Chapter 15

## Implementation

**Learning Objectives**

After studying this chapter, you should be able to

* Perform the implementation workflow.
* Perform black-box, glass-box, and non-execution-based unit testing.
* Perform integration testing, product testing, and acceptance testing.
* Appreciate the need for good programming practices and programming standards.

Mục tiêu học tập

Sau khi nghiên cứu chương này, bạn sẽ có thể

• Thực hiện quy trình thực hiện.

• Thực hiện kiểm thử đơn vị hộp đen, hộp thủy tinh và không dựa trên thực thi.

• Thực hiện kiểm thử tích hợp, kiểm thử sản phẩm và kiểm thử chấp nhận.

• Đánh giá cao nhu cầu thực hành lập trình tốt và các tiêu chuẩn lập trình.

Implementation is the process of translating the detailed design into code. When this is done by a single individual, the process is relatively well understood. But, most real-life products today are too large to be implemented by one programmer within the given time constraints. Instead, the product is implemented by a team, working at the same time on different components of the product; this is termed **programming-in-the-many**. Issues associated with programming-in-the-many are examined in this chapter.

Thực hiện là quá trình dịch thiết kế chi tiết thành mã. Khi điều này được thực hiện bởi một cá nhân duy nhất, quá trình này được hiểu tương đối rõ. Tuy nhiên, hầu hết các sản phẩm thực tế ngày nay đều quá lớn để một lập trình viên có thể triển khai trong thời gian hạn chế nhất định. Thay vào đó, sản phẩm được thực hiện bởi một nhóm, làm việc cùng lúc trên các thành phần khác nhau của sản phẩm; điều này được gọi là lập trình-trong-nhiều. Các vấn đề liên quan đến lập trình-trong-nhiều được xem xét trong chương này.

**15.1 Choice of Programming Language 15.1 Lựa chọn ngôn ngữ lập trình**

In most cases, the issue of which programming language to choose for the implementation simply does not arise. Suppose the client wants a product to be implemented in, say, Small- talk. Perhaps, in the opinion of the development team, Smalltalk is entirely unsuitable for the product. Such an opinion is irrelevant to the client. Management of the development organi- zation has only two choices: Implement the product in Smalltalk or turn down the job.

Similarly, if the product has to be implemented on a specific computer and the only language available on that computer is assembler, then again there is no choice. If no other language is available, either because no compiler has yet been developed for any high-level language on that computer or management is not prepared to pay for a new C++ compiler for the stipulated computer, then again clearly the issue of choice of programming language is not relevant.

**498**

A more interesting situation is this: A contract specifies that the product is to be imple- mented in “the most-suitable” programming language. What language should be chosen? To answer this question, consider the following scenario. QQQ Corporation has been writ- ing COBOL products for over 30 years. The entire 200-member software staff of QQQ, from the most junior programmer to the vice-president for software, has COBOL expertise. Why on earth should the most suitable programming language be anything but COBOL? The introduction of a new language, Java, for example, would mean having to hire new pro- grammers, or, at the very least, existing staff would have to be intensively retrained. Hav- ing invested all that money and effort in Java training, management might well decide that future products also should be implemented in Java. Nevertheless, all the existing COBOL products would have to be maintained. There then would be two classes of programmers, COBOL maintenance programmers and Java programmers writing the new applications. Quite undeservedly, maintenance almost always is considered inferior to developing new applications, so there would be distinct unhappiness among the ranks of the COBOL programmers. This unhappiness would be compounded by the fact that Java programmers usually are paid more than COBOL programmers because Java programmers are in short supply. Although QQQ has excellent development tools for COBOL, a Java compiler would have to be purchased, as well as appropriate Java CASE tools. Additional hardware may have to be purchased or leased to run this new software. Perhaps most serious of all, QQQ has accumulated hundreds of person-years of COBOL expertise, the kind of expertise that can be gained only through hands-on experience, such as what to do when a certain cryptic error message appears on the screen or how to handle the quirks of the compiler. In brief, it would seem that “the most suitable” programming language could be only COBOL—any other choice would be financial suicide, either from the viewpoint of the cost involved or as a consequence of plummeting staff morale leading to poor-quality code.

And yet, the most suitable programming language for QQQ Corporation’s latest project may indeed be some language other than COBOL. Notwithstanding its position as the world’s most widely used programming language (see Just in Case You Wanted to Know Box 15.1), COBOL is suited for only one class of software products, data-processing ap- plications. If QQQ Corporation has software needs outside this class, then COBOL rapidly loses its attractiveness. For example, if QQQ wishes to construct a knowledge-based prod- uct using artificial intelligence (AI) techniques, then an AI language such as Lisp could be used; COBOL is totally unsuitable for AI applications. If large-scale communications software is to be built, perhaps because QQQ requires satellite links to hundreds of branch offices all over the world, then a language such as Java would prove far more suitable than COBOL. If QQQ is to go into the business of writing systems software, such as operating systems, compilers, and linkers, then COBOL very definitely is unsuitable. And, if QQQ Corporation decides to go into defense contracting, management will soon discover that COBOL simply cannot be used for real-time embedded software.

The issue of which programming language to use often can be decided by using cost– benefit analysis (Section 5.2). That is, management must compute the dollar cost of an implementation in COBOL as well as the dollar benefits, present and future, of using COBOL. This computation must be repeated for every language under consideration. The language with the largest expected gain (that is, the difference between estimated benefits and estimated costs) is then the appropriate implementation language. Another way of de- ciding which programming language to select is to use risk analysis. For each language



**Just in Case You Wanted to Know Box 15.1**

**500** Part B *The Workflows of the Software Life Cycle*

Far more code has been implemented in COBOL than in all other programming languages put together. COBOL is the most widely used language primarily because COBOL is a prod- uct of the U.S. Department of Defense (DoD). Developed under the direction of the late Rear-Admiral Grace Murray Hopper, COBOL was approved by the DoD in 1960. Thereafter, the DoD would not buy hardware for running data-processing applications unless that hardware had a COBOL compiler [Sammet, 1978]. The DoD was, and still is, the world’s largest purchaser of computer hardware; and in the 1960s, a considerable proportion of DoD software was implemented for data processing. As a result, COBOL compilers were developed as a matter of urgency for virtually every computer. This widespread availability of COBOL, at a time when the only alternative language usually was assembler, resulted in COBOL becoming the world’s most popular programming language.

Languages such as C, C++, Java, and the 4GLs undoubtedly are growing in popularity for new applications. Nevertheless, postdelivery maintenance still is the major software activity, and this maintenance is being performed on existing COBOL software. In short,

the DoD put its stamp onto the world’s software via its first major programming language, COBOL.

Another reason for the popularity of COBOL is that COBOL frequently is the best lan- guage for implementing a data-processing product. In particular, COBOL generally is the language of choice when money is involved. Financial books have to balance, so rounding errors cannot be allowed to creep in. Therefore, all computations have to be performed using integer arithmetic. COBOL supports integer arithmetic on very large numbers (that is, billions of dollars). In addition, COBOL can handle very small numbers, such as fractions of a cent. Banking regulations require interest computations to be calculated to at least four decimal places of a cent, and COBOL can do this arithmetic with ease as well. Finally, COBOL probably has the best formatting, sorting, and report generation facilities of any third-generation language (or high-level language, see Section 15.2). All these reasons have made COBOL an excellent choice for implementing a data-processing product.

As mentioned in Section 8.11.4, the current COBOL language standard is for an object- oriented language. This standard surely will further boost the popularity of COBOL.

under consideration, a list is made of the potential risks and ways of resolving them. The language for which the overall risk is the smallest then is selected.

Currently, software organizations are under pressure to develop new software in an object-oriented language—any object-oriented language. The question that arises is this: Which is the appropriate object-oriented language? Twenty years ago, there really was only one choice, Smalltalk. Today, however, the most widely used object-oriented programming language is C++ [Borland, 2002], with Java in second place. There are a number of reasons for the popularity of C++. One is the widespread availability of C++ compilers. In fact, some C++ compilers simply translate the source code from C++ into C, and then invoke the C compiler. Therefore, any computer with a C compiler essentially can handle C++.

But the real explanation for the popularity of C++ is its apparent similarity to C. This is unfortunate, in that a number of managers view C++ as a superset of C and, therefore, conclude that any programmer who knows C can quickly pick up the additional pieces. Indeed, from just a syntactical viewpoint, C++ essentially is a superset of C. After all, virtually any C program can be compiled using a C++ compiler. Conceptually, however, C++ is totally different from C. C is a product of the classical paradigm, whereas C++

is for the object-oriented paradigm. Using C++ makes sense only if object-oriented techniques have been used and if the product is organized around objects and classes, not functions.

Therefore, before an organization adopts C++, it is essential that the relevant software professionals be trained in the object-oriented paradigm. It is particularly important that the information of Chapter 7 be taught. Unless it is clear to all involved, and particularly to management, that the object-oriented paradigm is a different way of developing software and what the precise differences are, the classical paradigm will just continue to be used but with the code implemented in C++ rather than C. When organizations are disappointed with the results of switching from C to C++, a major contributory factor is a lack of educa- tion in the object-oriented paradigm.

Suppose that an organization decides to adopt Java. In that case it is not possible to move gradually from the classical paradigm to the object-oriented paradigm. Java is a pure object-oriented programming language; it does not support the functions and procedures of the classical paradigm. Unlike a hybrid object-oriented language such as C++, Java programmers have to use the object-oriented paradigm (and only the object-oriented para- digm) from the very beginning. Because of the necessity of an abrupt transition from the one paradigm to the other, education and training are even more important when adopting Java (or another pure object-oriented language, such as Smalltalk) than if the organization were to switch to a hybrid object-oriented language like C++ or OO-COBOL.

Trong hầu hết các trường hợp, vấn đề chọn ngôn ngữ lập trình nào để triển khai đơn giản là không phát sinh. Giả sử khách hàng muốn một sản phẩm được triển khai trong, chẳng hạn như Small-talk. Có lẽ, theo ý kiến của nhóm phát triển, Smalltalk hoàn toàn không phù hợp với sản phẩm. Một ý kiến ​​như vậy là không liên quan đến khách hàng. Quản lý của tổ chức phát triển chỉ có hai lựa chọn: Triển khai sản phẩm trong Smalltalk hoặc từ chối công việc.

Tương tự, nếu sản phẩm phải được triển khai trên một máy tính cụ thể và ngôn ngữ duy nhất khả dụng trên máy tính đó là trình biên dịch chương trình hợp ngữ, thì một lần nữa không có lựa chọn nào khác. Nếu không có ngôn ngữ nào khác, bởi vì chưa có trình biên dịch nào được phát triển cho bất kỳ ngôn ngữ cấp cao nào trên máy tính đó hoặc ban quản lý không sẵn sàng trả tiền cho trình biên dịch C++ mới cho máy tính quy định, thì rõ ràng lại là vấn đề lựa chọn lập trình ngôn ngữ không liên quan.

Một tình huống thú vị hơn là: Một hợp đồng quy định rằng sản phẩm sẽ được triển khai bằng ngôn ngữ lập trình “thích hợp nhất”. Nên chọn ngôn ngữ nào? Để trả lời câu hỏi này, hãy xem xét kịch bản sau đây. QQQ Corporation đã sản xuất các sản phẩm COBOL trong hơn 30 năm. Toàn bộ nhân viên phần mềm gồm 200 thành viên của QQQ, từ lập trình viên trẻ nhất đến phó chủ tịch phụ trách phần mềm, đều có chuyên môn về COBOL. Tại sao ngôn ngữ lập trình phù hợp nhất lại không phải là COBOL? Ví dụ, việc giới thiệu một ngôn ngữ mới, Java, có nghĩa là phải thuê các lập trình viên mới, hoặc ít nhất, đội ngũ nhân viên hiện có sẽ phải được đào tạo lại chuyên sâu. Sau khi đã đầu tư tất cả số tiền và nỗ lực đó vào việc đào tạo Java, ban quản lý có thể quyết định rằng các sản phẩm trong tương lai cũng nên được triển khai bằng Java. Tuy nhiên, tất cả các sản phẩm COBOL hiện có sẽ phải được duy trì. Khi đó sẽ có hai lớp lập trình viên, lập trình viên bảo trì COBOL và lập trình viên Java viết ứng dụng mới. Hoàn toàn không xứng đáng, việc bảo trì hầu như luôn bị coi là kém hơn so với việc phát triển các ứng dụng mới, vì vậy sẽ có sự không hài lòng rõ rệt trong hàng ngũ các lập trình viên COBOL. Điều không vui này sẽ càng trầm trọng hơn bởi thực tế là các lập trình viên Java thường được trả nhiều hơn các lập trình viên COBOL bởi vì các lập trình viên Java đang bị thiếu hụt. Mặc dù QQQ có các công cụ phát triển tuyệt vời dành cho COBOL, nhưng sẽ phải mua một trình biên dịch Java, cũng như các công cụ Java CASE thích hợp. Phần cứng bổ sung có thể phải được mua hoặc thuê để chạy phần mềm mới này. Có lẽ nghiêm trọng nhất, QQQ đã tích lũy được kiến ​​thức chuyên môn hàng trăm năm về COBOL, loại kiến ​​thức chuyên môn chỉ có thể đạt được thông qua kinh nghiệm thực hành, chẳng hạn như phải làm gì khi một thông báo lỗi khó hiểu nào đó xuất hiện trên màn hình hoặc cách thực hiện. để xử lý các quirks của trình biên dịch. Tóm lại, có vẻ như ngôn ngữ lập trình “phù hợp nhất” chỉ có thể là COBOL—bất kỳ lựa chọn nào khác sẽ là tự sát về mặt tài chính, từ quan điểm về chi phí liên quan hoặc do hậu quả của tinh thần nhân viên sa sút dẫn đến mã kém chất lượng.

Chưa hết, ngôn ngữ lập trình phù hợp nhất cho dự án mới nhất của QQQ Corporation thực sự có thể là một ngôn ngữ nào đó khác ngoài COBOL. Bất chấp vị trí là ngôn ngữ lập trình được sử dụng rộng rãi nhất trên thế giới (xem Đề phòng trường hợp bạn muốn biết Hộp 15.1), COBOL chỉ phù hợp với một loại sản phẩm phần mềm, ứng dụng xử lý dữ liệu. Nếu QQQ Corporation có nhu cầu phần mềm bên ngoài lớp này, thì COBOL sẽ nhanh chóng mất đi sức hấp dẫn của nó. Ví dụ: nếu QQQ muốn xây dựng một sản phẩm dựa trên tri thức bằng kỹ thuật trí tuệ nhân tạo (AI), thì có thể sử dụng một ngôn ngữ AI như Lisp; COBOL hoàn toàn không phù hợp với các ứng dụng AI. Nếu phần mềm truyền thông quy mô lớn được xây dựng, có lẽ vì QQQ yêu cầu liên kết vệ tinh tới hàng trăm văn phòng chi nhánh trên khắp thế giới, thì một ngôn ngữ như Java sẽ tỏ ra phù hợp hơn nhiều so với COBOL. Nếu QQQ đi vào lĩnh vực viết phần mềm hệ thống, chẳng hạn như hệ điều hành, trình biên dịch và trình liên kết, thì COBOL chắc chắn là không phù hợp. Và, nếu QQQ Corporation quyết định ký hợp đồng quốc phòng, ban quản lý sẽ sớm phát hiện ra rằng COBOL đơn giản là không thể được sử dụng cho phần mềm nhúng thời gian thực.

Vấn đề sử dụng ngôn ngữ lập trình nào thường xuyên có thể được quyết định bằng cách sử dụng phân tích chi phí – lợi ích (Phần 5.2). Nghĩa là, ban quản lý phải tính toán chi phí bằng đô la của việc triển khai trong COBOL cũng như lợi ích bằng đô la, hiện tại và tương lai, của việc sử dụng COBOL. Tính toán này phải được lặp lại cho mọi ngôn ngữ đang được xem xét. Ngôn ngữ có lợi ích dự kiến ​​lớn nhất (nghĩa là sự khác biệt giữa lợi ích ước tính và chi phí ước tính) khi đó là ngôn ngữ triển khai phù hợp. Một cách khác để quyết định chọn ngôn ngữ lập trình nào là sử dụng phân tích rủi ro. Đối với mỗi ngôn ngữ đang được xem xét, một danh sách được tạo thành từ các rủi ro tiềm ẩn và cách giải quyết chúng. Ngôn ngữ mà rủi ro tổng thể là nhỏ nhất sau đó được chọn.

Hiện tại, các tổ chức phần mềm đang chịu áp lực phát triển phần mềm mới bằng ngôn ngữ hướng đối tượng—bất kỳ ngôn ngữ hướng đối tượng nào. Câu hỏi đặt ra là: Ngôn ngữ hướng đối tượng nào phù hợp? Hai mươi năm trước, thực sự chỉ có một sự lựa chọn, Smalltalk. Tuy nhiên, ngày nay, ngôn ngữ lập trình hướng đối tượng được sử dụng rộng rãi nhất là C++ [Borland, 2002], với Java ở vị trí thứ hai. Có một số lý do cho sự phổ biến của C++. Một là sự phổ biến rộng rãi của các trình biên dịch C++. Trên thực tế, một số trình biên dịch C++ chỉ cần dịch mã nguồn từ C++ sang C, sau đó gọi trình biên dịch C. Do đó, bất kỳ máy tính nào có trình biên dịch C về cơ bản đều có thể xử lý C++.

Nhưng lời giải thích thực sự cho sự phổ biến của C++ là sự giống nhau rõ ràng của nó với C. Điều này thật không may, ở chỗ một số nhà quản lý xem C++ như một siêu bộ của C và do đó, kết luận rằng bất kỳ lập trình viên nào biết C đều có thể nhanh chóng tiếp thu phần bổ sung. miếng. Thật vậy, chỉ xét về mặt cú pháp, C++ về cơ bản là một siêu bộ của C. Xét cho cùng, hầu như bất kỳ chương trình C nào cũng có thể được biên dịch bằng trình biên dịch C++. Tuy nhiên, về mặt khái niệm, C++ hoàn toàn khác với C. C là sản phẩm của mô hình cổ điển, trong khi C++

là dành cho mô hình hướng đối tượng. Sử dụng C++ chỉ có ý nghĩa nếu các kỹ thuật hướng đối tượng đã được sử dụng và nếu sản phẩm được tổ chức xung quanh các đối tượng và lớp chứ không phải các chức năng.

Do đó, trước khi một tổ chức chấp nhận C++, điều cần thiết là các chuyên gia phần mềm có liên quan phải được đào tạo về mô hình hướng đối tượng. Điều đặc biệt quan trọng là thông tin của Chương 7 phải được giảng dạy. Trừ khi tất cả những người liên quan, và đặc biệt là ban quản lý, hiểu rõ rằng mô hình hướng đối tượng là một cách khác để phát triển phần mềm và sự khác biệt chính xác là gì, thì mô hình cổ điển sẽ tiếp tục được sử dụng nhưng với mã được triển khai trong C++ thì đúng hơn. hơn C. Khi các tổ chức thất vọng với kết quả của việc chuyển đổi từ C sang C++, một yếu tố đóng góp chính là do thiếu giáo dục về mô hình hướng đối tượng.

Giả sử rằng một tổ chức quyết định áp dụng Java. Trong trường hợp đó, không thể chuyển dần từ mô hình cổ điển sang mô hình hướng đối tượng. Java là ngôn ngữ lập trình hướng đối tượng thuần túy; nó không hỗ trợ các chức năng và thủ tục của khung mẫu cổ điển. Không giống như một ngôn ngữ hướng đối tượng lai như C++, các lập trình viên Java phải sử dụng mô hình hướng đối tượng (và chỉ mô hình hướng đối tượng) ngay từ đầu. Do sự cần thiết của việc chuyển đổi đột ngột từ mô hình này sang mô hình khác, giáo dục và đào tạo thậm chí còn quan trọng hơn khi áp dụng Java (hoặc một ngôn ngữ hướng đối tượng thuần túy khác, chẳng hạn như Smalltalk) so với khi tổ chức chuyển sang một đối tượng lai. ngôn ngữ định hướng như C++ hoặc OO-COBOL.

* 1. Fourth-Generation Languages 15.2 Ngôn ngữ thế hệ thứ tư

The first computers had neither interpreters nor compilers. They were programmed in bi- nary, either hardwired with plug boards or by setting switches. Such a binary machine code was a **first-generation language**. The **second-generation languages** were assem- blers, developed in the late 1940s and early 1950s. Instead of having to program in binary, instructions could be expressed in symbolic notation such as

mov $17, next

In general, each assembler instruction is translated into one machine code instruction. So, although assembler was easier to write than machine code and easier for postdelivery maintenance programmers to comprehend, the assembler source code was the same length as the machine code.

