiết kế một mô hình cơ sở dữ liệu cấu hình có ghi lại thông tin về các thành phần hệ thống, các phiên bản, các bản phát hành và sự thay đổi. Một số yêu cầu cho mô hình này như sau:
  - Nó có khả năng truy xuất mọi phiên bản hoặc là một phiên bản đơn được chỉ ra của một thành phần.
  - Nó có khả năng truy xuất tới phiên bản gần nhất của thành phần
  - Nó có khả năng tìm kiếm những yêu cầu thay đổi nào đã được thực thi bởi một phiên bản thực tế của hệ thống.
  - Nó có khả năng khám phá ra phiên bản nào của các thành phần được bao gồm trong một phiên bản của hệ thống được chỉ định.
  - Nó có khả năng truy xuất tới một bản phát hành thực tế của hệ thống dựa theo ngày hoặc là theo khách hàng được phân phối.
3. Sử dụng một sơ đồ luồng dữ liệu, mô tả một thủ tục quản lý thay đổi có thể được sử dụng trong một tổ chức lớn có liên quan với việc phát triển phần mềm

cho những khách hàng mở rộng. Những thay đổi có thể được đề xuất hoặc là từ bên ngoài hoặc là từ bên trong.

4. Tại sao một hệ thống quản trị cấu hình dự án cơ sở như **CVS** lại làm đơn giản hóa tiến trình quản lý phiên bản.
5. Giải thích tại sao một hệ thống xác minh phiên bản dựa trên các thuộc tính cơ bản làm cho nó dễ dàng phát hiện ra mọi thành phần tạo nên phiên bản đồ của hệ thống.
6. Mô tả những khó khăn có thể tăng lên khi xây dựng một hệ thống từ những thành phần của nó. Vấn đề thực tế gì có thể xảy ra khi một hệ thống được xây dựng trên một máy tính chủ cho một số máy móc mục tiêu khác.
7. Với tham chiếu tới việc xây dựng hệ thống, giải thích tại sao bạn có thể đôi khi phải lưu trữ những máy tính cũ trên đó những hệ thống phần mềm đã được mở rộng.
8. Một vấn đề chung với xây dựng hệ thống nảy sinh khi tên những file vật lý được kết hợp trong mã hệ thống và cấu trúc file ngụ ý những tên này phân biệt với nó ở máy móc mục tiêu. Viết một tập hướng dẫn cho người lập trình tránh khỏi vấn đề này và những vấn đề xây dựng hệ thống khác mà bạn nghĩ có thể xảy ra.
9. Mô tả năm nhân tố nên được đem vào trong tài khoản bởi những kỹ sư trong suốt quá trình xây dựng một bản phát hành của hệ thống phần mềm lớn.
10. Mô tả hai cách trong đó những công cụ xây dựng hệ thống có thể trông mong vào quá trình xây dựng một phiên bản của hệ thống từ những thành phần của nó.

## PHỤ LỤC- CÁC CÂU HỎI ÔN TẬP

### 1. Chất lượng và đảm bảo chất lượng phần mềm

#### 1.1. Khái niệm về đảm bảo chất lượng

1. Chất lượng của một sản phẩm được sản xuất là gì? Đối với phần mềm, định nghĩa đó có đúng không? Làm thế nào để áp dụng định nghĩa đó?
2. Cái gì được dùng làm cơ sở để kiểm định chất lượng phần mềm?
3. Để làm cơ sở cho việc kiểm định chất lượng, đặc tả yêu cầu phần mềm cần thỏa mãn điều kiện gì? Nêu một vài ví dụ về điều kiện đưa ra?
4. Các nhân tố ảnh hưởng lên chất lượng phần mềm có mấy mức độ? Những loại nhân tố nào ảnh hưởng đến chất lượng?
5. Nêu các đặc trưng ảnh hưởng lên chất lượng của mỗi loại nhân tố (*đặc trưng chức năng, khả năng thích nghi với thay đổi, khả năng thích nghi với môi trường*) ?
6. Có thể đo trực tiếp chất lượng phần mềm không? Tại sao? Vậy phải đo bằng cách nào?
7. Kể ra các độ đo đặc trưng chất lượng chính của McCall và giải thích nội dung của nó?
8. Giải thích nội dung các thuộc tính chất lượng phần mềm sau đây và nêu ra các độ đo liên quan được sử dụng để đo thuộc tính đó:

*Tính đúng đắn, Tính tin cậy được, Tính hiệu quả, Tính toàn vẹn, Tính khả dụng, Tính bảo trì được, Tính mềm dẻo, Tính thử nghiệm được, Tính khả chuyển, Tính liên tác được?*