The idea behind a **third-generation language** (or high-level language), such as C, C++, Pascal, or Java, is that one statement of a high-level language is compiled to as many as 5 or 10 machine code instructions (this is another example of abstraction; see Section 7.4.1). High-level language code is hence considerably shorter than the equivalent assem- bler code. It is also simpler to understand and, therefore, easier to maintain than assembler code. The fact that the high-level language code may not be quite as efficient as the equiva- lent assembler code generally is a small price to pay for ease in postdelivery maintenance. This concept was taken further in the late 1970s. A major objective in the design of a **fourth-generation language (4GL)** is that each 4GL statement should be equivalent to 30, or even 50, machine code instructions. Products implemented in a 4GL such as Focus

or Natural are shorter and hence quicker to develop and easier to maintain.

Những chiếc máy tính đầu tiên không có trình thông dịch cũng như trình biên dịch. Chúng được lập trình ở chế độ nhị phân, hoặc được nối cứng bằng bảng phích cắm hoặc bằng cách cài đặt công tắc. Mã máy nhị phân như vậy là ngôn ngữ thế hệ đầu tiên. Các ngôn ngữ thế hệ thứ hai là ngôn ngữ hợp ngữ, được phát triển vào cuối những năm 1940 và đầu những năm 1950. Thay vì phải lập trình ở dạng nhị phân, các hướng dẫn có thể được biểu thị bằng ký hiệu tượng trưng, ​​chẳng hạn như

mov $17, tiếp theo

Nói chung, mỗi lệnh hợp ngữ được dịch thành một lệnh mã máy. Vì vậy, mặc dù trình biên dịch mã dễ viết hơn mã máy và các lập trình viên bảo trì sau khi phân phối dễ hiểu hơn, nhưng mã nguồn của trình biên dịch mã có cùng độ dài với mã máy.

Ý tưởng đằng sau ngôn ngữ thế hệ thứ ba (hoặc ngôn ngữ cấp cao), chẳng hạn như C, C++, Pascal hoặc Java, là một câu lệnh của ngôn ngữ cấp cao được biên dịch thành 5 hoặc 10 lệnh mã máy ( đây là một ví dụ khác về trừu tượng hóa; xem Phần 7.4.1). Do đó, mã ngôn ngữ cấp cao ngắn hơn đáng kể so với mã hợp ngữ tương đương. Nó cũng đơn giản hơn để hiểu và do đó dễ bảo trì hơn mã hợp ngữ. Thực tế là mã ngôn ngữ cấp cao có thể không hoàn toàn hiệu quả như mã trình hợp ngữ tương đương nói chung là một cái giá nhỏ phải trả để dễ dàng bảo trì sau khi giao hàng. Khái niệm này đã được đưa ra xa hơn vào cuối những năm 1970. Mục tiêu chính trong thiết kế ngôn ngữ thế hệ thứ tư (4GL) là mỗi câu lệnh 4GL phải tương đương với 30 hoặc thậm chí 50 hướng dẫn mã máy. Các sản phẩm được triển khai trong 4GL như Focus

hoặc Tự nhiên ngắn hơn và do đó phát triển nhanh hơn và dễ bảo trì hơn.

**502** Part B *The Workflows of the Software Life Cycle*



**Just in Case You Wanted to Know**

**Box 15.2**

Some years ago I hailed a cab outside Grand Central Station in New York City and said to the driver, “Please take me to Lincoln Center.” This was a *nonprocedural* request, because I expressed the desired result but left it to the driver to decide how to achieve that result. It turned out that the driver was an immigrant from Central Europe who had been in America less than 2 months and knew virtually nothing about the geography of New York City or the English language. As a result, I quickly replaced my nonprocedural request with a **procedural** request of the form, “Straight, straight. Take a right at the next light. I said right. Right, here, yes, right! Now straight. Slow down, please. I said slow down. For heaven’s sake, slow down!” and so on, until we finally reached Lincoln Center.

It is difficult to program in machine code. It is somewhat easier to program in assem- bler, and easier still to use a high-level language. A second major design objective of a 4GL is ease in programming. In particular, many 4GLs are **nonprocedural** (see Just in Case You Wanted to Know Box 15.2 for an insight into this term). For example, consider the command

**for every** surveyor

**if** rating **is** excellent

**add** 6500 **to** salary

It is up to the compiler of the 4GL to translate this nonprocedural instruction into a sequence of machine code instructions that can be executed procedurally.

Success stories abound from organizations that have switched to a 4GL. A few that previously used COBOL reported a 10-fold increase in productivity through use of a 4GL. Many organizations found that their productivity indeed increased through use of a 4GL but not spectacularly so. Other organizations tried a 4GL and were bitterly disappointed with the results.

One reason for this inconsistency is that it is unlikely that one 4GL will be appropriate for all products. On the contrary, it is important to select the correct 4GL for the specific product. For example, Playtex used IBM’s Application Development Facility (ADF) and reported an 80 to 1 productivity increase over COBOL. Notwithstanding this impressive result, Playtex subsequently returned to COBOL for products deemed by management to be less well suited to ADF [Martin, 1985].

A second reason for these inconsistent results is that many 4GLs are supported by power- ful CASE workbenches and environments (Section 5.7). CASE workbenches and environ- ments can be both a strength and a weakness. As explained in Section 5.12, it is inadvisable to introduce large-scale CASE within an organization with a low maturity level. The reason is that the purpose of a CASE workbench or environment is to support the software process. An organization at level 1 has no software process in place. If at this point CASE is introduced as part of the transition to a 4GL, this imposes a process onto an organization not ready for any sort of process. The usual consequences at best are unsatisfactory and can be disastrous. In fact, a number of reported 4GL failures can be ascribed to the effects of the associated CASE environment rather than to the 4GL itself.

The attitudes of 43 organizations to 4GLs are reported in [Guimaraes, 1985]. This re- search found that use of a 4GL reduced user frustration because the data-processing de- partment could respond more quickly when a user needed information extracted from the

organization’s database. However, there also were a number of problems. Some 4GLs proved to be slow and inefficient, with long response times. One product consumed 60 percent of the CPU cycles on an IBM 4331 mainframe, while supporting, at most, 12 con- current users. Overall, the 28 organizations that had been using a 4GL for over 3 years felt that the benefits outweighed the costs.

No one 4GL dominates the software market. Instead, there are hundreds of 4GLs; some of them, including DB2, Oracle, and PowerBuilder, have sizable user groups. This widespread proliferation of 4GLs is further evidence that care has to be taken in selecting the correct 4GL. Of course, few organizations can afford to support more than one 4GL. Once a 4GL has been chosen and used, the organization must either use that 4GL for subsequent products or fall back on the language used before the 4GL was introduced.

Notwithstanding the potential productivity gain, there could be danger in using a 4GL the wrong way. Many organizations currently have a large backlog of products to be devel- oped and a long list of postdelivery maintenance tasks to be performed. A design objective of many 4GLs is **end-user programming**, that is, programming by the person who will use the product. For example, before the advent of 4GLs, the investment manager of an insurance company would ask the data-processing manager for a product that would dis- play certain information regarding the bond portfolio. The investment manager then would wait a year or so for the data-processing group to find the time to develop the product. A 4GL was desired that would be so simple to use that the investment manager, previously untrained in programming, could implement the desired product unaided. End-user pro- gramming was intended to help reduce the development backlog, leaving the professionals to maintain existing products.

In practice, end-user programming can be dangerous. First, consider the situation when all product development is performed by computer professionals. Computer professionals are trained to mistrust computer output. After all, probably less than 1 percent of all out- put during product development is correct. On the other hand, the user is told to trust all computer output, because no product should be delivered to the user until it is fault free. Now consider the situation when end-user programming is encouraged. When a user who is inexperienced in programming implements code with a user-friendly, nonprocedural 4GL, the natural tendency is for that user to believe the output. After all, for years the user has been instructed to trust computer output. As a result, many business decisions have been based on data generated by hopelessly incorrect end-user code. In some cases, the user- friendliness of certain 4GLs has led to financial catastrophes.

Another potential danger lies in the tendency, in some organizations, to allow users to implement 4GL products that update the organization’s database. A programming mistake made by a user eventually may result in the corruption of the entire database. The lesson is clear: Programming by inexperienced or inadequately trained users can be exceedingly dangerous, if not fatal, to the financial health of a corporation.

The ultimate choice of a 4GL is made by management. In making such a decision, management should be guided by the many success stories resulting from the use of a 4GL. At the same time, management should carefully analyze the failures caused by using an inappropriate 4GL, by premature introduction of a CASE environment, and by poor management of the development process. For example, a common cause of failure is neglecting to train the development team thoroughly in all aspects of the 4GL, includ- ing relational database theory [Date, 2003] where appropriate. Management should study



##### Just in Case You Wanted to Know Box 15.3

In the late 1970s, a small software organization in Johannesburg, South Africa, consisted of two programming teams. Team A was made up of émigrés from Mozambique. They were of Portuguese extraction, and their native language was Portuguese. Their code was well writ- ten. Variable names were meaningful but unfortunately only to a speaker of Portuguese. Team B comprised Israeli immigrants whose native language was Hebrew. Their code was equally well written, and the names they chose for their variables were equally meaningful—but only to a speaker of Hebrew.

One day, team A resigned en masse, together with its team leader. Team B was totally unable to maintain any of the excellent code that team A had written, because they spoke no Portuguese. The variable names, meaningful as they were to Portuguese speakers, were incomprehensible to the Israelis, whose linguistic abilities were restricted to Hebrew and English. The owner of the software organization was unable to hire enough Portuguese- speaking programmers to replace team A, and the company soon went into bankruptcy, under the weight of numerous lawsuits from disgruntled customers whose code was now essentially unmaintainable.

The situation could have been avoided easily. The head of the company should have in- sisted from the start that all variable names be in English, the language understood by every South African computer professional. Variable names then would have been meaningful to any maintenance programmer.

both the successes and failures in the specific application area and learn from past mis- takes. Choosing the correct 4GL can mean the difference between a major success and dismal failure.

Having decided on the implementation language, the next issue is how software engi- neering principles can lead to better-quality code.

Rất khó để lập trình bằng mã máy. Có phần dễ dàng hơn để lập trình trong trình biên dịch mã hợp ngữ, và vẫn dễ dàng hơn khi sử dụng một ngôn ngữ cấp cao. Mục tiêu thiết kế chính thứ hai của 4GL là dễ lập trình. Đặc biệt, nhiều 4GL không theo thủ tục (xem Đề phòng trường hợp bạn muốn biết Hộp 15.2 để hiểu rõ hơn về thuật ngữ này). Ví dụ, xét lệnh

cho mọi nhà khảo sát

nếu đánh giá là xuất sắc

thêm 6500 vào lương

Tùy thuộc vào trình biên dịch của 4GL để dịch hướng dẫn phi thủ tục này thành một chuỗi các hướng dẫn mã máy có thể được thực thi theo thủ tục.

Có rất nhiều câu chuyện thành công từ các tổ chức đã chuyển sang 4GL. Một số COBOL đã sử dụng trước đây đã báo cáo năng suất tăng gấp 10 lần nhờ sử dụng 4GL. Nhiều tổ chức nhận thấy rằng năng suất của họ thực sự tăng lên nhờ sử dụng 4GL nhưng không ngoạn mục như vậy. Các tổ chức khác đã thử 4GL và vô cùng thất vọng với kết quả.

Một lý do cho sự không nhất quán này là không chắc rằng một 4GL sẽ phù hợp với tất cả các sản phẩm. Ngược lại, điều quan trọng là phải chọn đúng 4GL cho sản phẩm cụ thể. Ví dụ: Playtex đã sử dụng Cơ sở Phát triển Ứng dụng (ADF) của IBM và báo cáo năng suất tăng từ 80 lên 1 so với COBOL. Bất chấp kết quả ấn tượng này, Playtex sau đó đã quay trở lại COBOL vì các sản phẩm mà ban quản lý cho là kém phù hợp hơn với ADF [Martin, 1985].

Lý do thứ hai cho những kết quả không nhất quán này là nhiều 4GL được hỗ trợ bởi các bàn làm việc và môi trường CASE mạnh mẽ (Phần 5.7). Bàn làm việc và môi trường CASE có thể vừa là điểm mạnh vừa là điểm yếu. Như đã giải thích trong Phần 5.12, không nên giới thiệu CASE quy mô lớn trong một tổ chức có mức độ trưởng thành thấp. Lý do là mục đích của bàn làm việc hoặc môi trường CASE là để hỗ trợ quy trình phần mềm. Một tổ chức ở mức 1 không có quy trình phần mềm. Nếu tại thời điểm này, CASE được giới thiệu như một phần của quá trình chuyển đổi sang 4GL, thì điều này áp đặt một quy trình lên một tổ chức chưa sẵn sàng cho bất kỳ loại quy trình nào. Những hậu quả thông thường tốt nhất là không đạt yêu cầu và có thể là thảm họa. Trên thực tế, một số lỗi 4GL được báo cáo có thể được quy cho tác động của môi trường CASE liên quan hơn là do chính 4GL.

Thái độ của 43 tổ chức đối với 4GL được báo cáo trong [Guimaraes, 1985]. Nghiên cứu này cho thấy việc sử dụng 4GL làm giảm sự thất vọng của người dùng vì bộ phận xử lý dữ liệu có thể phản hồi nhanh hơn khi người dùng cần thông tin được trích xuất từ

cơ sở dữ liệu của tổ chức. Tuy nhiên, cũng có một số vấn đề. Một số 4GL tỏ ra chậm và kém hiệu quả với thời gian phản hồi lâu. Một sản phẩm tiêu thụ 60 phần trăm chu kỳ CPU trên máy tính lớn IBM 4331, trong khi hỗ trợ tối đa 12 người dùng đồng thời. Nhìn chung, 28 tổ chức đã sử dụng 4GL trong hơn 3 năm cảm thấy rằng lợi ích lớn hơn chi phí.

Không ai 4GL thống trị thị trường phần mềm. Thay vào đó, có hàng trăm 4GL; một số trong số chúng, bao gồm DB2, Oracle và PowerBuilder, có các nhóm người dùng khá lớn. Sự phổ biến rộng rãi của 4GL là bằng chứng nữa cho thấy cần phải cẩn thận trong việc lựa chọn 4GL chính xác. Tất nhiên, ít tổ chức có đủ khả năng hỗ trợ nhiều hơn một 4GL. Khi 4GL đã được chọn và sử dụng, tổ chức phải sử dụng 4GL đó cho các sản phẩm tiếp theo hoặc sử dụng lại ngôn ngữ được sử dụng trước khi 4GL được giới thiệu.

Mặc dù có thể tăng năng suất, nhưng vẫn có thể gặp nguy hiểm khi sử dụng 4GL sai cách. Nhiều tổ chức hiện đang có một lượng lớn sản phẩm tồn đọng cần phát triển và một danh sách dài các nhiệm vụ bảo trì sau khi giao hàng cần được thực hiện. Mục tiêu thiết kế của nhiều 4GL là lập trình người dùng cuối, nghĩa là lập trình bởi người sẽ sử dụng sản phẩm. Ví dụ, trước khi 4GL ra đời, người quản lý đầu tư của một công ty bảo hiểm sẽ hỏi người quản lý xử lý dữ liệu về một sản phẩm có thể hiển thị một số thông tin nhất định về danh mục đầu tư trái phiếu. Sau đó, người quản lý đầu tư sẽ đợi một năm hoặc lâu hơn để nhóm xử lý dữ liệu tìm thời gian phát triển sản phẩm. Một 4GL được mong muốn sẽ sử dụng đơn giản đến mức người quản lý đầu tư, trước đây chưa được đào tạo về lập trình, có thể triển khai sản phẩm mong muốn mà không cần trợ giúp. Lập trình dành cho người dùng cuối nhằm giúp giảm bớt tồn đọng phát triển, khiến các chuyên gia phải duy trì các sản phẩm hiện có.

Trong thực tế, lập trình người dùng cuối có thể nguy hiểm. Đầu tiên, hãy xem xét tình huống khi tất cả quá trình phát triển sản phẩm được thực hiện bởi các chuyên gia máy tính. Các chuyên gia máy tính được đào tạo để không tin tưởng vào đầu ra của máy tính. Xét cho cùng, có lẽ ít hơn 1 phần trăm của tất cả đầu ra trong quá trình phát triển sản phẩm là đúng. Mặt khác, người dùng được yêu cầu tin tưởng vào tất cả đầu ra của máy tính, bởi vì không có sản phẩm nào được giao cho người dùng cho đến khi nó không có lỗi. Bây giờ hãy xem xét tình huống khi chương trình của người dùng cuối được khuyến khích. Khi một người dùng thiếu kinh nghiệm lập trình triển khai mã với 4GL thân thiện với người dùng, không theo thủ tục, xu hướng tự nhiên là người dùng đó sẽ tin vào kết quả đầu ra. Rốt cuộc, trong nhiều năm, người dùng đã được hướng dẫn tin tưởng vào đầu ra của máy tính. Do đó, nhiều quyết định kinh doanh đã được dựa trên dữ liệu được tạo bởi mã người dùng cuối không chính xác. Trong một số trường hợp, tính thân thiện với người dùng của một số 4GL nhất định đã dẫn đến những thảm họa tài chính.

Một nguy cơ tiềm ẩn khác nằm ở xu hướng, trong một số tổ chức, cho phép người dùng triển khai các sản phẩm 4GL cập nhật cơ sở dữ liệu của tổ chức. Một lỗi lập trình do người dùng thực hiện cuối cùng có thể dẫn đến hỏng toàn bộ cơ sở dữ liệu. Bài học rất rõ ràng: Việc lập trình bởi những người dùng thiếu kinh nghiệm hoặc chưa được đào tạo đầy đủ có thể cực kỳ nguy hiểm, nếu không muốn nói là gây tử vong, đối với sức khỏe tài chính của một công ty.

Sự lựa chọn cuối cùng của 4GL được thực hiện bởi ban quản lý. Khi đưa ra quyết định như vậy, ban quản lý nên được hướng dẫn bởi nhiều câu chuyện thành công do việc sử dụng 4GL. Đồng thời, ban quản lý nên phân tích cẩn thận các lỗi gây ra do sử dụng 4GL không phù hợp, do giới thiệu sớm môi trường CASE và do quản lý quá trình phát triển kém. Ví dụ, một nguyên nhân thất bại phổ biến là do bỏ qua việc đào tạo nhóm phát triển kỹ lưỡng về mọi khía cạnh của 4GL, bao gồm cả lý thuyết cơ sở dữ liệu quan hệ [Date, 2003] khi thích hợp. Ban quản lý nên nghiên cứu cả những thành công và thất bại trong lĩnh vực ứng dụng cụ thể và học hỏi từ những sai lầm trong quá khứ. Chọn đúng 4GL có thể tạo ra sự khác biệt giữa thành công lớn và thất bại thảm hại.

Khi đã quyết định về ngôn ngữ triển khai, vấn đề tiếp theo là làm thế nào các nguyên tắc kỹ thuật phần mềm có thể dẫn đến mã chất lượng tốt hơn.

* 1. Good Programming Practice 15.3 Thực hành lập trình tốt

Many recommendations on good coding style are language specific. For example, sugges- tions regarding use of COBOL 88-level entries or parentheses in Lisp are of little interest to programmers implementing a product in Java. In contrast, recommendations regarding language-independent **good programming practice** are now given.

Nhiều khuyến nghị về phong cách mã hóa tốt là ngôn ngữ cụ thể. Ví dụ, các đề xuất liên quan đến việc sử dụng các mục nhập COBOL 88 cấp hoặc dấu ngoặc đơn trong Lisp ít được các lập trình viên triển khai một sản phẩm trong Java quan tâm. Ngược lại, các khuyến nghị liên quan đến thực hành lập trình tốt không phụ thuộc vào ngôn ngữ hiện đã được đưa ra.

* + 1. Use of Consistent and Meaningful Variable Names 15.3.1 Sử dụng tên biến nhất quán và có ý nghĩa

As stated in Chapter 1, on average at least two-thirds of a software budget is devoted to postdelivery maintenance. This implies that the programmer developing a code artifact is merely the first of many who will work on that code artifact. It is counterproductive for a programmer to give names to variables that are meaningful to only that programmer; within the context of software engineering, the term **meaningful variable names** means “meaningful from the viewpoint of future maintenance programmers.” This point is ampli- fied in Just in Case You Wanted to Know Box 15.3.

In addition to the use of meaningful variable names, it is equally essential that **consistent variable names** be chosen. For example, the following four variables are declared in a code artifact: averageFreq, frequencyMaximum, minFr, and frqncyTotl. A mainte- nance programmer who is trying to understand the code has to know if freq, frequency, fr, and frqncy all refer to the same thing. If yes, then the identical word should be used,



**Just in Case You Wanted to Know**

Chapter 15 *Implementation* **505**

**Box 15.4**

There are two explanations for the term **Hungarian Naming Conventions**. First, these conventions were invented by Charles Simonyi, who was born in Hungary. Second, it gen- erally is agreed that, to the uninitiated, programs with variable names conforming to the conventions are about as easy to read as Hungarian. Nevertheless, organizations (such as Microsoft) that use them claim that they enhance code readability for those with experi- ence in the Hungarian Naming Conventions.

preferably frequency, although freq or frqncy is marginally acceptable; fr is not. But if one or more variable names refer to a different quantity, then a totally different name, such as rate, should be used. Conversely, do not use two different names to denote the identical concept; for example, both average and mean should not be used in the same program.

A second aspect of consistency is the ordering of the components of variable names. For example, if one variable is named frequencyMaximum, then the name minimum- Frequency would be confusing; it should be frequencyMinimum. To make the code clear and unambiguous for future maintenance programmers, the four variables listed previously should be named frequencyAverage, frequencyMaximum, frequencyMinimum, and frequencyTotal, respectively. Alternatively, the frequency component can appear at the end of all four variable names, yielding the variable names averageFrequency, maximumFrequency, minimumFrequency, and totalFrequency. It clearly does not matter which of the two sets is chosen; what is important is that all the names be from one set or the other.

A number of different naming conventions have been put forward that are intended to make it easier to understand the code. The idea is that the name of a variable should in- corporate type information. For example, ptrChTmp might denote a temporary variable (Tmp) of type pointer (ptr) to an character (Ch). The best known of such schemes are the Hungarian Naming Conventions [Klunder, 1988]. (If you want to know why they are called Hungarian, see Just in Case You Wanted to Know Box 15.4.) One drawback of many such schemes is that the effectiveness of code inspections (Section 15.14) can be reduced when participants are unable to pronounce the names of variables. It is extremely frustrating to have to spell out variable names, letter by letter.

Như đã nêu trong Chương 1, trung bình ít nhất hai phần ba ngân sách phần mềm được dành cho việc bảo trì sau khi chuyển giao. Điều này ngụ ý rằng lập trình viên đang phát triển một tạo phẩm mã chỉ là người đầu tiên trong số nhiều người sẽ làm việc trên tạo phẩm mã đó. Việc một lập trình viên đặt tên cho các biến chỉ có ý nghĩa đối với lập trình viên đó là phản tác dụng; trong bối cảnh công nghệ phần mềm, thuật ngữ tên biến có ý nghĩa có nghĩa là “có ý nghĩa từ quan điểm của các lập trình viên bảo trì trong tương lai.” Điểm này được khuếch đại trong Just in Case You Want to Know Hộp 15.3.

Ngoài việc sử dụng các tên biến có ý nghĩa, điều quan trọng không kém là phải chọn các tên biến nhất quán. Ví dụ: bốn biến sau đây được khai báo trong một tạo phẩm mã: AverageFreq, frequencyMaximum, minFr và frqncyTotl. Một lập trình viên bảo trì đang cố gắng hiểu mã phải biết liệu tần số, tần suất, fr và frqncy đều có đề cập đến cùng một thứ hay không. Nếu có, thì từ giống hệt nhau nên được sử dụng, tốt nhất là tần suất, mặc dù freq hoặc frqncy có thể chấp nhận được một chút; không phải là. Nhưng nếu một hoặc nhiều tên biến đề cập đến một đại lượng khác, thì nên sử dụng một tên hoàn toàn khác, chẳng hạn như tỷ lệ. Ngược lại, không được dùng hai tên khác nhau để chỉ cùng một khái niệm; ví dụ: không nên sử dụng cả giá trị trung bình và giá trị trung bình trong cùng một chương trình.

Khía cạnh thứ hai của tính nhất quán là thứ tự các thành phần của tên biến. Ví dụ: nếu một biến được đặt tên là frequencyMaximum, thì tên minimum- Tần suất sẽ gây nhầm lẫn; nó phải là tần suất tối thiểu. Để làm cho mã rõ ràng và dễ hiểu đối với các lập trình viên bảo trì trong tương lai, bốn biến được liệt kê trước đây phải được đặt tên tương ứng là frequencyAverage, frequencyMaximum, frequencyMinimum và frequencyTotal. Ngoài ra, thành phần tần số có thể xuất hiện ở cuối cả bốn tên biến, mang lại các tên biến là tần số trung bình, tần số tối đa, tần số tối thiểu và tần số tổng. Rõ ràng là không quan trọng bộ nào trong hai bộ được chọn; điều quan trọng là tất cả các tên đều thuộc bộ này hay bộ kia.

Một số quy ước đặt tên khác nhau đã được đưa ra nhằm mục đích làm cho mã dễ hiểu hơn. Ý tưởng là tên của một biến nên thông tin loại trong công ty. Ví dụ: ptrChTmp có thể biểu thị một biến tạm thời (Tmp) của con trỏ kiểu (ptr) thành một ký tự (Ch). Kế hoạch được biết đến nhiều nhất là Quy ước đặt tên Hungary [Klunder, 1988]. (Nếu bạn muốn biết tại sao chúng được gọi là tiếng Hungary, hãy xem Đề phòng trường hợp bạn muốn biết Hộp 15.4.) Một nhược điểm của nhiều kế hoạch như vậy là hiệu quả của việc kiểm tra mã (Phần 15.14) có thể giảm khi người tham gia không thể phát âm tên của các biến. Thật khó chịu khi phải đánh vần các tên biến, từng chữ một.

* + 1. The Issue of Self-Documenting Code 15.3.2 Vấn đề Mã tự ghi

When asked why their code contains no comments whatsoever, programmers often proudly reply, “I write **self-documenting code**.” The implication is that their variable names are chosen so carefully and their code crafted so exquisitely that there is no need for comments. Self-documenting code does exist, but it is exceedingly rare. Instead, the usual scenario is that the programmer appreciates every nuance of the code at the time the code artifact is implemented. It is conceivable that the programmer uses the same style for every code artifact and that in 5 years’ time, the code still is crystal clear in every respect to the original programmer. Unfortunately, this is irrelevant. The important point is whether the code artifact can be understood easily and unambiguously by all the other programmers who have to read it, starting with the software quality assurance group and including a number of different postdelivery maintenance programmers. The problem becomes more acute in the light of the unfortunate practice of assigning postdelivery

maintenance tasks to inexperienced programmers and not supervising them closely. The undocumented code of the artifact may be only partially comprehensible to an experi- enced programmer. How much worse, then, is the situation when the maintenance pro- grammer is inexperienced.

To see the sorts of problems that can arise, consider the variable xCoordinateOfPosition- OfRobotArm. Such a variable name undoubtedly is self-documenting in every sense of the word, but few programmers are prepared to use a 31-character variable name, especially if that name is used frequently. Instead, a shorter name is used, xCoord, for example. The rea- soning behind this is that if the entire code artifact deals with the movement of the arm of a robot, xCoord can refer only to the *x* coordinate of the position of the arm of the robot. Although that argument holds water within the context of the development process, it is not necessarily true for postdelivery maintenance. The maintenance programmer may not have sufficient knowledge of the product as a whole to realize that, within this code arti- fact, xCoord refers to the arm of the robot or may not have the necessary documentation to understand the workings of the code artifact. The way to avoid this sort of problem is to insist that every variable name be explained at the beginning of the code artifact, in the **prologue comments**. If this rule is followed, the maintenance programmer quickly will understand that variable xCoord is used for the *x* coordinate of the position of the robot arm.

Prologue comments are mandatory in every code artifact. The minimum information that must be provided at the top of every code artifact is listed in Figure 15.1.

Even if a code artifact is clearly written, it is unreasonable to expect someone to have to read every line to understand what the code artifact does and how it does it. Prologue comments make it easy for others to understand the key points. Only a member of the SQA group or a maintenance programmer modifying a specific code artifact should be expected to have to read every line of that code artifact.

**FIGURE 15.1**

The name of the code artifact

A brief description of what the code artifact does The programmer’s name

The date the code artifact was coded The date the code artifact was approved

The name of the person who approved the code artifact The arguments of the code artifact

A list of the name of each variable of the code artifact, preferably in alphabetical order, and a brief description of its use

The names of any files accessed by this code artifact The names of any files changed by this code artifact Input–output, if any

Error-handling capabilities

The name of the file containing test data (to be used later for regression testing)

A list of each modification made to the code artifact, the date the modification was made, and who approved the modification

Any known faults

Minimal

prologue

comments for a code artifact.

In addition to prologue comments, inline comments should be inserted into the code to assist maintenance programmers in understanding that code. It has been suggested that inline comments should be used only when the code is implemented in a nonobvious way or uses some subtle aspect of the language. On the contrary, confusing code should be reimplemented in a clearer way. Inline comments are a means of helping maintenance pro- grammers and should not be used to promote or excuse poor programming practice.

Khi được hỏi tại sao mã của họ không chứa bất kỳ bình luận nào, các lập trình viên thường tự hào trả lời: “Tôi viết mã tự ghi lại.” Ngụ ý là tên biến của chúng được chọn rất cẩn thận và mã của chúng được chế tạo tinh xảo đến mức không cần bình luận. Có tồn tại mã tự tạo tài liệu, nhưng nó cực kỳ hiếm. Thay vào đó, kịch bản thông thường là lập trình viên đánh giá cao mọi sắc thái của mã tại thời điểm tạo phẩm mã được triển khai. Có thể hình dung rằng lập trình viên sử dụng cùng một kiểu cho mọi mã tạo tác và trong thời gian 5 năm, mã vẫn rõ ràng về mọi mặt so với lập trình viên ban đầu. Thật không may, điều này là không liên quan. Điểm quan trọng là liệu tạo tác mã có thể được hiểu một cách dễ dàng và rõ ràng bởi tất cả các lập trình viên khác, những người phải đọc nó hay không, bắt đầu với nhóm đảm bảo chất lượng phần mềm và bao gồm một số lập trình viên bảo trì sau chuyển giao khác nhau. Vấn đề trở nên nghiêm trọng hơn do thực tế đáng tiếc là chỉ định sau khi giao hàng

nhiệm vụ bảo trì cho các lập trình viên thiếu kinh nghiệm và không giám sát họ chặt chẽ. Mã không có tài liệu của tạo tác có thể chỉ được hiểu một phần đối với một lập trình viên có kinh nghiệm. Sau đó, tình huống tồi tệ hơn biết bao khi người lập trình bảo trì thiếu kinh nghiệm.

Để xem các loại vấn đề có thể phát sinh, hãy xem xét biến xCoordinateOfPosition- OfRobotArm. Một tên biến như vậy chắc chắn là tự ghi lại theo mọi nghĩa của từ này, nhưng rất ít lập trình viên sẵn sàng sử dụng tên biến gồm 31 ký tự, đặc biệt nếu tên đó được sử dụng thường xuyên. Thay vào đó, một tên ngắn hơn được sử dụng, chẳng hạn như xCoord. Lý do đằng sau điều này là nếu toàn bộ tạo tác mã liên quan đến chuyển động của cánh tay rô bốt, thì xCoord chỉ có thể tham chiếu đến tọa độ x của vị trí cánh tay rô bốt. Mặc dù lập luận đó đúng trong bối cảnh của quá trình phát triển, nhưng nó không nhất thiết đúng đối với việc bảo trì sau khi giao hàng. Người lập trình bảo trì có thể không có đủ kiến ​​thức về toàn bộ sản phẩm để nhận ra rằng, trong tạo phẩm mã này, xCoord đề cập đến cánh tay của rô-bốt hoặc có thể không có tài liệu cần thiết để hiểu hoạt động của tạo phẩm mã. Cách để tránh loại vấn đề này là nhấn mạnh rằng mọi tên biến đều được giải thích ở phần đầu của tạo phẩm mã, trong các nhận xét mở đầu. Nếu tuân theo quy tắc này, người lập trình bảo trì sẽ nhanh chóng hiểu rằng biến xCoord được sử dụng cho tọa độ x của vị trí cánh tay rô-bốt.

Nhận xét mở đầu là bắt buộc trong mọi tạo tác mã. Thông tin tối thiểu phải được cung cấp ở đầu mỗi tạo phẩm mã được liệt kê trong Hình 15.1.

Ngay cả khi một mã tạo tác được viết rõ ràng, thì cũng không hợp lý khi mong đợi ai đó phải đọc từng dòng để hiểu mã tạo tác đó làm gì và nó làm như thế nào. Lời bình mở đầu giúp người khác dễ dàng hiểu được những điểm chính. Chỉ một thành viên của nhóm SQA hoặc một lập trình viên bảo trì sửa đổi một tạo phẩm mã cụ thể mới được phép đọc mọi dòng của tạo phẩm mã đó.

Ngoài các nhận xét mở đầu, các nhận xét nội tuyến nên được chèn vào mã để hỗ trợ các lập trình viên bảo trì hiểu mã đó. Có ý kiến ​​cho rằng chỉ nên sử dụng nhận xét nội tuyến khi mã được triển khai theo cách không rõ ràng hoặc sử dụng một số khía cạnh tinh vi của ngôn ngữ. Ngược lại, mã khó hiểu nên được triển khai lại theo cách rõ ràng hơn. Nhận xét nội tuyến là một phương tiện trợ giúp các lập trình viên bảo trì và không nên được sử dụng để thúc đẩy hoặc bào chữa cho việc thực hành lập trình kém

* + 1. Use of Parameters 15.3.3 Sử dụng Tham số

There are very few genuine constants, that is, variables whose values *never* change. For instance, satellite photographs have caused changes to be made in submarine navigation systems incorporating the latitude and longitude of Pearl Harbor, Hawaii, to reflect more accurate geographic data regarding the exact location of Pearl Harbor. To take another example, sales tax is not a genuine constant; legislators tend to change the sales tax rate from time to time. Suppose that the sales tax rate currently is 6.0 percent. If the value 6.0 has been hard coded in a number of code artifacts of a product, then changing the product is a major exercise, with the likely outcome of one or two instances of the “constant” 6.0 being overlooked and, perhaps, changing an unrelated 6.0 by mistake. A better solution is a C++ declaration such as

or, in Java,

**const float** salesTaxRate = 6.0;

**public static final float** salesTaxRate = (**float**) 6.0;

Then, wherever the value of the sales tax rate is needed, the constant salesTaxRate should be used and not the number 6.0. If the sales tax rate changes, then only the line containing the value of salesTaxRate need be altered using an editor. Better still, the value of the sales tax rate should be read in from a parameter file at the beginning of the run. All such appar- ent constants should be treated as parameters. If a value should change for any reason, this change can be implemented quickly and effectively.

Có rất ít hằng số chính hãng, nghĩa là các biến có giá trị không bao giờ thay đổi. Ví dụ, ảnh vệ tinh đã gây ra những thay đổi được thực hiện trong hệ thống định vị tàu ngầm kết hợp vĩ độ và kinh độ của Trân Châu Cảng, Hawaii, để phản ánh dữ liệu địa lý chính xác hơn về vị trí chính xác của Trân Châu Cảng. Lấy một ví dụ khác, thuế bán hàng không phải là một hằng số thực sự; các nhà lập pháp có xu hướng thay đổi thuế suất bán hàng theo thời gian. Giả sử rằng thuế suất doanh thu hiện tại là 6,0 phần trăm. Nếu giá trị 6.0 đã được mã hóa cứng trong một số tạo phẩm mã của sản phẩm, thì việc thay đổi sản phẩm là một bài tập lớn, với kết quả có khả năng là một hoặc hai trường hợp của “hằng số” 6.0 bị bỏ qua và có lẽ, thay đổi một không liên quan 6.0 do nhầm lẫn. Một giải pháp tốt hơn là khai báo C++ chẳng hạn như

hoặc, trong Java,

const float salesTaxRate = 6.0;

public tĩnh cuối cùng float salesTaxRate = (float) 6.0;

Sau đó, bất cứ nơi nào giá trị của thuế suất doanh thu là cần thiết, thì nên sử dụng SalesTaxRate không đổi và không phải là số 6.0. Nếu thuế suất thuế bán hàng thay đổi, thì chỉ dòng chứa giá trị của salesTaxRate cần được thay đổi bằng trình chỉnh sửa. Tốt hơn nữa, giá trị của thuế suất bán hàng nên được đọc từ tệp tham số khi bắt đầu chạy. Tất cả các hằng số rõ ràng như vậy nên được coi là tham số. Nếu một giá trị thay đổi vì bất kỳ lý do gì, thay đổi này có thể được thực hiện nhanh chóng và hiệu quả.

* + 1. Code Layout for Increased Readability 15.3.4 Bố cục mã để tăng khả năng đọc

It is relatively simple to make a code artifact easy to read. For example, no more than one statement should appear on a line, even though many programming languages permit more than one. Indentation is perhaps the most important technique for increasing readability. Just imagine how difficult it would be to read the code examples in Chapter 7 if indentation had not been used to assist in understanding the code. In C++ or Java, indentation can be used to connect corresponding { . . . } pairs. Indentation also shows which statements belong in a given block. In fact, correct indentation is too important to be left to humans. Instead, as described in Section 5.8, CASE tools should be used to ensure that indentation is done correctly.

Another useful aid is blank lines. Methods should be separated by blank lines; in addi- tion, it often is helpful to break up large blocks of code with blank lines. The extra “white space” makes the code easier to read and, hence, comprehend.

Nó tương đối đơn giản để làm cho một tạo phẩm mã dễ đọc. Ví dụ, không nên có nhiều hơn một câu lệnh xuất hiện trên một dòng, mặc dù nhiều ngôn ngữ lập trình cho phép nhiều hơn một câu lệnh. Thụt lề có lẽ là kỹ thuật quan trọng nhất để tăng khả năng đọc. Chỉ cần tưởng tượng sẽ khó đọc mã ví dụ trong Chương 7 như thế nào nếu không sử dụng thụt đầu dòng để hỗ trợ việc hiểu mã. Trong C ++ hoặc Java, thụt đầu dòng có thể được sử dụng để kết nối { . . . } cặp. Việc thụt lề cũng cho biết câu lệnh nào thuộc về một khối nhất định. Trên thực tế, việc thụt đầu dòng chính xác là quá quan trọng để giao cho con người. Thay vào đó, như được mô tả trong Phần 5.8, các công cụ CASE nên được sử dụng để đảm bảo rằng việc thụt đầu dòng được thực hiện chính xác.

Một trợ giúp hữu ích khác là các dòng trống. Các phương thức nên được phân tách bằng các dòng trống; ngoài ra, việc chia nhỏ các khối mã lớn bằng các dòng trống thường rất hữu ích. “Khoảng trắng” bổ sung làm cho mã dễ đọc hơn và do đó, dễ hiểu hơn.

* + 1. Nested if Statements 15.3.5 Câu lệnh if lồng nhau

Consider the following example. A map consists of two squares, as shown in Figure 15.2. It is required to write code to determine whether a point on the Earth’s surface lies in

**FIGURE 15.2**

Coordinates for a map.

latitude

90°

60°

map- Square- 2

map- Square- 1

30°

90° 120°

150°

180°

longitude

**FIGURE 15.3**

Badly formatted nested **if** statements.

**if** (latitude > 30 && longitude > 120) {**if** (latitude <= 60 && longitude <= 150) mapSquareNo = 1; **else if** (latitude <= 90 && longitude <= 150) mapSquareNo = 2 **else** *print* “Not on the map”;} **else** *print* “Not on the map”;

**FIGURE 15.4**

Well-formatted but badly constructed nested **if** statements.

**if** (latitude > 30 && longitude > 120)

{

**if** (latitude <= 60 && longitude <= 150) mapSquareNo = 1;

**else**

**if** (latitude <= 90 && longitude <= 150) mapSquareNo = 2;

**else**

*print* ”Not on the map”;

}

**else**

*print* ”Not on the map”;

**FIGURE 15.5**

Acceptably nested **if** statements.

**if** (longitude > 120 && longitude <= 150 && latitude > 30 && latitude <= 60) mapSquareNo = 1;

**else**

**if** (longitude > 120 && longitude <= 150 && latitude > 60 && latitude <= 90) mapSquareNo = 2;

**else**

*print* ”Not on the map”;

mapSquare1, mapSquare2, or not on the map at all. The solution of Figure 15.3 is so badly formatted that it is incomprehensible. A properly formatted version appears in Figure 15.4. Notwithstanding this, the combination of **if-if** and **if-else-if** constructs is so complex that it is difficult to check whether the code fragment is correct. This is fixed in Figure 15.5. When faced with complex code containing the **if-if** construct, one way to simplify it is to use the fact that the **if-if** combination

**if** <*condition* 1>

**if** <*condition* 2>

is equivalent to the single condition

**if** <*condition* 1> **and** <*condition* 2>

provided that <*condition* 2> is defined even if <*condition* 1> does not hold. For example,

<*condition* 1> might check that a pointer is not null and, if so, then <*condition* 2> can use that pointer. (This problem does not arise in Java or C++. The && operator is defined such that if <*condition* 1> is false, then <*condition* 2> is not evaluated—see Problems 15.9 and 15.10.)