9. Nêu các đặc trưng chất lượng theo Hawlett? Giải thích nội dung mỗi loại?

#### 1.2. Tiến hóa của hoạt động đảm bảo chất lượng

10. Đảm bảo chất lượng phần mềm xuất phát từ đâu? Tiến triển của nó như thế nào?
11. Tại sao cần đảm bảo chất lượng phần mềm? Nó đóng vai trò gì trong một doanh nghiệp phát triển phần mềm?
12. Khi nào cần thực hiện các hoạt động đảm bảo chất lượng phần mềm?
13. Trong một tổ chức, những ai tham gia vào hoạt động đảm bảo chất lượng? Vai trò và trách nhiệm của mỗi đối tượng đó là gì?
14. Mục tiêu của SQA là gì? Các hoạt động chính đảm bảo chất lượng phần mềm là những hoạt động nào?
15. Giải thích nội dung tóm tắt của mỗi hoạt động chính đảm bảo chất lượng?

#### 1.3. Rà soát phần mềm

16. Rà soát phần mềm được hiểu là gì (Khái niệm, mục tiêu, cách thức áp dụng)? Nêu các lợi ích của việc rà soát? Nếu không thực hiện rà soát thì sao?
17. Các hình thức của hoạt động rà soát? Trình bày khái niệm, mục tiêu của rà soát kỹ thuật chính thức?

18. Vẽ sơ đồ tiến trình của hoạt động rà soát và giải thích sơ bộ nội dung mỗi bước?
19. Trình bày nội dung cơ bản một cuộc họp rà soát: *thành phần, thời gian, công việc cần làm, phương châm ?*
20. Các sản phẩm của cuộc họp rà soát là gì? Nội dung, vai trò của mỗi sản phẩm đó?
21. Khi nào tiến hành rà soát? cần rà soát những sản phẩm gì?
22. Trình bày nội dung, danh mục rà soát của
  - a. rà soát kỹ nghệ hệ thống?
  - b. rà soát việc lập kế hoạch?
  - c. rà soát phân tích yêu cầu phần mềm ?
  - e. rà soát thiết kế phần mềm ?
  - f. rà soát khâu lập mã phần mềm?
  - g. rà soát kiểm thử phần mềm ?

## 2. Các độ đo đặc trưng chất lượng phần mềm

### 2.1. Các độ đo chỉ số chất lượng chương trình

23. Nêu các ký hiệu và giải thích nội dung, ý nghĩa các đại lượng :  $s_1, s_2, s_3, s_4, s_5, s_6, s_7$  và các độ đo trung gian:  $D1=1\&0$ ,  $(D2=1-s_2/s_1)$ ,  $(D3=1-s_3/s_1)$ ,  $(D4=1-s_5/s_4)$ ,  $(D5=1-s_6/s_4)$ ,  $(D6=1-s_7/s_1)$ ?
24. Sử dụng công thức  $\sum w_i D_i$  với  $\sum w_i = 1$  như thế nào và để làm gì?
25. Giải thích nội dung các thành phần và ý nghĩa của độ đo

$$SMI = \frac{MT - Fa - Fc - Fd}{MT} \text{ và cách sử dụng nó?}$$

26. Số đo độ phức tạp của McCabe dựa trên cái gì và những đại lượng cụ thể nào?
27. Đảm bảo chất lượng phần mềm dựa trên thống kê nghĩa là gì? Nó gồm những công việc gì? Kể ít nhất 5 nguyên nhân của những khiếm khuyết trong phần mềm?
28. Nêu công thức tính khiếm khuyết của sản phẩm ở một pha phát triển? và công thức tính khiếm khuyết của sản phẩm cuối cùng? Giải thích ý nghĩa của nó?
29. Tiếp cận hình thức cho SQA nghĩa là gì? Quá trình phòng sạch là gì? Phương châm của kỹ thuật này là gì?

### 2.2. Các độ đo về sự tin cậy và an toàn

30. Độ tin cậy của phần mềm hiểu là cái gì? Đo độ tin cậy dựa trên những dữ liệu nào?
31. Thế nào là thất bại của phần mềm? Có mấy thang bậc? là những thang bậc nào?
32. Nêu chỉ tiêu để tính độ tin cậy? Nêu công thức tính độ sẵn sàng? Giải thích ý nghĩa của chúng?
33. Có những mô hình độ tin cậy nào? Nó dựa trên tham biến nào và trên giả thiết nào? Mô hình độ tin cậy gieo hạt dựa trên ý tưởng nào? Mục tiêu để làm gì?