Another problem with the **if-if** construct is that nesting **if** statements too deeply leads to code that can be difficult to read. As a rule of thumb, **if** statements nested to a depth greater than three is poor programming practice and should be avoided.

Hãy xem xét ví dụ sau. Một bản đồ bao gồm hai ô vuông, như trong Hình 15.2. Cần phải viết mã để xác định xem một điểm trên bề mặt Trái đất có nằm trong không với điều kiện là <điều kiện 2> được xác định ngay cả khi <điều kiện 1> không đúng. Ví dụ,

<điều kiện 1> có thể kiểm tra xem một con trỏ có phải là null không và nếu vậy thì <điều kiện 2> có thể sử dụng con trỏ đó. (Vấn đề này không phát sinh trong Java hoặc C++. Toán tử && được định nghĩa sao cho nếu <điều kiện 1> là sai, thì <điều kiện 2> không được đánh giá—xem Vấn đề 15.9 và 15.10.)

Một vấn đề khác với cấu trúc if-if là các câu lệnh if được lồng quá sâu dẫn đến mã có thể khó đọc. Theo nguyên tắc thông thường, nếu các câu lệnh được lồng vào nhau ở độ sâu lớn hơn ba là cách thực hành lập trình kém và nên tránh.

* 1. Coding Standards 15.4 Tiêu chuẩn mã hóa

**Coding standards** can be both a blessing and a curse. Section 7.2.1 pointed out that modules with coincidental cohesion (that is, modules that perform multiple, completely unrelated operations) generally arise as a consequence of rules such as, “Every module will consist of between 35 and 50 executable statements.” Instead of stating a rule in such a dogmatic fashion, a better formulation is, “Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements.” The point is that no coding standard can be applicable under all possible circumstances.

Coding standards imposed from above tend to be ignored. As mentioned previously, a useful rule of thumb is that **if** statements should not be nested to a depth greater than three. If programmers are shown examples of unreadable code resulting from nesting **if** state- ments too deeply, then it is likely that they will conform to such a regulation. But they are unlikely to adhere to a list of coding rules imposed on them with no discussion or explana- tion. Furthermore, such standards are likely to lead to friction between programmers and their managers.

In addition, unless a coding standard can be checked by machine, it is going to either waste a lot of the SQA group’s time or simply be ignored by the programmers and SQA group alike. On the other hand, consider the following rules (see Problems 15.11–15.13):

* Nesting of **if** statements should not exceed a depth of three, except with prior approval from the team leader.
* Modules should consist of between 35 and 50 statements, except with prior approval from the team leader.
* The use of **goto** statements should be avoided. However, with prior approval from the team leader, a forward **goto** may be used for error handling.

Such rules may be checked by machine, provided some mechanism is set up for capturing the data relating to permission to deviate from the standard.

The aim of coding standards is to make maintenance easier. However, if the effect of a standard is to make the life of software developers difficult, then such a standard should be modified, even in the middle of a project. Overly restrictive coding standards are counter- productive, in that the quality of software production inevitably must suffer if programmers have to develop software within such a framework. On the other hand, standards such as those just listed regarding nesting of **if** statements, module size, and **goto** statements, coupled with a mechanism for deviating from those standards, can lead to improved soft- ware quality, which, after all, is a major goal of software engineering.

Các tiêu chuẩn mã hóa có thể vừa là một may mắn vừa là một lời nguyền. Mục 7.2.1 đã chỉ ra rằng các mô-đun có sự gắn kết ngẫu nhiên (nghĩa là các mô-đun thực hiện nhiều thao tác hoàn toàn không liên quan) thường phát sinh do các quy tắc như “Mỗi mô-đun sẽ bao gồm từ 35 đến 50 câu lệnh thực thi”. Thay vì nêu một quy tắc theo kiểu giáo điều như vậy, một công thức tốt hơn là, “Các lập trình viên nên hỏi ý kiến ​​người quản lý của họ trước khi xây dựng một mô-đun có ít hơn 35 hoặc nhiều hơn 50 câu lệnh có thể thực thi được.” Vấn đề là không có tiêu chuẩn mã hóa nào có thể được áp dụng trong mọi trường hợp có thể.

Tiêu chuẩn mã hóa áp đặt từ trên có xu hướng bị bỏ qua. Như đã đề cập trước đây, một nguyên tắc ngón tay cái hữu ích là các câu lệnh if không nên được lồng vào nhau ở độ sâu lớn hơn ba. Nếu các lập trình viên được cho xem các ví dụ về mã không thể đọc được do lồng các câu lệnh if quá sâu, thì có khả năng là họ sẽ tuân theo quy định đó. Nhưng họ không có khả năng tuân thủ danh sách các quy tắc mã hóa áp đặt cho họ mà không cần thảo luận hoặc giải thích. Hơn nữa, các tiêu chuẩn như vậy có khả năng dẫn đến xích mích giữa các lập trình viên và người quản lý của họ.

Ngoài ra, trừ khi một tiêu chuẩn viết mã có thể được kiểm tra bằng máy, nếu không thì nó sẽ lãng phí rất nhiều thời gian của nhóm SQA hoặc đơn giản là bị các lập trình viên cũng như nhóm SQA bỏ qua. Mặt khác, hãy xem xét các quy tắc sau (xem Bài toán 15.11–15.13):

• Việc lồng các câu lệnh if không được vượt quá độ sâu của ba, trừ khi có sự chấp thuận trước của trưởng nhóm.

• Các mô-đun nên bao gồm từ 35 đến 50 câu, trừ khi có sự chấp thuận trước của trưởng nhóm.

• Nên tránh sử dụng câu lệnh goto. Tuy nhiên, với sự chấp thuận trước của trưởng nhóm, goto chuyển tiếp có thể được sử dụng để xử lý lỗi.

Các quy tắc như vậy có thể được kiểm tra bằng máy, với điều kiện là một số cơ chế được thiết lập để thu thập dữ liệu liên quan đến quyền đi chệch khỏi tiêu chuẩn.

Mục đích của các tiêu chuẩn mã hóa là làm cho việc bảo trì dễ dàng hơn. Tuy nhiên, nếu tác động của một tiêu chuẩn là làm cho cuộc sống của các nhà phát triển phần mềm trở nên khó khăn, thì tiêu chuẩn đó nên được sửa đổi, ngay cả khi đang ở giữa dự án. Các tiêu chuẩn mã hóa quá hạn chế sẽ phản tác dụng, ở chỗ chất lượng sản xuất phần mềm chắc chắn phải bị ảnh hưởng nếu các lập trình viên phải phát triển phần mềm trong một khuôn khổ như vậy. Mặt khác, các tiêu chuẩn như những tiêu chuẩn vừa được liệt kê liên quan đến việc lồng các câu lệnh if, kích thước mô-đun và câu lệnh goto, cùng với một cơ chế để đi chệch khỏi các tiêu chuẩn đó, có thể dẫn đến chất lượng phần mềm được cải thiện, mà xét cho cùng, đó là một mục tiêu chính của công nghệ phần mềm.

* 1. Code Reuse 15.5 Tái sử dụng mã

Reuse was presented in detail in Chapter 8. In fact, the material on reuse could have ap- peared virtually anywhere in this book, because artifacts from all workflows of the software process are reused, including portions of specifications, contracts, plans, designs, and code artifacts. That is why the material on reuse was put into the first part of the book, rather than tying it to one or another specific workflow. In particular, it was important that the material on reuse not be presented in this chapter to underline the fact that, even though reuse of code is by far the most common form of reuse, more than just code can be reused.

Tái sử dụng đã được trình bày chi tiết trong Chương 8. Trên thực tế, tài liệu về tái sử dụng có thể xuất hiện hầu như ở bất cứ đâu trong cuốn sách này, bởi vì các tạo phẩm từ tất cả các luồng công việc của quy trình phần mềm đều được tái sử dụng, bao gồm các phần của đặc tả, hợp đồng, kế hoạch, thiết kế, và tạo tác mã. Đó là lý do tại sao tài liệu về tái sử dụng được đưa vào phần đầu tiên của cuốn sách, thay vì buộc nó vào quy trình làm việc cụ thể này hay quy trình cụ thể khác. Đặc biệt, điều quan trọng là tài liệu về tái sử dụng không được trình bày trong chương này để nhấn mạnh thực tế rằng, mặc dù việc tái sử dụng mã cho đến nay là hình thức tái sử dụng phổ biến nhất, nhưng không chỉ mã có thể được tái sử dụng.

* 1. Integration 15.6 Tích hợp

Consider the product depicted in Figure 15.6. One approach to **integration** of the product is to code and test each code artifact separately, link together all 13 code artifacts, and test the product as a whole. There are two difficulties with this approach. First, consider artifact

1. It cannot be tested on its own, because it calls artifacts b, c, and d. Therefore, to unit test artifact a, artifacts b, c, and d must be coded as **stubs**. In its simplest form, a stub is an empty artifact. A more effective stub prints a message such as artifact displayRadarPattern called. Best of all, a stub should return values corresponding to preplanned test cases.
2. Xét sản phẩm được mô tả trong Hình 15.6. Một cách tiếp cận để tích hợp sản phẩm là viết mã và kiểm tra riêng từng mã tạo phẩm, liên kết tất cả 13 mã tạo tác lại với nhau và kiểm tra toàn bộ sản phẩm. Có hai khó khăn với cách tiếp cận này. Đầu tiên, hãy xem xét tạo tác
3. một. Nó không thể tự kiểm tra, bởi vì nó gọi các tạo tác b, c và d. Do đó, để kiểm tra đơn vị tạo phẩm a, tạo tác b, c và d phải được mã hóa thành sơ khai. Ở dạng đơn giản nhất, sơ khai là một tạo phẩm rỗng. Sơ khai hiệu quả hơn sẽ in một thông báo chẳng hạn như artifact displayRadarPattern được gọi. Trên hết, một sơ khai nên trả về các giá trị tương ứng với các trường hợp thử nghiệm được lên kế hoạch trước.

**FIGURE 15.6**

a

b

c

d

e

f

g

h

i

j

k

l

m

A typical interconnection diagram.

Now consider artifact h. To test it on its own requires a **driver**, a code artifact that calls it one or more times, if possible checking the values returned by the artifact under test. Similarly, testing artifact d requires a driver and two stubs. Therefore, one problem that arises with separate implementation and integration is that effort has to be put into con- structing stubs and drivers, all of which are thrown away after unit testing is completed.

The second, and much more important, difficulty that arises when implementation is completed before integration starts is lack of fault isolation. If the product as a whole is tested against a specific test case and the product fails, then the fault could lie in any of the 13 code artifacts or 13 interfaces. In a large product with, say, 103 code artifacts and 108 interfaces, the fault might lie in no fewer than 211 places.

The solution to both difficulties is to combine unit and integration testing.

Bây giờ hãy xem xét tạo tác h. Để tự kiểm tra nó, cần có trình điều khiển, một tạo phẩm mã gọi nó một hoặc nhiều lần, nếu có thể, hãy kiểm tra các giá trị do tạo phẩm được kiểm tra trả về. Tương tự, thử nghiệm tạo phẩm d yêu cầu một trình điều khiển và hai sơ khai. Do đó, một vấn đề phát sinh với việc triển khai và tích hợp riêng biệt là nỗ lực phải được đưa vào việc xây dựng sơ khai và trình điều khiển, tất cả đều bị loại bỏ sau khi hoàn thành kiểm thử đơn vị.

Khó khăn thứ hai, và quan trọng hơn nhiều, phát sinh khi quá trình triển khai hoàn tất trước khi bắt đầu tích hợp là thiếu khả năng cách ly lỗi. Nếu toàn bộ sản phẩm được thử nghiệm dựa trên một trường hợp thử nghiệm cụ thể và sản phẩm không thành công, thì lỗi có thể nằm ở bất kỳ mã nào trong số 13 tạo phẩm mã hoặc 13 giao diện. Trong một sản phẩm lớn có 103 tạo phẩm mã và 108 giao diện, lỗi có thể nằm ở không ít hơn 211 vị trí.

Giải pháp cho cả hai khó khăn là kết hợp thử nghiệm đơn vị và tích hợp.

* + 1. Top-down Integration 15.6.1 Tích hợp từ trên xuống

In **top-down integration**, if code artifact mAbove sends a message to artifact mBelow, then mAbove is implemented and integrated before mBelow. Suppose that the product shown in Figure 15.6 is implemented and integrated top down. One possible top-down ordering is a, b, c, d, e, f, g, h, i, j, k, l, and m. First, artifact a is coded and tested with b, c, and d implemented as stubs. Next stub b is expanded into artifact b, linked to artifact a, and tested with artifact e implemented as a stub. Implementation and integration proceed in this way until all the artifacts have been integrated into the product. Another possible top-down ordering is a, b, e, h, c, d, f, i, g, j, k, l, and m. With this ordering, portions of the integration can proceed in parallel in the following way. After a has been coded and tested, one programmer can use artifact a to implement and integrate b, e, and h, while another programmer can use a to work in parallel on c, d, f, and i. Once d and f are completed, a third programmer can start work on g, j, k, l, and m.

Suppose that artifact a by itself executes correctly on a specific test case. However, when the same test data are submitted after b has been coded and integrated into the prod- uct, now consisting of artifacts a and b linked together, the test fails. The fault can be in one of two places, in artifact b or the interface between artifacts a and b. In general, when- ever a code artifact mNew is added to what has been tested so far and a previously suc- cessful test case fails, the fault almost certainly lies either in mNew or in the interface(s) between mNew and the rest of the product. Accordingly, top-down integration supports fault isolation.

Another strength of top-down integration is that major design flaws show up early. The artifacts of a product can be divided into two groups, logic artifacts and operational artifacts. The **logic artifacts** essentially incorporate the decision-making flow of control aspects of the product. The logic artifacts generally are those situated close to the root in the intercon- nection diagram. For example, in Figure 15.6, it is reasonable to expect artifacts a, b, c, d, and perhaps g and j to be logic artifacts. The **operational artifacts**, on the other hand, per- form the actual operations of the product. For example, an operational artifact may be named getLineFromTerminal or measureTemperatureOfReactorCore. The operational artifacts generally are found in the lower levels, close to the leaves, of the interconnection diagram. In Figure 15.6, artifacts e, f, h, i, k, l, and m are operational artifacts.

It is always important to code and test the logic artifacts before coding and testing the operational artifacts. This ensures that any major design faults show up early. Suppose the whole product is completed before a major fault is detected. Large parts of the product

have to be reimplemented, especially the logic artifacts that embody the flow of control. Many of the operational artifacts probably are reusable in the rebuilt product; for example, an artifact like getLineFromTerminal or measureTemperatureOfReactorCore is needed no matter how the product is restructured. However, the way the operational artifacts are connected to the other artifacts in the product may have to be changed, resulting in unnec- essary work. Therefore, the earlier a design fault is detected, the quicker and less costly it is to correct the product and get back on the development schedule. The order in which arti- facts are implemented and integrated using the top-down strategy essentially ensures that logic artifacts indeed are implemented and integrated before operational artifacts, because logic artifacts almost always are the ancestors of operational artifacts in the interconnection diagram. This is a major strength of top-down integration.

Nevertheless, top-down integration has a weakness: Potentially reusable code artifacts may not be adequately tested, as will be explained. Reuse of an artifact that is thought, incorrectly, to have been thoroughly tested is likely to be less cost-effective than writing that artifact from scratch, because the assumption that an artifact is correct can lead to wrong conclusions when the product fails. Instead of suspecting the insufficiently tested, reused artifact, the tester may think that the fault lies elsewhere, resulting in a waste of effort.

Logic artifacts are likely to be somewhat problem specific and hence unusable in another context. However, operational artifacts, particularly if they have informational cohesion (Section 7.2.7), probably are reusable in future products and, therefore, require thorough testing. Unfortunately, the operational artifacts generally are the lower-level code artifacts in the interconnection diagram and hence are not tested as frequently as the upper-level artifacts. For example, if there are 184 artifacts, the root artifact is tested 184 times, whereas the last artifact to be integrated into the product is tested only once. Top-down integration makes reuse a risky undertaking as a consequence of inadequate testing of operational artifacts.

The situation is exacerbated if the product is well designed; in fact, the better the design, the less thoroughly the artifacts are likely to be tested. To see this, consider an artifact computeSquareRoot. This artifact takes two arguments, a floating-point number x whose square root is to be determined and an errorFlag that is set to true if x is negative. Suppose further that computeSquareRoot is invoked by artifact a3 and that a3 contains the statement

**if** (x > = 0)

y = computeSquareRoot (x, errorFlag);

In other words, computeSquareRoot is never invoked unless the value of x is non- negative; therefore, the artifact can never be tested with negative values of x to see if it behaves correctly. The type of design where the calling artifact includes a safety check of this kind is referred to as **defensive programming**. As a result of defensive program- ming, subordinate operational artifacts are unlikely to be thoroughly tested if integrated top down. An alternative to defensive programming is the use of responsibility-driven design (Section 1.9). Here, the necessary safety checks are built into the invoked artifact, rather than the invoker. Another approach is the use of assertions in the invoked artifact (Section 6.5.3).

Trong tích hợp từ trên xuống, nếu tạo tác mã mAbove gửi thông báo tới tạo tác mBelow, thì mAbove được triển khai và tích hợp trước mBelow. Giả sử rằng sản phẩm trong Hình 15.6 được triển khai và tích hợp từ trên xuống. Một thứ tự từ trên xuống có thể có là a, b, c, d, e, f, g, h, i, j, k, l và m. Đầu tiên, tạo phẩm a được mã hóa và thử nghiệm với b, c và d được triển khai dưới dạng sơ khai. Sơ khai tiếp theo b được mở rộng thành tạo phẩm b, được liên kết với tạo tác a và được kiểm tra với tạo phẩm e được triển khai dưới dạng sơ khai. Việc triển khai và tích hợp tiến hành theo cách này cho đến khi tất cả các tạo phẩm đã được tích hợp vào sản phẩm. Một thứ tự từ trên xuống có thể khác là a, b, e, h, c, d, f, i, g, j, k, l và m. Với thứ tự này, các phần của tích hợp có thể tiến hành song song theo cách sau. Sau khi a đã được mã hóa và kiểm tra, một lập trình viên có thể sử dụng tạo phẩm a để triển khai và tích hợp b, e và h, trong khi một lập trình viên khác có thể sử dụng a để làm việc song song trên c, d, f và i. Sau khi hoàn thành d và f, lập trình viên thứ ba có thể bắt đầu làm việc trên g, j, k, l và m.

Giả sử rằng tạo phẩm a tự nó thực thi chính xác trong một trường hợp thử nghiệm cụ thể. Tuy nhiên, khi cùng một dữ liệu thử nghiệm được gửi sau khi b đã được mã hóa và tích hợp vào sản phẩm, hiện bao gồm các tạo phẩm a và b được liên kết với nhau, thì thử nghiệm không thành công. Lỗi có thể ở một trong hai vị trí, trong vật phẩm b hoặc giao diện giữa vật phẩm a và b. Nói chung, bất cứ khi nào một tạo tác mã mNew được thêm vào những gì đã được thử nghiệm cho đến nay và trường hợp thử nghiệm thành công trước đó không thành công, lỗi gần như chắc chắn nằm ở mNew hoặc (các) giao diện giữa mNew và phần còn lại của sản phẩm. Theo đó, tích hợp từ trên xuống hỗ trợ cách ly lỗi.

Một điểm mạnh khác của tích hợp từ trên xuống là các lỗi thiết kế lớn xuất hiện sớm. Các tạo phẩm của một sản phẩm có thể được chia thành hai nhóm, tạo tác logic và tạo tác hoạt động. Các tạo tác logic về cơ bản kết hợp luồng ra quyết định của các khía cạnh kiểm soát của sản phẩm. Các thành phần logic nói chung là những thành phần nằm gần gốc trong sơ đồ kết nối. Ví dụ, trong Hình 15.6, thật hợp lý khi kỳ vọng các tạo phẩm a, b, c, d và có lẽ g và j là các tạo tác logic. Mặt khác, các tạo phẩm hoạt động thực hiện các hoạt động thực tế của sản phẩm. Ví dụ: một tạo phẩm hoạt động có thể được đặt tên là getLineFromTerminal hoặc measureTemperatureOfReactorCore. Các hiện vật hoạt động thường được tìm thấy ở các cấp độ thấp hơn, gần các lá, của sơ đồ kết nối. Trong Hình 15.6, các thành phần e, f, h, i, k, l và m là các thành phần hoạt động.

Điều luôn luôn quan trọng là viết mã và kiểm tra các tạo phẩm logic trước khi viết mã và kiểm tra các tạo tác hoạt động. Điều này đảm bảo rằng bất kỳ lỗi thiết kế lớn nào cũng xuất hiện sớm. Giả sử toàn bộ sản phẩm được hoàn thành trước khi phát hiện ra một lỗi lớn. Các bộ phận lớn của sản phẩm

phải được thực hiện lại, đặc biệt là các tạo tác logic thể hiện luồng kiểm soát. Nhiều tạo phẩm hoạt động có thể được tái sử dụng trong sản phẩm được xây dựng lại; ví dụ: cần có một cấu phần phần mềm như getLineFromTerminal hoặc measureTemperatureOfReactorCore cho dù sản phẩm được tái cấu trúc như thế nào. Tuy nhiên, cách các tạo phẩm hoạt động được kết nối với các tạo phẩm khác trong sản phẩm có thể phải thay đổi, dẫn đến các công việc không cần thiết. Do đó, lỗi thiết kế được phát hiện càng sớm thì việc sửa sản phẩm và quay lại lịch trình phát triển càng nhanh và ít tốn kém hơn. Thứ tự mà các tạo phẩm được triển khai và tích hợp bằng cách sử dụng chiến lược từ trên xuống về cơ bản đảm bảo rằng các tạo phẩm logic thực sự được triển khai và tích hợp trước các tạo phẩm hoạt động, bởi vì các tạo phẩm logic hầu như luôn là tổ tiên của các tạo phẩm hoạt động trong sơ đồ kết nối. Đây là sức mạnh chính của tích hợp từ trên xuống.

Tuy nhiên, tích hợp từ trên xuống có một điểm yếu: Các tạo phẩm mã có khả năng tái sử dụng có thể không được kiểm tra đầy đủ, như sẽ được giải thích. Việc tái sử dụng một tạo tác được cho là không chính xác, đã được kiểm tra kỹ lưỡng có thể sẽ kém hiệu quả hơn so với việc viết tạo tác đó từ đầu, bởi vì giả định rằng một tạo phẩm là chính xác có thể dẫn đến kết luận sai khi sản phẩm bị lỗi. Thay vì nghi ngờ tạo tác được kiểm tra lại, được sử dụng lại không đầy đủ, người kiểm tra có thể nghĩ rằng lỗi nằm ở chỗ khác, dẫn đến lãng phí công sức.

Các tạo phẩm logic có thể là một số vấn đề cụ thể và do đó không sử dụng được trong bối cảnh khác. Tuy nhiên, các tạo phẩm hoạt động, đặc biệt nếu chúng có sự gắn kết thông tin (Phần 7.2.7), có thể được tái sử dụng trong các sản phẩm trong tương lai và do đó, cần phải kiểm tra kỹ lưỡng. Thật không may, các tạo phẩm hoạt động nói chung là các tạo phẩm mã cấp thấp hơn trong sơ đồ kết nối và do đó không được kiểm tra thường xuyên như các tạo phẩm cấp trên. Ví dụ: nếu có 184 cấu phần phần mềm, cấu phần phần mềm gốc được kiểm tra 184 lần, trong khi cấu phần phần mềm cuối cùng được tích hợp vào sản phẩm chỉ được kiểm tra một lần. Tích hợp từ trên xuống làm cho việc tái sử dụng trở thành một công việc rủi ro do hậu quả của việc kiểm tra không đầy đủ các thành phần hoạt động.

Tình hình trở nên trầm trọng hơn nếu sản phẩm được thiết kế tốt; trên thực tế, thiết kế càng tốt thì khả năng thử nghiệm các đồ tạo tác càng kém kỹ càng. Để thấy điều này, hãy xem xét một máy tính giả tạo. Thành phần tạo tác này nhận hai đối số, một số dấu phẩy động x có căn bậc hai được xác định và một errorFlag được đặt thành true nếu x âm. Giả sử thêm rằng computeSquareRoot được gọi bởi tạo phẩm a3 và a3 chứa câu lệnh

nếu (x > = 0)

y = computeSquareRoot(x, errorFlag);

Nói cách khác, computeSquareRoot không bao giờ được gọi trừ khi giá trị của x không âm; do đó, hiện vật không bao giờ có thể được kiểm tra với các giá trị âm của x để xem liệu nó có hoạt động chính xác hay không. Loại thiết kế trong đó vật phẩm gọi bao gồm kiểm tra an toàn thuộc loại này được gọi là lập trình phòng thủ. Do lập trình phòng thủ, các tạo phẩm hoạt động cấp dưới khó có thể được kiểm tra kỹ lưỡng nếu được tích hợp từ trên xuống. Một giải pháp thay thế cho lập trình phòng thủ là sử dụng thiết kế hướng đến trách nhiệm (Phần 1.9). Ở đây, các kiểm tra an toàn cần thiết được tích hợp vào tạo phẩm được gọi, thay vì người gọi. Một cách tiếp cận khác là sử dụng các xác nhận trong tạo phẩm được gọi (Phần 6.5.3).

**FIGURE 15.7**

The product

of Figure 15.6 developed using sandwich integration.

* + 1. Bottom-up Integration 15.6.2 Tích hợp từ dưới lên

In **bottom-up integration**, if artifact mAbove sends a message to artifact mBelow, then mBelow is implemented and integrated before mAbove. In Figure 15.6, one possible bottom-up ordering is l, m, h, i, j, k, e, f, g, b, c, d, and a. To have the product coded by a team, a better bottom-up ordering is as follows: h, e, and b are given to one programmer and i, f, and c to another. The third programmer starts with l, m, j, k, and g, and then imple- ments d and integrates his or her work with the work of the second programmer. Finally, when b, c, and d have been successfully integrated, a can be implemented and integrated. The operational artifacts thereby are tested thoroughly when a bottom-up strategy is used. In addition, the testing is done with the aid of drivers, rather than by fault-shielding, defensively programmed artifacts. Although bottom-up integration solves the major dif- ficulty of top-down integration and shares with top-down integration the advantage of fault isolation, it unfortunately has a difficulty of its own. Specifically, major design faults are detected late in the implementation workflow. The logic artifacts are integrated last; hence, if there is a major design fault, it will be picked up at the end of the implementation workflow with the resulting huge cost of redesigning and recoding large portions of the

product.

Therefore, both top-down and bottom-up integration have their strengths and weak- nesses. The solution for product development is to combine the two strategies in such a way as to use their strengths and minimize their weaknesses. This leads to the idea of sandwich integration.

Trong tích hợp từ dưới lên, nếu tạo phẩm mAbove gửi thông báo đến tạo tác mBelow, thì mBelow được triển khai và tích hợp trước mAbove. Trong Hình 15.6, một thứ tự từ dưới lên có thể có là l, m, h, i, j, k, e, f, g, b, c, d, và a. Để sản phẩm được mã hóa bởi một nhóm, thứ tự từ dưới lên tốt hơn như sau: h, e và b được trao cho một lập trình viên và i, f và c cho người khác. Lập trình viên thứ ba bắt đầu với l, m, j, k và g, sau đó thực hiện d và tích hợp công việc của mình với công việc của lập trình viên thứ hai. Cuối cùng, khi b, c và d đã được tích hợp thành công, a có thể được triển khai và tích hợp. Do đó, các tạo phẩm hoạt động được kiểm tra kỹ lưỡng khi sử dụng chiến lược từ dưới lên. Ngoài ra, thử nghiệm được thực hiện với sự trợ giúp của trình điều khiển, thay vì bằng các tạo phẩm được lập trình phòng thủ, che chắn lỗi. Mặc dù tích hợp từ dưới lên giải quyết được khó khăn chính của tích hợp từ trên xuống và chia sẻ với tích hợp từ trên xuống lợi thế của việc cô lập lỗi, nhưng không may là nó có một khó khăn riêng. Cụ thể, các lỗi thiết kế chính được phát hiện muộn trong quy trình triển khai. Các tạo phẩm logic được tích hợp sau cùng; do đó, nếu có lỗi thiết kế lớn, nó sẽ được xử lý ở cuối quy trình triển khai với chi phí thiết kế lại và mã hóa lại phần lớn của quy trình.

sản phẩm.

Do đó, cả tích hợp từ trên xuống và từ dưới lên đều có điểm mạnh và điểm yếu. Giải pháp để phát triển sản phẩm là kết hợp hai chiến lược theo cách sử dụng điểm mạnh của chúng và giảm thiểu điểm yếu của chúng. Điều này dẫn đến ý tưởng tích hợp bánh sandwich

* + 1. Sandwich Integration 15.6.3 Tích hợp bánh sandwich

Consider the interconnection diagram shown in Figure 15.7. Six of the code artifacts—a,

b, c, d, g, and j—are logic artifacts and therefore should be integrated top down. Seven are

a

b

c

d

g

e

f

h

i

k

l

m

j

Logic artifact

Operational artifact

Interface connecting logic and operational artifacts



**Just in Case You Wanted to Know**

**Box 15.5**

The term *sandwich integration* [Myers, 1979] comes from viewing the logic artifacts and the operational artifacts as the top and the bottom of a sandwich, and the interfaces that con- nect them as the sandwich filling. This can be seen (sort of) in Figure 15.7.

operational artifacts—e, f, h, i, k, l, and m—and should be integrated bottom up. Because neither top-down nor bottom-up integration is suitable for all the artifacts, the solution is to partition them. The six logic artifacts are integrated top down and any major design faults can be caught early. The seven operational artifacts are integrated bottom up. They therefore receive a thorough testing, unshielded by defensively programmed artifacts that invoke them, and therefore they can be reused with confidence in other products. When all artifacts have been appropriately integrated, the interfaces between the two groups of arti- facts are tested, one by one. There is fault isolation at all times during this process, called **sandwich integration** (see Just in Case You Wanted to Know Box 15.5).

**514** Part B *The Workflows of the Software Life Cycle*

Figure 15.8 summarizes the strengths and weaknesses of sandwich integration, as well as the other integration techniques previously discussed in this chapter.

Sandwich integration is summarized in How to Perform Box 15.1.

Xem xét sơ đồ kết nối được hiển thị trong Hình 15.7. Sáu trong số các tạo tác mã—a,

b, c, d, g và j—là các tạo tác logic và do đó nên được tích hợp từ trên xuống. Bảy là các tạo phẩm hoạt động—e, f, h, i, k, l và m—và nên được tích hợp từ dưới lên. Bởi vì cả tích hợp từ trên xuống lẫn từ dưới lên đều không phù hợp với tất cả các tạo phẩm, nên giải pháp là phân vùng chúng. Sáu tạo tác logic được tích hợp từ trên xuống và bất kỳ lỗi thiết kế chính nào đều có thể được phát hiện sớm. Bảy tạo tác hoạt động được tích hợp từ dưới lên. Do đó, chúng được kiểm tra kỹ lưỡng, không được bảo vệ bởi các thành phần tạo tác được lập trình phòng thủ gọi chúng và do đó chúng có thể được sử dụng lại một cách an toàn trong các sản phẩm khác. Khi tất cả các tạo tác đã được tích hợp một cách thích hợp, các giao diện giữa hai nhóm tạo tác được kiểm tra, từng cái một. Luôn có sự cách ly lỗi trong quá trình này, được gọi là tích hợp bánh sandwich (xem Đề phòng trường hợp bạn muốn biết Hộp 15.5).

Hình 15.8 tóm tắt các điểm mạnh và điểm yếu của tích hợp bánh sandwich, cũng như các kỹ thuật tích hợp khác đã thảo luận trước đó trong chương này.

Tích hợp bánh sandwich được tóm tắt trong Cách thực hiện Hộp 15.1.

* + 1. Integration of Object-Oriented Products 15.6.4 Tích hợp các sản phẩm hướng đối tượng

Objects can be integrated either bottom up or top down. If top-down integration is chosen, stubs are used for each method in the same way as with classical modules.

If bottom-up integration is used, the objects that do not send messages to other objects are implemented and integrated first. Then, the objects that send messages to those objects

**FIGURE 15.8**

|  |
| --- |
| **Approach Strengths Weaknesses**  Implementation then — No fault isolation  integration Major design faults show up  (Section 15.6) late  Potentially reusable code artifacts are not adequately tested  Top-down integration Fault isolation Potentially reusable code (Section 15.6.1) Major design faults show up artifacts are not  early adequately tested  Bottom-up integration Fault isolation Major design faults show up  (Section 15.6.2) Potentially reusable code late  artifacts are adequately tested  Sandwich integration Fault isolation — (Section 15.6.3) Major design faults show up  early  Potentially reusable code artifacts are adequately tested |

A summary of the integration approaches presented in this chapter and the section in which each is described.



* **In parallel,**

Implement and integrate the logic artifacts top down. Implement and integrate the operational artifacts bottom up.

* Test the interfaces between the logic artifacts and the operational artifacts.

**Box 15.1**

**How to Perform Sandwich Integration**

are implemented and integrated, and so on, until all the objects in the product have been implemented and integrated. (This process must be modified if there is recursion.)

Because both top-down and bottom-up integration are supported, sandwich integration also can be used. If the product is implemented in a hybrid object-oriented language like C++, the classes generally are operational artifacts and therefore integrated bottom up.

Many of the artifacts that are not classes are logic artifacts. These are implemented and integrated in a top-down manner. The other artifacts are operational, so they are imple- mented and integrated bottom up. Finally, all the nonobject artifacts are integrated with the objects.

Even when the product is implemented using a pure object-oriented language like Java, class methods (sometimes referred to as **static methods**) such as main and utility meth- ods usually are similar in structure to logic modules of the classical paradigm. Therefore, class methods are also implemented top down and then integrated with the other objects. In other words, when implementing and integrating an object-oriented product, variants of sandwich integration are used.

Các đối tượng có thể được tích hợp từ dưới lên hoặc từ trên xuống. Nếu tích hợp từ trên xuống được chọn, sơ khai được sử dụng cho từng phương pháp giống như với các mô-đun cổ điển.

Nếu tích hợp từ dưới lên được sử dụng, các đối tượng không gửi tin nhắn đến các đối tượng khác sẽ được triển khai và tích hợp trước. Sau đó, các đối tượng gửi thông điệp đến các đối tượng đó được triển khai và tích hợp, v.v. cho đến khi tất cả các đối tượng trong sản phẩm đã được triển khai và tích hợp. (Quá trình này phải được sửa đổi nếu có đệ quy.)

Bởi vì cả tích hợp từ trên xuống và từ dưới lên đều được hỗ trợ, nên tích hợp bánh sandwich cũng có thể được sử dụng. Nếu sản phẩm được triển khai bằng ngôn ngữ hướng đối tượng kết hợp như C++, thì các lớp nói chung là các tạo phẩm hoạt động và do đó được tích hợp từ dưới lên.

Nhiều tạo phẩm không phải là lớp là tạo tác logic. Chúng được thực hiện và tích hợp theo cách từ trên xuống. Các tạo phẩm khác đang hoạt động, vì vậy chúng được triển khai và tích hợp từ dưới lên. Cuối cùng, tất cả các tạo tác phi đối tượng được tích hợp với các đối tượng.

Ngay cả khi sản phẩm được triển khai bằng cách sử dụng một ngôn ngữ hướng đối tượng thuần túy như Java, các phương thức lớp (đôi khi được gọi là các phương thức tĩnh) chẳng hạn như các phương thức chính và tiện ích thường có cấu trúc tương tự như các mô-đun logic của mô hình cổ điển. Do đó, các phương thức của lớp cũng được triển khai từ trên xuống và sau đó được tích hợp với các đối tượng khác. Nói cách khác, khi triển khai và tích hợp một sản phẩm hướng đối tượng, các biến thể của tích hợp bánh sandwich được sử dụng.

* + 1. Management of Integration 15.6.5 Quản lý tích hợp

A problem for management is discovering, at integration time, that the code artifacts simply do not fit together. For example, suppose that programmer 1 coded object o1, and programmer 2 coded object o2. In the version of the design documentation used by programmer 1, object o1 sends a message to object o2 passing four arguments, but the version of the design documentation used by programmer 2 states clearly that only three arguments are passed to o2. A problem like this can arise when a change is made to only one copy of the design document, without informing all the members of the develop- ment group. Both programmers know that they are in the right; neither is prepared to compromise, because the programmer who gives in must recode large portions of the product.

To solve these and similar problems of incompatibility, the entire integration process should be run by the SQA group. Furthermore, as with testing during other workflows, the SQA group has the most to lose if the integration testing is performed improperly. The SQA group therefore is the most likely to ensure that the testing is performed thoroughly. Hence, the manager of the SQA group should have responsibility for all aspects of inte- gration testing. He or she must decide which artifacts are implemented and integrated top down and which bottom up and assign integration-testing tasks to the appropriate

individuals. The SQA group, which will have drawn up the integration test plan in the software project management plan, is responsible for implementing that plan.

At the end of the integration process, all the code artifacts will have been tested and combined into a single product.

Một vấn đề đối với ban quản lý là phát hiện ra rằng tại thời điểm tích hợp, các tạo phẩm mã đơn giản là không khớp với nhau. Ví dụ: giả sử rằng lập trình viên 1 mã hóa đối tượng o1 và lập trình viên 2 mã hóa đối tượng o2. Trong phiên bản của tài liệu thiết kế được lập trình viên 1 sử dụng, đối tượng o1 gửi một thông báo tới đối tượng o2 để truyền bốn đối số, nhưng phiên bản của tài liệu thiết kế do lập trình viên 2 sử dụng nêu rõ rằng chỉ có ba đối số được truyền cho o2. Một vấn đề như thế này có thể phát sinh khi một thay đổi được thực hiện đối với chỉ một bản sao của tài liệu thiết kế mà không thông báo cho tất cả các thành viên của nhóm phát triển. Cả hai lập trình viên đều biết rằng họ đúng; cả hai đều không sẵn sàng thỏa hiệp, bởi vì lập trình viên nhượng bộ phải mã hóa lại phần lớn sản phẩm.

Để giải quyết những vấn đề này và các vấn đề tương tự về tính không tương thích, toàn bộ quá trình tích hợp phải được điều hành bởi nhóm SQA. Hơn nữa, cũng giống như thử nghiệm trong các quy trình công việc khác, nhóm SQA sẽ mất nhiều nhất nếu thử nghiệm tích hợp được thực hiện không đúng cách. Do đó, nhóm SQA có nhiều khả năng nhất để đảm bảo rằng thử nghiệm được thực hiện kỹ lưỡng. Do đó, người quản lý của nhóm SQA nên có trách nhiệm đối với tất cả các khía cạnh của thử nghiệm tích hợp. Anh ấy hoặc cô ấy phải quyết định các tạo phẩm nào được triển khai và tích hợp từ trên xuống và từ dưới lên và phân công các nhiệm vụ kiểm tra tích hợp cho các cá nhân thích hợp. Nhóm SQA, nhóm sẽ vạch ra kế hoạch kiểm tra tích hợp trong kế hoạch quản lý dự án phần mềm, chịu trách nhiệm thực hiện kế hoạch đó.

Khi kết thúc quá trình tích hợp, tất cả các tạo phẩm mã sẽ được kiểm tra và kết hợp thành một sản phẩm duy nhất.

* 1. The Implementation Workflow 15.7 Quy trình thực hiện

The overall aim of the **implementation workflow** is to implement the target software product in the selected implementation language. More precisely, as explained in Section 14.9, a large software product is partitioned into smaller subsystems, which are then imple- mented in parallel by coding teams. The subsystems, in turn, consist of **components** or **code artifacts**.

As soon as a code artifact has been coded, the programmer tests it; this is termed **unit testing**. Once the programmer is satisfied that the code artifact is correct, it is passed on to the quality assurance group for further testing. This testing by the quality assurance group is part of the test workflow, described in Sections 15.20 through 15.22.

Mục đích tổng thể của quy trình triển khai là triển khai sản phẩm phần mềm đích bằng ngôn ngữ triển khai đã chọn. Chính xác hơn, như đã giải thích trong Phần 14.9, một sản phẩm phần mềm lớn được phân chia thành các hệ thống con nhỏ hơn, sau đó được triển khai song song bởi các nhóm viết mã. Đến lượt mình, các hệ thống con bao gồm các thành phần hoặc mã tạo tác.

Ngay sau khi mã tạo tác đã được mã hóa, lập trình viên sẽ kiểm tra nó; điều này được gọi là thử nghiệm đơn vị. Sau khi lập trình viên hài lòng rằng tạo tác mã là chính xác, nó sẽ được chuyển cho nhóm đảm bảo chất lượng để kiểm tra thêm. Thử nghiệm này của nhóm đảm bảo chất lượng là một phần của quy trình thử nghiệm, được mô tả trong Phần 15.20 đến 15.22.

*Case Study*



* 1. The Implementation Workflow: The MSG Foundation Case Study

Complete implementations of the MSG Foundation product in both C++ and Java can be downloaded from [www.mhhe.com/schach](http://www.mhhe.com/schach). The programmers included a variety of comments to aid the postdelivery maintenance programmers.

Testing during the implementation workflow is examined next.

15.8 Quy trình thực hiện: Nghiên cứu tình huống của Quỹ MSG

Có thể tải xuống toàn bộ các triển khai của sản phẩm MSG Foundation trong cả C++ và Java từ www.mhhe.com/schach. Các lập trình viên đã đưa vào nhiều nhận xét khác nhau để hỗ trợ các lập trình viên bảo trì sau khi giao hàng.