34. Độ an toàn phần mềm là cái gì? Có những phương pháp nào để phân tích độ an toàn?
35. Khảo sát nhu cầu SQA gồm những nội dung gì? nhằm trả lời cho câu hỏi gì? Nếu có nhu cầu thì làm gì?
36. Có những vấn đề gì đặt ra khi triển khai SQA? Lợi ích của SQA là gì? Nguyên tắc chi phí hiệu quả của SQA là gì?

### **3. Kiểm thử phần mềm**

#### **3.1. Khái niệm về kiểm thử**

37. Tại sao phải kiểm thử phần mềm? Mục tiêu kiểm thử là gì? Từ đó có những quan niệm sai gì về kiểm thử phần mềm?
38. Thế nào là một ca kiểm thử tốt? ca kiểm thử thành công? Lợi ích phụ của kiểm thử là gì?
39. Biểu đồ dòng thông tin kiểm thử mô tả cái gì? vẽ biểu đồ của nó?
40. Nêu các đối tượng, các phương pháp kiểm thử phần mềm? chúng thường được sử dụng vào giai đoạn nào của quá trình phát triển?
41. Một ca kiểm thử là cái gì? Mục tiêu thiết kế ca kiểm thử? các bước để thiết kế một ca kiểm thử?
42. Kiểm thử hộp trắng là cái gì? Nêu các đặc trưng của nó?
43. Kiểm thử hộp đen là cái gì? Nêu các đặc trưng của nó?
44. Chiến lược kiểm thử phần mềm là cái gì? Nêu các nguyên tắc trong chiến lược kiểm thử phần mềm?
45. Nêu các bước của chiến lược kiểm thử thời gian thực và giải thích nội dung mỗi bước?
46. Có những loại công cụ tự động nào trợ giúp kiểm thử? Mô tả nội dung mỗi loại?
47. Ai là người phải tham gia kiểm thử phần mềm? Nêu vai trò và trách nhiệm của mỗi đối tượng?

#### **3.2. Các phương pháp kiểm thử**

##### **a. Kiểm thử hộp trắng**

48. Kiểm thử hộp trắng dựa trên cơ sở nào để thiết kế các ca kiểm thử? Thiết kế ca kiểm thử phải đảm bảo điều kiện gì?
49. Đồ thị dòng gồm những yếu tố nào? xây dựng nó dựa vào đâu? Nó có các đặc trưng gì? Đồ thị dòng dùng để làm gì?
50. Con đường cơ bản trong đồ thị dòng là cái gì? Độ phức tạp của chu trình là gì? Nêu các công thức tính độ phức tạp?
51. Ma trận thử nghiệm được cấu trúc như thế nào? Nó dùng để làm gì?
52. Nêu các loại điều kiện trong cấu điều khiển và cho ví dụ? Có những loại sai nào trong điều kiện khi kiểm thử?

53. Chiến lược kiểm thử phân nhánh nghĩa là gì? Yêu cầu đặt ra cho kiểm thử phân nhánh là gì?
54. Chiến lược kiểm thử miền là cái gì? Nó dựa trên tư tưởng nào?
55. Chiến lược kiểm thử BRO là cái gì? Nó dựa trên tư tưởng nào?
56. Lấy ví dụ về các điều kiện “ràng buộc ra” cho các trường hợp: 1 biến Bool, hợp của biến Bool và biểu thức quan hệ, hợp của hai biểu thức quan hệ?
57. Kiểm thử điều khiển dòng dữ liệu nghĩa là gì? Cho ví dụ?
58. Kiểm thử điều khiển vòng lặp nghĩa là gì? Cho ví dụ?

#### **b. Kiểm thử hộp đen**

59. Mô hình của kiểm thử hộp đen quan tâm đến nhân tố nào của phần mềm? Nó nhằm tìm ra các loại sai nào? Nêu các phương pháp áp dụng cho nó?
60. Trình bày phương pháp phân hoạch: nguyên tắc, mục tiêu và thiết kế ca kiểm thử? Phương châm xác định lớp tương đương là gì?
61. Phân tích giá trị biên nghĩa là gì? Phương châm phân tích giá trị biên là gì?
62. Kỹ thuật nhân quả nghĩa là gì? Nêu các bước của kỹ thuật này?
63. Chiến lược kiểm thử thời gian thực gồm mấy bước? là những bước nào? Giải thích nội dung cơ bản mỗi bước?