Kiểm tra trong quy trình thực hiện sẽ được kiểm tra tiếp theo.

* 1. **The Test Workflow: Implementation 15.9 Quy trình kiểm tra: Thực hiện**

A number of different types of testing have to be performed during the implementation workflow, including unit testing, integration testing, product testing, and acceptance test- ing. These types of testing are discussed in the following sections.

As pointed out in Section 6.6, code artifacts (modules, classes) undergo two types of testing: informal unit testing performed by the programmer while developing the code artifact and methodical unit testing carried out by the SQA group after the programmer is satisfied that the artifact appears to function correctly. This methodical testing is de- scribed in Sections 15.10 through 15.14. In turn, there are two basic types of methodical testing, **non-execution-based testing**, in which the artifact is reviewed by a team, and **execution-based testing** in which the artifact is run against test cases. Techniques for selecting test cases now are described.

Một số loại thử nghiệm khác nhau phải được thực hiện trong quy trình triển khai, bao gồm thử nghiệm đơn vị, thử nghiệm tích hợp, thử nghiệm sản phẩm và thử nghiệm chấp nhận. Các loại thử nghiệm này được thảo luận trong các phần sau.

Như đã chỉ ra trong Phần 6.6, tạo phẩm mã (mô-đun, lớp) trải qua hai loại thử nghiệm: thử nghiệm đơn vị không chính thức do lập trình viên thực hiện trong khi phát triển tạo phẩm mã và thử nghiệm đơn vị có phương pháp do nhóm SQA thực hiện sau khi lập trình viên hài lòng rằng tạo phẩm đó dường như hoạt động chính xác. Thử nghiệm có phương pháp này được mô tả trong Mục 15.10 đến 15.14. Đổi lại, có hai loại thử nghiệm phương pháp cơ bản, thử nghiệm không dựa trên thực thi, trong đó tạo tác được xem xét bởi một nhóm và thử nghiệm dựa trên thực thi trong đó tạo tác được chạy đối với các trường hợp thử nghiệm. Các kỹ thuật để chọn các trường hợp thử nghiệm hiện được mô tả.



**Just in Case You Wanted to Know**

Chapter 15 *Implementation* **517**

**Box 15.6**

It is reasonable to ask why so many different names are given for the same testing concept. As so often happens in software engineering, the same concept was discovered, indepen- dently, by a number of different researchers, each of whom invented his or her own term. By the time the software engineering community realized that these were different names for the identical concept it was too late—the diverse names had crept into the software engineering vocabulary.

In this book, I use the terms *black-box testing* and *glass-box testing*. These terms are par- ticularly descriptive. When we test to specifications, we treat the code as a totally opaque black box. Conversely, when we test to code, we need to be able to see inside the box: hence the term *glass-box testing*. I avoid the term *white-box testing* because it is somewhat confusing. After all, a box painted white is just as opaque as one painted black.

Thật hợp lý khi đặt câu hỏi tại sao có quá nhiều tên gọi khác nhau cho cùng một khái niệm thử nghiệm. Như thường xảy ra trong công nghệ phần mềm, cùng một khái niệm đã được phát hiện một cách độc lập bởi một số nhà nghiên cứu khác nhau, mỗi người trong số họ đã phát minh ra thuật ngữ của riêng mình. Vào thời điểm cộng đồng công nghệ phần mềm nhận ra rằng đây là những tên gọi khác nhau cho cùng một khái niệm thì đã quá muộn—những tên gọi đa dạng đã len lỏi vào từ vựng công nghệ phần mềm.

Trong cuốn sách này, tôi sử dụng thuật ngữ kiểm thử hộp đen và kiểm thử hộp thủy tinh. Những thuật ngữ này đặc biệt mang tính mô tả. Khi chúng tôi kiểm tra các thông số kỹ thuật, chúng tôi coi mã như một hộp đen hoàn toàn mờ đục. Ngược lại, khi chúng tôi kiểm tra mã, chúng tôi cần có thể nhìn thấy bên trong hộp: do đó có thuật ngữ kiểm tra hộp thủy tinh. Tôi tránh thuật ngữ kiểm thử hộp trắng vì nó hơi khó hiểu. Rốt cuộc, một chiếc hộp được sơn màu trắng cũng mờ đục như một chiếc hộp được sơn màu đen.

* 1. Test Case Selection 15.10 Lựa chọn trường hợp kiểm thử

The worst way to test a code artifact is to use haphazard test data. The tester sits in front of the keyboard, and whenever the artifact requests input, the tester responds with arbitrary data. As will be shown, there is never time to test more than the tiniest fraction of all pos- sible test cases, which easily can number many more than 10100. The few test cases that can be run, perhaps, on the order of 1000, are too valuable to waste on haphazard data. Worse, there is a tendency when the machine solicits input to respond more than once with the same data, wasting even more test cases. It is clear that test cases must be constructed systematically.

Cách tồi tệ nhất để kiểm tra mã tạo tác là sử dụng dữ liệu kiểm tra ngẫu nhiên. Người kiểm tra ngồi trước bàn phím và bất cứ khi nào tạo tác yêu cầu đầu vào, người kiểm tra sẽ phản hồi với dữ liệu tùy ý. Như sẽ được chỉ ra, không bao giờ có thời gian để kiểm thử nhiều hơn phần nhỏ nhất của tất cả các trường hợp kiểm thử có thể, có thể dễ dàng đánh số nhiều hơn 10100. Một số ít trường hợp kiểm thử có thể chạy, có lẽ, theo thứ tự 1000, là quá có giá trị để lãng phí trên dữ liệu lộn xộn. Tồi tệ hơn, có xu hướng khi máy yêu cầu đầu vào phản hồi nhiều lần với cùng một dữ liệu, thậm chí còn lãng phí nhiều trường hợp thử nghiệm hơn. Rõ ràng là các trường hợp kiểm thử phải được xây dựng một cách có hệ thống.

* + 1. Testing to Specifications versus Testing to Code 15.10.1 Kiểm tra thông số kỹ thuật so với Kiểm tra mã

Test data for unit testing can be constructed systematically in two basic ways. The first is to **test to specifications**. This technique also is called **black-box**, **behavioral**, **data- driven**, **functional**, and **input/output-driven testing**. In this approach, the code itself is ignored; the only information used in drawing up test cases is the specification document. The other extreme is to **test to code** and to ignore the specification document when se- lecting test cases. Other names for this technique are **glass-box**, **white-box**, **structural**, **logic-driven**, and **path-oriented testing** (for an explanation of why there are so many different terms, see Just in Case You Wanted to Know Box 15.6).

We now consider the feasibility of each of these two techniques, starting with testing to specifications.

Dữ liệu thử nghiệm cho thử nghiệm đơn vị có thể được xây dựng một cách có hệ thống theo hai cách cơ bản. Đầu tiên là kiểm tra các thông số kỹ thuật. Kỹ thuật này còn được gọi là kiểm thử hộp đen, hành vi, dữ liệu, chức năng và đầu vào/đầu ra. Theo cách tiếp cận này, bản thân mã bị bỏ qua; thông tin duy nhất được sử dụng trong việc xây dựng các ca kiểm thử là tài liệu đặc tả. Một thái cực khác là kiểm thử mã và bỏ qua tài liệu đặc tả khi lựa chọn các trường hợp kiểm thử. Các tên gọi khác của kỹ thuật này là thử nghiệm hộp thủy tinh, hộp trắng, cấu trúc, dựa trên logic và hướng theo đường dẫn (để biết giải thích tại sao có quá nhiều thuật ngữ khác nhau, hãy xem Đề phòng trường hợp bạn muốn biết Hộp 15.6).

Bây giờ chúng ta xem xét tính khả thi của từng kỹ thuật trong số hai kỹ thuật này, bắt đầu bằng việc thử nghiệm các thông số kỹ thuật.

* + 1. Feasibility of Testing to Specifications 15.10.2 Tính khả thi của Thử nghiệm đối với Thông số kỹ thuật

Consider the following example. Suppose that the specifications for a certain data- processing product state that five types of commission and seven types of discount must be incorporated. Testing every possible combination of just commission and discount requires 35 test cases. It is no use saying that commission and discount are computed in two entirely separate code artifacts and hence may be tested independently—in black- box testing, the product is treated as a black box, and its internal structure therefore is completely irrelevant.

This example contains only two factors, commission and discount, taking on five and seven different values, respectively. Any realistic product has hundreds, if not thousands,

of different factors. Even if there are only 20 factors, each taking on only four different values, a total of 420 or 1.1 × 1012 different test cases must be examined.

To see the implications of over a trillion test cases, consider how long it would take to test

them all. If a team of programmers could be found that could generate, run, and examine test cases at an average rate of one every 30 seconds, then it would take more than a million years to test the product exhaustively.

Therefore, exhaustive testing to specifications is impossible in practice because of the combinatorial explosion. There simply are too many test cases to consider. Testing to code now is examined.

Hãy xem xét ví dụ sau. Giả sử rằng các thông số kỹ thuật cho một sản phẩm xử lý dữ liệu nhất định nêu rõ rằng phải kết hợp năm loại hoa hồng và bảy loại chiết khấu. Thử nghiệm mọi sự kết hợp có thể có giữa hoa hồng và chiết khấu yêu cầu 35 trường hợp thử nghiệm. Không có ích gì khi nói rằng hoa hồng và chiết khấu được tính bằng hai tạo phẩm mã hoàn toàn riêng biệt và do đó có thể được thử nghiệm độc lập—trong thử nghiệm hộp đen, sản phẩm được coi là hộp đen và do đó cấu trúc bên trong của nó hoàn toàn không liên quan.

Ví dụ này chỉ chứa hai yếu tố, hoa hồng và chiết khấu, lần lượt nhận 5 và 7 giá trị khác nhau. Bất kỳ sản phẩm thực tế nào cũng có hàng trăm, nếu không muốn nói là hàng nghìn yếu tố khác nhau. Ngay cả khi chỉ có 20 yếu tố, mỗi yếu tố chỉ nhận bốn giá trị khác nhau, thì phải kiểm tra tổng cộng 420 hoặc 1,1 × 1012 trường hợp thử nghiệm khác nhau.

Để xem ý nghĩa của hơn một nghìn tỷ trường hợp thử nghiệm, hãy xem xét sẽ mất bao lâu để thử nghiệm

khu mua sắm. Nếu có thể tìm thấy một nhóm lập trình viên có thể tạo, chạy và kiểm tra các trường hợp thử nghiệm với tốc độ trung bình cứ sau 30 giây, thì sẽ mất hơn một triệu năm để kiểm tra sản phẩm một cách triệt để.

Do đó, việc kiểm tra toàn diện các thông số kỹ thuật là không thể trong thực tế do vụ nổ tổ hợp. Đơn giản là có quá nhiều trường hợp thử nghiệm để xem xét. Kiểm tra mã bây giờ được kiểm tra.

* + 1. Feasibility of Testing to Code 15.10.3 Tính khả thi của việc kiểm tra mã

The most common form of testing to code requires that each path through the code artifact be executed at least once.

* To see the infeasibility of this, consider the code fragment of Figure 15.9. The cor- responding flowchart is shown in Figure 15.10. Even though the flowchart appears to be almost trivial, it has over 1012 different paths. There are five possible paths through the central group of six shaded boxes, and the total number of possible paths through the flowchart therefore is

1 2 3

. . .

18 5 × (518—1) 12

5 + 5 + 5 +

+ 5 = (5—1) = 4.77 × 10

If there can be this many paths through a simple flowchart containing a single loop, it is not difficult to imagine the total number of different paths in a code artifact of reason- able size and complexity, let alone a large artifact with many loops. In short, the huge number of possible paths renders exhaustive testing to code as infeasible as exhaustive testing to specifications.

Hình thức kiểm tra mã phổ biến nhất yêu cầu mỗi đường dẫn qua tạo phẩm mã phải được thực thi ít nhất một lần.

• Để thấy tính không khả thi của điều này, hãy xem xét đoạn mã của Hình 15.9. Lưu đồ tương ứng được thể hiện trong Hình 15.10. Mặc dù sơ đồ có vẻ gần như tầm thường, nhưng nó có hơn 1012 đường dẫn khác nhau. Có năm đường đi có thể đi qua nhóm trung tâm gồm sáu hộp được tô bóng và do đó, tổng số đường đi có thể đi qua lưu đồ là

1 2 3

. . .

18 5 × (518—1) 12

5 + 5 + 5 +

+ 5 = (5—1) = 4,77 × 10

Nếu có thể có nhiều đường dẫn như vậy thông qua một lưu đồ đơn giản chứa một vòng lặp, thì không khó để tưởng tượng tổng số đường dẫn khác nhau trong một tạo phẩm mã có kích thước và độ phức tạp hợp lý, chứ đừng nói đến một tạo phẩm lớn với nhiều vòng lặp. Nói tóm lại, số lượng lớn các đường dẫn có thể làm cho việc kiểm tra toàn diện đối với mã cũng không khả thi bằng việc kiểm tra toàn diện các thông số kỹ thuật.

**FIGURE 15.9**

A code fragment.

*read* (kmax) // kmax *is an integer between* 1 *and* 18

**for** (k = 0; k < kmax; k++) **do**

{

*read* (myChar) // myChar *is the character* A, B, *or* C

**switch** (myChar)

{

**case** ’A’: blockA;

**if** (cond1) blockC;

**break**; **case** ’B’: blockB;

**if** (cond2) blockC;

**break**; **case** ’C’: blockC; **break**;

}

blockD;

}

**FIGURE 15.10**



'A'

myChar

'B'

'C'

cond1

true

true

cond2

false

false

loop ≤ 18 times

blockD

blockC

blockA

blockB

A flowchart with over 1012 possible paths.

* Furthermore, testing to code requires the tester to exercise every path. It is possible to exercise every path without detecting every fault in the product; that is, testing to code is not reliable. To see this, consider the code fragment shown in Figure 15.11 [Myers, 1976]. The fragment was written to test the equality of three integers, x, y, and z, using the totally fallacious assumption that if the average of three numbers is equal to the first number, then the three numbers are equal. Two test cases are shown in Figure 15.11. In the first test case the value of the average of the three numbers is 6/3 or 2, which is not equal to 1. The product therefore correctly informs the tester that x, y, and z are unequal. The integers x, y, and z all equal 2 in the second test case, so the product computes their average as 2, which is equal to the value of x, and the product correctly concludes that the three numbers are equal. Accordingly, both paths through the product have been exercised without the fault being detected. Of course, the fault would come to light if test data such as x = 2, y = 1, z = 3 are used.
* A third difficulty with path testing is that a path can be tested only if it is present. Consider the code fragment shown in Figure 15.12(a). Clearly, two paths are to be

**FIGURE 15.11**

An incorrect code fragment for determining if three integers are equal, together with two test cases.

**if** ((x + y + z)/3 == x)

*print* “x, y, z are equal in value”;

**else**

*print* “x, y, z are unequal”;

Test case 1: x = 1, y = 2, z = 3 Test case 2: x = y = z = 2

**FIGURE 15.12**

Two code fragments for computing a quotient.

**if** (d == 0) zeroDivisionRoutine ();

**else**

x = n/d;

(a)

x = n/d;

(b)

tested, corresponding to the cases d = 0 and d  0. Next, consider the single statement of Figure 15.12(b). Now there is only one path, and this path can be tested without the fault being detected. In fact, a programmer who omits checking whether d = 0 in his or her code is likely to be unaware of the potential danger, and the case d = 0 will not be included in the programmer’s test data. This problem is an additional argument for having an independent software quality assurance group whose job includes detecting faults of this type.

These examples show conclusively that the criterion “exercise all paths in the product” is not **reliable**, as products exist for which some data exercising a given path detect a fault and different data exercising the same path do not. However, path-oriented testing is **valid**, because it does not inherently preclude selecting test data that might reveal the fault.

Because of the combinatorial explosion, neither exhaustive testing to specifications nor exhaustive testing to code is feasible. A compromise is needed, using techniques that high- light as many faults as possible, while accepting that there is no way to guarantee that all faults have been detected. A reasonable way to proceed is to use black-box test cases first (testing to specifications) and then develop additional test cases using glass-box techniques (testing to code).

• Hơn nữa, kiểm tra mã yêu cầu người kiểm tra thực hiện mọi đường dẫn. Có thể thực hiện mọi đường dẫn mà không phát hiện mọi lỗi trong sản phẩm; nghĩa là, kiểm tra mã không đáng tin cậy. Để thấy điều này, hãy xem xét đoạn mã được hiển thị trong Hình 15.11 [Myers, 1976]. Đoạn này được viết để kiểm tra sự bằng nhau của ba số nguyên x, y và z, sử dụng giả định hoàn toàn sai lầm rằng nếu trung bình cộng của ba số bằng số đầu tiên thì ba số đó bằng nhau. Hai trường hợp thử nghiệm được thể hiện trong Hình 15.11. Trong trường hợp thử nghiệm đầu tiên, giá trị trung bình cộng của ba số là 6/3 hoặc 2, không bằng 1. Do đó, sản phẩm thông báo chính xác cho người thử nghiệm rằng x, y và z không bằng nhau. Các số nguyên x, y và z đều bằng 2 trong trường hợp thử nghiệm thứ hai, vì vậy tích sẽ tính giá trị trung bình của chúng là 2, bằng với giá trị của x và tích kết luận đúng rằng ba số bằng nhau. Theo đó, cả hai đường dẫn thông qua sản phẩm đã được thực hiện mà không phát hiện ra lỗi. Tất nhiên, lỗi sẽ được đưa ra ánh sáng nếu dữ liệu thử nghiệm như x = 2, y = 1, z = 3 được sử dụng.

• Khó khăn thứ ba với kiểm tra đường dẫn là một đường dẫn chỉ có thể được kiểm tra nếu nó hiện diện. Xem xét đoạn mã được hiển thị trong Hình 15.12(a). Rõ ràng, có hai đường đi cần kiểm tra, tương ứng với các trường hợp d = 0 và d  0. Tiếp theo, hãy xem xét câu lệnh đơn của Hình 15.12(b). Bây giờ chỉ có một đường dẫn và đường dẫn này có thể được kiểm tra mà không phát hiện ra lỗi. Trên thực tế, một lập trình viên bỏ qua việc kiểm tra xem d = 0 trong mã của mình có khả năng không nhận thức được mối nguy hiểm tiềm tàng và trường hợp d = 0 sẽ không được đưa vào dữ liệu kiểm tra của lập trình viên. Vấn đề này là một lý lẽ bổ sung cho việc có một nhóm đảm bảo chất lượng phần mềm độc lập có công việc bao gồm phát hiện các lỗi thuộc loại này.

Các ví dụ này cho thấy một cách thuyết phục rằng tiêu chí "thực hiện tất cả các đường dẫn trong sản phẩm" là không đáng tin cậy, vì có những sản phẩm mà một số dữ liệu thực hiện một đường dẫn nhất định phát hiện ra lỗi và các dữ liệu khác thực hiện cùng một đường dẫn thì không. Tuy nhiên, kiểm tra hướng dẫn là hợp lệ, bởi vì nó vốn không loại trừ việc chọn dữ liệu kiểm tra có thể phát hiện ra lỗi.

Do bùng nổ tổ hợp, không phải thử nghiệm toàn diện đối với các thông số kỹ thuật cũng như thử nghiệm toàn diện đối với mã là khả thi. Cần có sự thỏa hiệp, sử dụng các kỹ thuật làm nổi bật càng nhiều lỗi càng tốt, đồng thời chấp nhận rằng không có cách nào để đảm bảo rằng tất cả các lỗi đã được phát hiện. Một cách hợp lý để tiến hành là sử dụng các trường hợp thử nghiệm hộp đen trước (thử nghiệm theo thông số kỹ thuật) và sau đó phát triển các trường hợp thử nghiệm bổ sung bằng kỹ thuật hộp thủy tinh (thử nghiệm theo mã).

* 1. Black-Box Unit-Testing Techniques 15.11 Kỹ thuật kiểm thử đơn vị hộp đen

Exhaustive black-box testing generally requires billions and billions of test cases. The art of testing is to devise a small, manageable set of test cases to maximize the chances of detecting a fault while minimizing the chances of wasting a test case by having the same

fault detected by more than one test case. Every test case must be chosen to detect a previ- ously undetected fault. One such black-box technique is equivalence testing combined with boundary value analysis.

Kiểm thử hộp đen toàn diện thường yêu cầu hàng tỉ tỉ trường hợp kiểm thử. Nghệ thuật kiểm thử là tạo ra một tập hợp các trường hợp kiểm thử nhỏ, có thể quản lý để tối đa hóa cơ hội phát hiện lỗi đồng thời giảm thiểu khả năng lãng phí một trường hợp kiểm thử bằng cách phát hiện cùng một lỗi bởi nhiều trường hợp kiểm thử. Mỗi trường hợp thử nghiệm phải được chọn để phát hiện lỗi chưa được phát hiện trước đó. Một kỹ thuật hộp đen như vậy là thử nghiệm tương đương kết hợp với phân tích giá trị biên.

* + 1. **Equivalence Testing and Boundary Value Analysis 15.11.1 Kiểm tra tương đương và phân tích giá trị biên** Suppose the specifications for a database product state that the product must be able to handle any number of records from 1 through 16,383 (214 – 1). If the product can handle 34 records and 14,870 records, then the chances are good that it will work fine for, say, 8252 records. In fact, the chances of detecting a fault, if present, are likely to be equally good if any test case from 1 through 16,383 records is selected. Conversely, if the product works correctly for any one test case in the range from 1 through 16,383, then it prob-

ably will work for any other test case in the range. The range from 1 through 16,383 constitutes an **equivalence class**, that is, a set of test cases such that any one member of the class is as good a test case as any other. To be more precise, the specified range of numbers of records that the product must be able to handle defines three equivalence classes:

Equivalence class 1. Less than 1 record. Equivalence class 2. From 1 through 16,383 records. Equivalence class 3. More than 16,383 records.

Testing the database product using the technique of equivalence classes then requires that one test case from each equivalence class be selected. The test case from equivalence class 2 should be handled correctly, whereas error messages should be printed for the test cases from class 1 and class 3.

A successful test case detects a previously undetected fault. To maximize the chances of finding such a fault, a high-payoff technique is **boundary value analysis**.

Experience has shown that, when a test case on or just to one side of the boundary of an equivalence class is selected, the probability of detecting a fault increases. Therefore, when testing the database product, seven test cases should be selected:

Test case 1. 0 records: Member of equivalence class 1 and adjacent to boundary value.

Test case 2. 1 record: Boundary value.

Test case 3. 2 records: Adjacent to boundary value.

Test case 4. 723 records: Member of equivalence class 2. Test case 5. 16,382 records: Adjacent to boundary value. Test case 6. 16,383 records: Boundary value.

Test case 7. 16,384 records: Member of equivalence class 3 and adjacent to boundary value.

This example applies to the input specifications. An equally powerful technique is to examine the output specifications. For example, in 2008, the minimum Social Security deduction or, more precisely, the minimum Old-Age, Survivors, and Disability Insurance (OASDI) deduction from any one paycheck permitted by the U.S. tax code was $0 and the maximum was $6324, the latter corresponding to gross earnings of $102,000. Therefore,



* **For** both the input and output specifications

**For** each range (*L*, *U*)

Select five test cases: less than *L*, equal to *L*, greater than *L* but less than *U*, equal to *U*, and greater than *U*.

**For** each set *S*

Select two test cases: a member of *S* and a nonmember of *S*.

**For** each precise value *P*

Select two cases: *P* and anything else.

**Box 15.2**

**How to Perform Equivalence Testing**

when testing a payroll product, the test cases for the Social Security deduction from pay- checks should include input data that are expected to result in deductions of exactly $0 and

$6324. In addition, test data should be set up that might result in deductions of less than $0 or more than $6324.

In general, for each range (*R*1, *R*2) listed in either the input or the output specifica- tions, five test cases should be selected, corresponding to values less than *R*1, equal to *R*1, greater than *R*1 but less than *R*2, equal to *R*2, and greater than *R*2. Where it is specified that

an item has to be a member of a certain set (for example, the input must be a letter), two

equivalence classes must be tested, a member of the specified set and a nonmember of the set. Where the specifications lay down a precise value (for example, the response must be followed by a # sign), then again there are two equivalence classes, the specified value and anything else.

The use of equivalence classes, together with boundary value analysis, to test both the input specifications and the output specifications is a valuable technique for gen- erating a relatively small set of test data with the potential of uncovering a number of faults that might well remain hidden if less powerful techniques for test data selection were used.

The process of equivalence testing is summarized in How to Perform Box 15.2.

Giả sử các thông số kỹ thuật cho trạng thái sản phẩm cơ sở dữ liệu rằng sản phẩm phải có khả năng xử lý bất kỳ số lượng bản ghi nào từ 1 đến 16.383 (214 – 1). Nếu sản phẩm có thể xử lý 34 bản ghi và 14.870 bản ghi, thì rất có thể nó sẽ hoạt động tốt, chẳng hạn như 8252 bản ghi. Trên thực tế, cơ hội phát hiện lỗi, nếu có, có khả năng cao như nhau nếu bất kỳ trường hợp thử nghiệm nào từ 1 đến 16.383 bản ghi được chọn. Ngược lại, nếu sản phẩm hoạt động chính xác cho bất kỳ trường hợp thử nghiệm nào trong phạm vi từ 1 đến 16.383, thì có thể

ably sẽ hoạt động cho bất kỳ trường hợp thử nghiệm nào khác trong phạm vi. Phạm vi từ 1 đến 16.383 tạo thành một lớp tương đương, nghĩa là, một tập hợp các trường hợp thử nghiệm sao cho bất kỳ thành viên nào của lớp cũng là một trường hợp thử nghiệm tốt như bất kỳ trường hợp thử nghiệm nào khác. Nói chính xác hơn, phạm vi số lượng bản ghi được chỉ định mà sản phẩm phải có khả năng xử lý xác định ba loại tương đương:

Lớp tương đương 1. Ít hơn 1 bản ghi. Tương đương hạng 2. Từ 1 đến 16.383 hồ sơ. Tương đương hạng 3. Hơn 16.383 hồ sơ.

Việc kiểm tra sản phẩm cơ sở dữ liệu bằng cách sử dụng kỹ thuật của các lớp tương đương sau đó yêu cầu một trường hợp thử nghiệm từ mỗi lớp tương đương được chọn. Trường hợp kiểm tra từ lớp tương đương 2 phải được xử lý chính xác, trong khi các thông báo lỗi phải được in cho các trường hợp kiểm tra từ lớp 1 và lớp 3.

Một trường hợp thử nghiệm thành công phát hiện một lỗi chưa được phát hiện trước đó. Để tối đa hóa cơ hội tìm ra một lỗi như vậy, một kỹ thuật mang lại hiệu quả cao là phân tích giá trị biên.

Kinh nghiệm cho thấy rằng, khi một trường hợp thử nghiệm trên hoặc chỉ một phía của ranh giới của một lớp tương đương được chọn, xác suất phát hiện lỗi sẽ tăng lên. Do đó, khi kiểm thử sản phẩm cơ sở dữ liệu, nên chọn bảy trường hợp kiểm thử:

Trường hợp thử nghiệm 1. 0 bản ghi: Thành viên của lớp tương đương 1 và liền kề với giá trị biên.

Ca kiểm thử 2. 1 bản ghi: Giá trị biên.

Test case 3. 2 bản ghi: Liền kề với giá trị biên.

Test case 4. 723 bản ghi: Thuộc lớp tương đương 2. Test case 5. 16.382 bản ghi: Liền kề giá trị biên. Test case 6. 16.383 bản ghi: Giá trị biên.

Test case 7. 16.384 bản ghi: Thuộc loại tương đương cấp 3 và liền kề với giá trị biên.

Ví dụ này áp dụng cho các thông số kỹ thuật đầu vào. Một kỹ thuật mạnh mẽ không kém là kiểm tra các thông số kỹ thuật đầu ra. Ví dụ: trong năm 2008, khoản khấu trừ An sinh Xã hội tối thiểu hay chính xác hơn là khoản khấu trừ Bảo hiểm Người già, Người sống sót và Người khuyết tật (OASDI) tối thiểu từ bất kỳ một phiếu lương nào được luật thuế Hoa Kỳ cho phép là 0 đô la và mức tối đa là 6324 đô la. thứ hai tương ứng với tổng thu nhập là 102.000 đô la. Do đó, khi thử nghiệm một sản phẩm trả lương, các trường hợp thử nghiệm cho khoản khấu trừ An sinh xã hội từ séc lương phải bao gồm dữ liệu đầu vào dự kiến ​​sẽ dẫn đến các khoản khấu trừ chính xác là $0 và

$6324. Ngoài ra, dữ liệu thử nghiệm phải được thiết lập để có thể dẫn đến các khoản khấu trừ ít hơn $0 hoặc nhiều hơn $6324.

Nói chung, đối với mỗi phạm vi (R1, R2) được liệt kê trong thông số kỹ thuật đầu vào hoặc đầu ra, nên chọn năm trường hợp thử nghiệm, tương ứng với các giá trị nhỏ hơn R1, bằng R1, lớn hơn R1 nhưng nhỏ hơn R2, bằng đến R2, và lớn hơn R2. Trường hợp được chỉ định rằng

một mục phải là thành viên của một tập hợp nhất định (ví dụ: đầu vào phải là một chữ cái), hai

các lớp tương đương phải được kiểm tra, một phần tử của tập xác định và không phải phần tử của tập hợp. Trong trường hợp các thông số kỹ thuật đặt ra một giá trị chính xác (ví dụ: phản hồi phải được theo sau bởi dấu #), thì lại có hai lớp tương đương, giá trị được chỉ định và bất kỳ thứ gì khác.

Việc sử dụng các lớp tương đương, cùng với phân tích giá trị biên, để kiểm tra cả thông số kỹ thuật đầu vào và thông số kỹ thuật đầu ra là một kỹ thuật có giá trị để tạo ra một tập hợp dữ liệu thử nghiệm tương đối nhỏ với khả năng phát hiện ra một số lỗi có thể vẫn còn tồn tại. ẩn nếu các kỹ thuật kém hiệu quả hơn để lựa chọn dữ liệu thử nghiệm được sử dụng.

Quy trình kiểm tra tính tương đương được tóm tắt trong Cách thực hiện Hộp 15.2.

* + 1. Functional Testing 15.11.2 Kiểm tra chức năng

An alternative form of black-box testing is to base the test data on the functionality of a code artifact. In functional testing [Howden, 1987], each item of functionality or func- tion implemented in the code artifact is identified. Typical functions in a classical mod- ule for a computerized warehouse product might be get\_next\_database\_record or determine\_whether\_quantity\_on\_hand\_is\_below\_the\_reorder\_point. In a weapons control system, a module might include the function compute\_trajectory. In a module of an operating system, one function might be determine\_whether\_file\_is\_empty.

After determining all the functions of a code artifact, test data are devised to test each function separately. Now, the functional testing is taken a step further. If the code artifact consists of a hierarchy of lower-level functions, connected by the control structures of

structured programming, then functional testing proceeds recursively. For example, if a higher-level function is of the form

Một dạng khác của kiểm thử hộp đen là dựa trên dữ liệu kiểm thử về chức năng của mã tạo tác. Trong thử nghiệm chức năng [Howden, 1987], mỗi mục chức năng hoặc chức năng được triển khai trong tạo phẩm mã được xác định. Các chức năng điển hình trong mô-đun cổ điển dành cho sản phẩm kho hàng được vi tính hóa có thể là get\_next\_database\_record hoặc detect\_whether\_quantity\_on\_hand\_is\_below\_the\_reorder\_point. Trong hệ thống điều khiển vũ khí, một mô-đun có thể bao gồm chức năng compute\_trajectory. Trong một mô-đun của hệ điều hành, một chức năng có thể là detect\_whether\_file\_is\_empty.

Sau khi xác định tất cả các chức năng của một mã tạo tác, dữ liệu thử nghiệm được tạo ra để kiểm tra từng chức năng riêng biệt. Bây giờ, thử nghiệm chức năng được tiến thêm một bước. Nếu tạo phẩm mã bao gồm một hệ thống phân cấp các chức năng cấp thấp hơn, được kết nối bởi các cấu trúc điều khiển của

lập trình có cấu trúc, sau đó kiểm tra chức năng tiến hành đệ quy. Ví dụ: nếu một hàm cấp cao hơn có dạng

<*higher-level function*> ::= **if** <*conditional expression*>

<*lower-level function* 1>;

**else**

<*lower-level function* 2>;

then, because <*conditional expression*>, <*lower-level function* 1>, and <*lower-level function* 2> have been subjected to functional testing, <*higher-level function*> can be tested using branch coverage, a glass-box technique described in Section 15.13.1. Note that this form of structural testing is a hybrid technique—the lower-level functions are tested using a black-box technique, but the higher-level functions are tested using a glass-box technique.

In practice, however, higher-level functions are not constructed in such a structured fashion from lower-level functions. Instead, the lower-level functions usually are inter- twined in some way. To determine faults in this situation, **functional analysis** is required, a somewhat complex procedure; for details, see [Howden, 1987]. A further complicat- ing factor is that functionality frequently does not coincide with code artifact boundaries. Therefore, the distinction between unit testing and integration testing becomes blurred; one code artifact cannot be tested without, at the same time, testing the other code artifacts whose functionality it uses. This problem also arises in the object-oriented paradigm when a method of one object sends a message to (invokes) a method of a different object.

The random interrelationships between code artifacts from the viewpoint of functional testing may have unacceptable consequences for management. For example, milestones and deadlines can become somewhat ill defined, making it difficult to determine the status of the product with respect to the software project management plan.

sau đó, vì <biểu thức điều kiện>, <hàm cấp dưới 1> và <hàm cấp dưới 2> đã được kiểm tra chức năng, nên <hàm cấp cao hơn> có thể được kiểm tra bằng cách sử dụng phạm vi phân nhánh, một kỹ thuật hộp thủy tinh được mô tả trong Mục 15.13.1. Lưu ý rằng hình thức kiểm tra cấu trúc này là một kỹ thuật lai—các chức năng cấp thấp hơn được kiểm tra bằng kỹ thuật hộp đen, nhưng các chức năng cấp cao hơn được kiểm tra bằng kỹ thuật hộp thủy tinh.

Tuy nhiên, trong thực tế, các chức năng cấp cao hơn không được xây dựng theo kiểu có cấu trúc như vậy từ các chức năng cấp thấp hơn. Thay vào đó, các chức năng cấp thấp hơn thường được kết hợp với nhau theo một cách nào đó. Để xác định lỗi trong tình huống này, cần phải phân tích chức năng, một quy trình hơi phức tạp; để biết chi tiết, xem [Howden, 1987]. Một yếu tố phức tạp hơn nữa là chức năng thường không trùng khớp với các ranh giới tạo tác mã. Do đó, sự khác biệt giữa kiểm thử đơn vị và kiểm thử tích hợp trở nên mờ nhạt; không thể kiểm tra một tạo phẩm mã mà không đồng thời kiểm tra các tạo phẩm mã khác có chức năng mà nó sử dụng. Vấn đề này cũng phát sinh trong mô hình hướng đối tượng khi một phương thức của một đối tượng gửi một thông điệp đến (gọi) một phương thức của một đối tượng khác.

Mối quan hệ tương hỗ ngẫu nhiên giữa các tạo phẩm mã từ quan điểm kiểm tra chức năng có thể gây ra những hậu quả không thể chấp nhận được đối với việc quản lý. Ví dụ, các cột mốc và thời hạn có thể hơi khó xác định, gây khó khăn cho việc xác định trạng thái của sản phẩm đối với kế hoạch quản lý dự án phần mềm.

###### Case Study



* 1. Black-Box Test Cases:

The MSG Foundation Case Study 15.12 Các trường hợp kiểm thử hộp đen:

Nghiên cứu điển hình về Quỹ MSG

Figures 15.13 and 15.14 contain black-box test cases for the MSG Foundation case study. First consider test cases derived from equivalence classes and boundary value analysis. The first test case in Figure 15.13 tests whether the product detects an error if the itemName of an investment does not begin with an alphabetic character. The next set of five test cases checks that an itemName consists of between 1 and 25 characters. Similar test cases check other statements in the specifications, as reflected in Figure 15.13.

Turning now to functional testing, 10 functions are listed in the specification doc- ument, as shown in Figure 15.14. An additional 11 test cases correspond to misuses of these functions.

It is important to be aware that these test cases could have been developed as soon as the analysis workflow was complete; the only reason that they appear here is that

**FIGURE 15.13**

Black-box test cases for the MSG Foundation case study derived from equivalence classes and

boundary value analysis.

**Investment data:**

Equivalence classes for itemName.

1. First character not alphabetic Error
2. < 1 character Error
3. 1 character Acceptable
4. Between 1 and 25 characters Acceptable
5. 25 characters Acceptable
6. > 25 characters Error (name too long)

Equivalence classes for itemNumber.

1. Character instead of digit Error (not a number)
2. < 12 digits Acceptable
3. 12 digits Acceptable
4. > 12 digits Error (too many digits)

Equivalence classes for estimatedAnnualReturn and expectedAnnualOperatingExpenses.

1. < $0.00 Error

1. $0.00 Acceptable
2. $0.01 Acceptable
3. Between $0.01 and $999,999,999.97 Acceptable 5. $999,999,999.98 Acceptable

6. $999,999,999.99 Acceptable

7. $1,000,000,000.00 Error

8. > $1,000,000,000.00 Error

9. Character instead of digit Error (not a number)

**Mortgage information:**

Equivalence classes for accountNumber are same as for itemNumber above. Equivalence classes for last name of mortgagees

1. First character not alphabetic Error
2. < 1 character Error
3. 1 character Acceptable
4. Between 1 and 21 characters Acceptable
5. 21 characters Acceptable
6. > 21 characters Acceptable (truncated to 21 characters)

Equivalence classes for original price of home, current family income, and mortgage balance.

1. < $0.00 Error

1. $0.00 Acceptable
2. $0.01 Acceptable
3. Between $0.01 and $999,999.98 Acceptable

5. $999,999.98 Acceptable

6. $999,999.99 Acceptable

7. $1,000,000.00 Error

8. > $1,000,000.00 Error

9. Character instead of digit Error (not a number)

**FIGURE 15.13**

(*continued*)

**FIGURE 15.14**

Functional analysis test cases for the MSG

Foundation case study.

Equivalence classes for annual property tax and annual homeowner’s premium.

1. < $0.00 Error

1. $0.00 Acceptable
2. $0.01 Acceptable
3. Between $0.01 and $99,999.98 Acceptable
4. $99,999.98 Acceptable
5. $99,999.99 Acceptable

7. $100,000.00 Error

8. > $100,000.00 Error

9. Character instead of digit Error (not a number)

The functions outlined in the specifications document are used to create test cases:

1. Add a mortgage.
2. Add an investment.
3. Modify a mortgage.
4. Modify an investment.
5. Delete a mortgage.
6. Delete an investment.
7. Update operating expenses.
8. Compute funds to purchase houses.
9. Print list of mortgages.
10. Print list of investments.

In addition to these direct tests, it is necessary to perform the following additional tests:

1. Attempt to add a mortgage that is already on file.
2. Attempt to add an investment that is already on file.
3. Attempt to delete a mortgage that is not on file.
4. Attempt to delete an investment that is not on file.
5. Attempt to modify a mortgage that is not on file.
6. Attempt to modify an investment that is not on file.
7. Attempt to delete twice a mortgage that is already on file.
8. Attempt to delete twice an investment that is already on file.
9. Attempt to update each field of a mortgage twice and check that the second version is stored.
10. Attempt to update each field of an investment twice and check that the second version is stored.
11. Attempt to update operating expenses twice and check that second version is stored.

test case selection is a topic of this chapter, rather than an earlier chapter. A major component of every test plan should be a stipulation that black-box test cases be drawn up as soon as the analysis artifacts have been approved, for use by the SQA group during the implementation workflow.

Hình 15.13 và 15.14 chứa các trường hợp thử nghiệm hộp đen cho nghiên cứu tình huống của Tổ chức MSG. Đầu tiên hãy xem xét các trường hợp thử nghiệm bắt nguồn từ các lớp tương đương và phân tích giá trị biên. Trường hợp thử nghiệm đầu tiên trong Hình 15.13 kiểm tra xem sản phẩm có phát hiện lỗi hay không nếu mụcTên khoản đầu tư không bắt đầu bằng một ký tự chữ cái. Nhóm năm trường hợp thử nghiệm tiếp theo sẽ kiểm tra xem một itemName bao gồm từ 1 đến 25 ký tự. Các trường hợp thử nghiệm tương tự kiểm tra các câu lệnh khác trong thông số kỹ thuật, như được phản ánh trong Hình 15.13.

Bây giờ chuyển sang kiểm tra chức năng, 10 chức năng được liệt kê trong tài liệu đặc tả, như trong Hình 15.14. Thêm 11 trường hợp kiểm tra tương ứng với việc sử dụng sai các chức năng này.

Điều quan trọng cần lưu ý là các trường hợp thử nghiệm này có thể đã được phát triển ngay sau khi quy trình phân tích hoàn tất; lý do duy nhất mà chúng xuất hiện ở đây là việc lựa chọn trường hợp thử nghiệm là một chủ đề của chương này, chứ không phải là một chương trước đó. Một thành phần chính của mọi kế hoạch kiểm thử phải là quy định rằng các trường hợp kiểm thử hộp đen được lập ngay sau khi các tạo phẩm phân tích đã được phê duyệt, để nhóm SQA sử dụng trong quy trình thực hiện.