#### **c. Kiểm thử đơn vị**

64. Kiểm thử đơn vị là gì? Quan hệ của nó với hoạt động mã hóa như thế nào?
65. Hoạt động kiểm thử đơn vị gồm những nội dung gì? Nó liên quan đến những nhân tố nào? Nêu một vài câu hỏi cần kiểm thử cho các nhân tố đó?
66. Kỹ thuật kiểm thử đơn vị sử dụng là gì? vì sao phải sử dụng kỹ thuật đó? Có những khó khăn, thuận lợi gì?

#### **d. Kiểm thử tích hợp**

67. Kiểm thử tích hợp thực hiện khi nào? Tại sao phải kiểm thử tích hợp? Nêu một số câu hỏi đặt ra cho kiểm thử tích hợp?
68. Có những phương pháp gì được áp dụng cho kiểm thử tích hợp? mô tả tóm tắt nội dung mỗi phương pháp?
69. Nêu các bước kiểm thử tích hợp từ trên xuống? Ưu nhược điểm của cách tiếp cận này?
70. Nêu các bước kiểm thử tích hợp từ dưới lên? Ưu nhược điểm của cách tiếp cận này?
71. Các tài liệu kiểm thử tích hợp gồm những loại gì?

#### **e. Kiểm thử hệ thống**

72. Kiểm thử Beta là cái gì? Kiểm thử Alpha là cái gì? Nêu sự giống và khác nhau cơ bản giữa chúng ?



- 73. Nội dung chính của kiểm thử hệ thống ? Nêu một số câu hỏi đặt ra cho việc kiểm thử hệ thống ?
- 74. Kiểm thử phục hồi là gì ?
- 75. Kiểm thử an ninh là gì ?
- 76. Kiểm thử áp lực là gì
- 77. Kiểm thử thi hành là gì
- 78. Gỡ rối được hiểu là gì ? Nó thực hiện khi nào ? Khó khăn của việc gỡ rối là gì?
- 79. Trình bày tiến trình gỡ rối ? các cách thức gỡ rối ? ưu nhược điểm của chúng?

#### **4. Quản lý cấu hình phần mềm**

- 80. Quản lý cấu hình phần mềm là gì? Nội dung của hoạt động quản lý cấu hình gồm những công việc gì?
- 81. Cấu hình phần mềm được hiểu là cái gì? nội dung các khoản mục chính của cấu hình phần gồm những gì?
- 82. Quản lý cấu hình nhằm mục tiêu gì? Năm nhiệm vụ của quản lý cấu hình là gì?
- 83. Phương pháp gì được áp dụng cho việc quản lý cấu hình? Mốc giới là cái gì? Sử dụng mốc giới để kiểm soát sự thay đổi như thế nào?
- 84. Trình bày tiến trình kiểm soát sự thay đổi?
- 85. Phiên bản là cái gì? Làm thế nào để kiểm soát các phiên bản
- 86. Kiểm toán cấu hình phần mềm nghĩa là gì? Hoạt động kiểm toán cần trả lời những câu hỏi gì?
- 87. Báo cáo hiện trạng nghĩa là gì? Nó cần trả lời được những câu hỏi gì? Đầu ra của báo cáo hiện trạng dành cho ai? mục tiêu của nó là gì?

## TÀI LIỆU THAM KHẢO

- [1] Hoàng Văn Kiêm, *Giáo trình chuyên đề Nguyên lý và phương pháp ngôn ngữ lập trình*, Đại học Quốc gia TP. Hồ Chí Minh, 2005.
- [2] Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing. A Context-Driven Approach*, John Wiley & Sons, 2001.
- [3] Software Testing and Quality Control - Knowledge Bases, <http://www.compinfo-center.com/tpsw12-t.htm>
- [4] Cem Kaner, James Bach, *Black Box Software Testing*, Center for Software Testing Education & Research, Florida Institute of Technology, 2005.
- [5] The Test Management Guide - A to Z and FAQs, <http://www.ruleworks.co.uk/testguide/>
- [6] The Test Management Guide - A to Z and FAQs, <http://www.ruleworks.co.uk/testguide/>
- [8] IPL, *An Introduction to Software Testing*, IPL Information Processing Ltd, 2002.
- [9] [BEI90] Beizer, B., *Software Testing Techniques*, 2d ed., Van Nostrand Reinhold, 1990,
- [10] [DEU79] Detsch, M., “Verification and Validation” in *software Engineering*, (R. Jensen and C. Tonies, eds.) Prentice-Hall, 1979, pp 329-408.
- [11] *Software Engineering A Practitioner’s Approach*, Roger S. Pressman.