* 1. **Glass-Box Unit-Testing Techniques 15.13 Kỹ thuật kiểm tra đơn vị hộp thủy tinh**

In glass-box techniques, test cases are selected on the basis of examination of the code rather than the specifications. There are a number of different forms of glass-box testing, including statement, branch, and path coverage.

Trong kỹ thuật hộp thủy tinh, các ca kiểm thử được lựa chọn trên cơ sở kiểm tra mã hơn là các thông số kỹ thuật. Có một số hình thức thử nghiệm hộp thủy tinh khác nhau, bao gồm câu lệnh, nhánh và phạm vi đường dẫn.

* + 1. Structural Testing: Statement, Branch, and Path Coverage 15.13.1 Kiểm tra cấu trúc: Phạm vi bao phủ câu lệnh, nhánh và đường dẫn

The simplest form of glass-box unit testing is **statement coverage***,* that is, running a series of test cases during which every statement is executed at least once. To keep track of which statements are still to be executed, a CASE tool keeps a record of how many times each state- ment has been executed over the series of tests; PureCoverage is an example of such a tool.

A weakness of this approach is that there is no guarantee that all outcomes of branches are properly tested. To see this, consider the code fragment of Figure 15.15. The programmer made a mistake; the compound conditional s > 1 && t == 0 should read s > 1 || t == 0. The test data shown in the figure allow the statement x = 9 to be executed without the fault being highlighted.

An improvement over statement coverage is **branch coverage**, that is, running a series of tests to ensure that all branches are tested at least once. Again, a tool usually is needed to help the tester keep track of which branches have or have not been tested; Generic Cover- age Tool (*gct*) is an example of a branch coverage tool for C programs. Techniques such as statement or branch coverage are termed **structural tests**.

The most powerful form of structural testing is **path coverage**, that is, testing all paths. As shown previously, in a product with loops, the number of paths can be very large indeed. As a result, researchers have been investigating ways of reducing the number of paths to be examined while uncovering more faults than would be possible using branch coverage. One criterion for selecting paths is to restrict test cases to **linear code sequences** [Woodward, Hedley, and Hennell, 1980]. To do this, first identify the set of points *L* from which control flow may jump. The set *L* includes entry and exit points and branch statements such as an **if** or **goto** statement. The linear code sequences are those paths that begin at an element of *L* and end at an element of *L*. The technique has been successful in that it has uncovered many faults without having to test every path.

Another way of reducing the number of paths to test is **all-definition-use-path coverage** [Rapps and Weyuker, 1985]. In this technique, each occurrence of a variable pqr, say, in the source code is labeled either as a *definition* of the variable, such as pqr = 1 or read (pqr), or a *use* of the variable, such as y = pqr + 3 or **if** (pqr < 9) errorB (). All paths between the definition of a variable and the use of that definition are identified, nowadays by means of an automatic tool. Finally, a test case is set up for each such path. All-definition-use-path coverage is an excellent test technique in that large numbers of faults frequently are detected by relatively few test cases. However, all-definition-use-path coverage has the weakness that the upper bound on the number of paths is 2*d*, where *d* is the number of decision statements (branches) in the product. Examples can be constructed exhibiting the upper bound. However, it has been shown that, for real products as opposed to artificial examples, this upper bound is not reached, and the actual number of paths is proportional to *d* [Weyuker, 1988]. In other words, the number of test cases needed for

**FIGURE 15.15**

Code fragment with test data.

**if** (s > 1 && t == 0) x = 9;

Test case: s = 2, t = 0.

**FIGURE 15.16**

Two examples of infeasible paths.

**if** (k < 2)

{

**if** (k > 3) [*should be* k > −3] x = ↑x \* k;

}

(a)

**for** (j = 0; j < 0; j++) [*should be* j < 10] total = tot↑al + value[j];

(b)

all-definition-use-path coverage generally is much smaller than the theoretical upper bound. Therefore, all-definition-use-path coverage is a practical **test case selection** technique.

When using structural testing, the tester simply might not come up with a test case that ex- ercises a specific statement, branch, or path. What may have happened is that an infeasible path (“dead code”) is in the code artifact, that is, a path that cannot possibly be executed for any input data. Figure 15.16 shows two examples of infeasible paths. In Figure 15.16(a) the programmer omitted a minus sign. If k is less than 2, then k cannot possibly be greater than 3, so the state- ment x = x \* k cannot be reached. Similarly, in Figure 15.16(b), j is never less than 0, so the statement total = total + value[j] can never be reached; the programmer had intended the test to be j < 10, but made a typing mistake. A tester using statement coverage would soon realize that neither statement could be reached and the faults would be found.

Hình thức đơn giản nhất của thử nghiệm đơn vị hộp thủy tinh là phạm vi câu lệnh, nghĩa là chạy một loạt các trường hợp thử nghiệm trong đó mỗi câu lệnh được thực hiện ít nhất một lần. Để theo dõi những câu lệnh nào vẫn còn được thực thi, một công cụ CASE sẽ ghi lại số lần mỗi câu lệnh đã được thực hiện trong một loạt các thử nghiệm; PureCoverage là một ví dụ về một công cụ như vậy.

Một điểm yếu của phương pháp này là không có gì đảm bảo rằng tất cả các kết quả của các nhánh đều được kiểm tra đúng cách. Để thấy điều này, hãy xem xét đoạn mã của Hình 15.15. Lập trình viên đã mắc lỗi; điều kiện ghép s > 1 && t == 0 nên đọc là s > 1 || t == 0. Dữ liệu thử nghiệm được hiển thị trong hình cho phép câu lệnh x = 9 được thực thi mà không có lỗi được đánh dấu.

Một cải tiến so với phạm vi câu lệnh là phạm vi nhánh, nghĩa là chạy một loạt các thử nghiệm để đảm bảo rằng tất cả các nhánh đều được kiểm tra ít nhất một lần. Một lần nữa, một công cụ thường là cần thiết để giúp người kiểm tra theo dõi những nhánh nào đã được kiểm tra hoặc chưa được kiểm tra; Công cụ bảo hiểm chung (gct) là một ví dụ về công cụ bảo hiểm chi nhánh cho các chương trình C. Các kỹ thuật như câu lệnh hoặc bao phủ nhánh được gọi là kiểm thử cấu trúc.

Hình thức kiểm tra cấu trúc hiệu quả nhất là bao phủ đường dẫn, nghĩa là kiểm tra tất cả các đường dẫn. Như đã trình bày trước đây, trong một sản phẩm có vòng lặp, số lượng đường dẫn thực sự có thể rất lớn. Do đó, các nhà nghiên cứu đã nghiên cứu các cách giảm số lượng đường dẫn được kiểm tra trong khi phát hiện ra nhiều lỗi hơn so với khả năng sử dụng phạm vi chi nhánh. Một tiêu chí để chọn đường dẫn là hạn chế các trường hợp thử nghiệm đối với các chuỗi mã tuyến tính [Woodward, Hedley và Hennell, 1980]. Để thực hiện điều này, đầu tiên hãy xác định tập hợp các điểm L mà từ đó luồng điều khiển có thể nhảy tới. Tập hợp L bao gồm các điểm vào và điểm ra và các câu lệnh rẽ nhánh như câu lệnh if hoặc goto. Chuỗi mã tuyến tính là những đường dẫn bắt đầu tại một phần tử của L và kết thúc tại một phần tử của L. Kỹ thuật này đã thành công ở chỗ nó đã phát hiện ra nhiều lỗi mà không cần phải kiểm tra mọi đường dẫn.

Một cách khác để giảm số lượng đường dẫn để kiểm tra là bao phủ đường dẫn sử dụng tất cả định nghĩa [Rapps và Weyuker, 1985]. Trong kỹ thuật này, mỗi lần xuất hiện của một biến pqr, chẳng hạn, trong mã nguồn được gắn nhãn hoặc là định nghĩa của biến, chẳng hạn như pqr = 1 hoặc đọc (pqr), hoặc sử dụng biến, chẳng hạn như y = pqr + 3 hoặc nếu (pqr < 9) errorB (). Ngày nay, tất cả các đường dẫn giữa định nghĩa của một biến và việc sử dụng định nghĩa đó đều được xác định bằng một công cụ tự động. Cuối cùng, một trường hợp thử nghiệm được thiết lập cho mỗi đường dẫn như vậy. Phạm vi bao phủ đường dẫn sử dụng tất cả định nghĩa là một kỹ thuật kiểm tra xuất sắc trong đó số lượng lớn lỗi thường xuyên được phát hiện bởi tương đối ít trường hợp kiểm tra. Tuy nhiên, phạm vi bao phủ đường dẫn sử dụng tất cả định nghĩa có điểm yếu là giới hạn trên của số lượng đường dẫn là 2d, trong đó d là số lượng câu lệnh quyết định (nhánh) trong sản phẩm. Các ví dụ có thể được xây dựng thể hiện giới hạn trên. Tuy nhiên, người ta đã chỉ ra rằng, đối với các sản phẩm thực trái ngược với các ví dụ nhân tạo, giới hạn trên không đạt được và số lượng đường dẫn thực tế tỷ lệ với d [Weyuker, 1988]. Nói cách khác, số lượng trường hợp thử nghiệm cần thiết cho phạm vi bao phủ đường dẫn sử dụng tất cả định nghĩa nói chung nhỏ hơn nhiều so với giới hạn trên lý thuyết. Do đó, phạm vi bao phủ đường dẫn sử dụng tất cả định nghĩa là một kỹ thuật lựa chọn trường hợp thử nghiệm thực tế.

Khi sử dụng kiểm thử cấu trúc, người kiểm thử có thể không nghĩ ra một trường hợp kiểm thử thực hiện một câu lệnh, nhánh hoặc đường dẫn cụ thể. Điều có thể đã xảy ra là một đường dẫn không khả thi (“mã chết”) nằm trong mã tạo tác, nghĩa là một đường dẫn không thể được thực thi đối với bất kỳ dữ liệu đầu vào nào. Hình 15.16 cho thấy hai ví dụ về các đường dẫn không khả thi. Trong Hình 15.16(a) người lập trình đã bỏ qua dấu trừ. Nếu k nhỏ hơn 2, thì k không thể lớn hơn 3, do đó không thể đạt được mệnh đề x = x \* k. Tương tự như vậy, trong Hình 15.16(b), j không bao giờ nhỏ hơn 0, do đó không bao giờ đạt được câu lệnh total = total + value[j]; lập trình viên đã dự định thử nghiệm là j < 10, nhưng đã mắc lỗi đánh máy. Người kiểm tra sử dụng phạm vi câu lệnh sẽ sớm nhận ra rằng không thể đạt được câu lệnh nào và các lỗi sẽ được tìm thấy.

* + 1. Complexity Metrics 15.13.2 Số liệu về độ phức tạp

The quality assurance viewpoint provides another approach to glass-box unit testing. Sup- pose a manager is told that code artifact m1 is more complex than code artifact m2. Irrespective of the precise way in which the term *complex* is defined, the manager intuitively believes that m1 is likely to have more faults than m2. Following this idea, computer scien- tists have developed a number of metrics of software **complexity** as an aid in determining which code artifacts are most likely to have faults. If the complexity of a code artifact is found to be unreasonably high, a manager may direct that the artifact be redesigned and reimplemented on the grounds that it probably is less costly and faster to start from scratch than to attempt to debug a fault-prone code artifact.

A simple metric for predicting numbers of faults is lines of code. The underlying as- sumption is that there is a constant probability, *p*, that a line of code contains a fault. If a tester believes that, on average, a line of code has a 2 percent chance of containing a fault, and the artifact under test is 100 lines long, then this implies that the artifact is expected to contain two faults; and an artifact that is twice as long is likely to have four faults. Basili and Hutchens [1983] as well as Takahashi and Kamayachi [1985] showed that the number of faults indeed is related to the size of the product as a whole.

Attempts have been made to find more sophisticated predictors of faults based on measures of product complexity. A typical contender is McCabe’s [1976] measure of **cyclomatic complexity**, the number of binary decisions (predicates) plus 1. As described in Section 14.15, the cyclomatic complexity essentially is the number of branches in the

code artifact. Accordingly, cyclomatic complexity can be used as a metric for the number of test cases needed for branch coverage of a code artifact. This is the basis for so-called **structured testing** [Watson and McCabe, 1996].

McCabe’s metric can be computed almost as easily as lines of code. In some cases, it has been shown to be a good metric for predicting faults; the higher the value of *M*, the greater is the chance that a code artifact contains a fault. For example, Walsh [1979] analyzed 276 modules in the Aegis system, a shipboard combat system. Measuring the cyclomatic com- plexity, *M,* he found that 23 percent of the modules with *M* greater than or equal to 10 had 53 percent of the faults detected. In addition, the modules with *M* greater than or equal to 10 had 21 percent more faults per line of code than the modules with smaller *M* values. However, the validity of McCabe’s metric has been questioned seriously on both theoretical grounds and on the basis of the many different experiments cited in [Shepperd and Ince, 1994].

Musa, Iannino, and Okumoto [1987] analyzed the data available on fault densities. They concluded that most complexity metrics, including McCabe’s, show a high correlation with the number of lines of code or, more precisely, the number of deliverable, executable source instructions. In other words, when researchers measure what they believe to be the com- plexity of a code artifact or a product, the result they obtain may be largely a reflection of the number of lines of code, a measure that correlates strongly with the number of faults. In addition, complexity metrics provide little improvement over lines of code for predicting fault rates. Other problems with complexity are discussed in [Shepperd and Ince, 1994].

Quan điểm đảm bảo chất lượng cung cấp một cách tiếp cận khác đối với thử nghiệm đơn vị hộp thủy tinh. Giả sử người quản lý được thông báo rằng mã tạo tác m1 phức tạp hơn mã tạo tác m2. Bất kể cách chính xác mà thuật ngữ phức tạp được định nghĩa, người quản lý trực giác tin rằng m1 có khả năng có nhiều lỗi hơn m2. Theo ý tưởng này, các nhà khoa học máy tính đã phát triển một số thước đo về độ phức tạp của phần mềm như một công cụ hỗ trợ trong việc xác định mã tạo phẩm nào có nhiều khả năng có lỗi nhất. Nếu độ phức tạp của một mã tạo tác được phát hiện là cao một cách bất hợp lý, người quản lý có thể chỉ đạo rằng tạo phẩm đó được thiết kế lại và triển khai lại với lý do rằng việc bắt đầu lại từ đầu có thể ít tốn kém hơn và nhanh hơn so với cố gắng gỡ lỗi một mã dễ bị lỗi hiện vật.

Một thước đo đơn giản để dự đoán số lỗi là các dòng mã. Giả định cơ bản là có một xác suất không đổi, p, rằng một dòng mã có lỗi. Nếu một người kiểm tra tin rằng, trung bình, một dòng mã có 2% khả năng chứa lỗi và tạo phẩm được kiểm tra dài 100 dòng, thì điều này ngụ ý rằng tạo phẩm dự kiến ​​sẽ chứa hai lỗi; và một đồ tạo tác dài gấp đôi có khả năng mắc bốn lỗi. Basili và Hutchens [1983] cũng như Takahashi và Kamayachi [1985] đã chỉ ra rằng số lỗi thực sự có liên quan đến kích thước của toàn bộ sản phẩm.

Các nỗ lực đã được thực hiện để tìm ra các yếu tố dự đoán lỗi phức tạp hơn dựa trên các biện pháp đo lường độ phức tạp của sản phẩm. Một ứng cử viên điển hình là phép đo độ phức tạp của chu trình của McCabe [1976], số lượng các quyết định nhị phân (các vị từ) cộng với 1. Như được mô tả trong Phần 14.15, độ phức tạp của chu trình về cơ bản là số lượng nhánh trong

hiện vật mã. Theo đó, độ phức tạp của chu trình có thể được sử dụng làm thước đo cho số lượng trường hợp thử nghiệm cần thiết để bao phủ nhánh của một tạo phẩm mã. Đây là cơ sở cho cái gọi là thử nghiệm có cấu trúc [Watson và McCabe, 1996].

Số liệu của McCabe có thể được tính toán gần như dễ dàng như các dòng mã. Trong một số trường hợp, nó đã được chứng minh là một thước đo tốt để dự đoán lỗi; giá trị của M càng cao thì khả năng tạo tác mã chứa lỗi càng lớn. Ví dụ, Walsh [1979] đã phân tích 276 mô-đun trong hệ thống Aegis, một hệ thống chiến đấu trên tàu. Khi đo độ phức tạp theo chu kỳ, M, ông phát hiện ra rằng 23% mô-đun có M lớn hơn hoặc bằng 10 có 53% lỗi được phát hiện. Ngoài ra, các mô-đun có M lớn hơn hoặc bằng 10 có nhiều lỗi hơn 21% trên mỗi dòng mã so với các mô-đun có giá trị M nhỏ hơn. Tuy nhiên, tính hợp lệ của thước đo McCabe đã bị nghi ngờ nghiêm trọng trên cả cơ sở lý thuyết và trên cơ sở của nhiều thí nghiệm khác nhau được trích dẫn trong [Shepperd và Ince, 1994].

Musa, Iannino và Okumoto [1987] đã phân tích dữ liệu có sẵn về mật độ đứt gãy. Họ kết luận rằng hầu hết các chỉ số phức tạp, bao gồm cả của McCabe, cho thấy mối tương quan cao với số lượng dòng mã hoặc chính xác hơn là số lượng hướng dẫn nguồn có thể thực thi được. Nói cách khác, khi các nhà nghiên cứu đo lường những gì họ tin là độ phức tạp của mã tạo tác hoặc sản phẩm, kết quả họ thu được có thể phần lớn phản ánh số lượng dòng mã, một phép đo có tương quan chặt chẽ với số lượng dòng mã. lỗi lầm. Ngoài ra, các phép đo độ phức tạp cung cấp một chút cải tiến so với các dòng mã để dự đoán tỷ lệ lỗi. Các vấn đề phức tạp khác được thảo luận trong [Shepperd và Ince, 1994].

* 1. Code Walkthroughs and Inspections 15.14 Hướng dẫn và kiểm tra mã

Section 6.2 made a strong case for the use of walkthroughs and inspections in general. The same arguments hold for code walkthroughs and inspections. In brief, the fault-detecting power of these two non-execution-based techniques leads to rapid, thorough, and early fault detection. The additional time required for code walkthroughs or inspections is more than repaid by increased productivity due to the presence of fewer faults when integration is performed. Furthermore, code inspections have led to a reduction of up to 95 percent in corrective maintenance costs [Crossman, 1982].

Another reason why code inspections should be performed is that the alternative, execution-based testing (test cases), can be extremely expensive in two ways. First, it is time consuming. Second, inspections lead to detection and correction of faults earlier in the life cycle than with execution-based testing. As reflected in Figure 1.6, the earlier a fault is detected and corrected, the less it costs. An extreme case of the high cost of running test cases is that 80 percent of the budget for the software of the NASA Apollo program was consumed by testing [Dunn, 1984].

Further arguments in favor of walkthroughs and inspections are given in Section 15.15.

Phần 6.2 đã đưa ra một trường hợp mạnh mẽ cho việc sử dụng hướng dẫn và kiểm tra nói chung. Các lập luận tương tự áp dụng cho các hướng dẫn và kiểm tra mã. Tóm lại, khả năng phát hiện lỗi của hai kỹ thuật không dựa trên thực thi này giúp phát hiện lỗi nhanh chóng, kỹ lưỡng và sớm. Thời gian bổ sung cần thiết cho việc hướng dẫn hoặc kiểm tra mã được đền đáp nhiều hơn bằng năng suất tăng lên do có ít lỗi hơn khi thực hiện tích hợp. Hơn nữa, việc kiểm tra mã đã giúp giảm tới 95% chi phí bảo trì khắc phục [Crossman, 1982].

Một lý do khác khiến việc kiểm tra mã nên được thực hiện là việc kiểm tra thay thế, dựa trên thực thi (các trường hợp kiểm tra), có thể cực kỳ tốn kém theo hai cách. Đầu tiên, nó tốn thời gian. Thứ hai, kiểm tra giúp phát hiện và sửa lỗi sớm hơn trong vòng đời so với kiểm tra dựa trên thực thi. Như được phản ánh trong Hình 1.6, lỗi được phát hiện và sửa chữa càng sớm thì chi phí càng ít. Một trường hợp cực đoan về chi phí chạy thử nghiệm cao là 80% ngân sách dành cho phần mềm của chương trình Apollo của NASA đã được sử dụng cho thử nghiệm [Dunn, 1984].

Các lập luận khác ủng hộ hướng dẫn và kiểm tra được đưa ra trong Phần 15.15.

* 1. Comparison of Unit-Testing Techniques 15.15 So sánh các kỹ thuật kiểm tra đơn vị

A number of studies have compared strategies for unit testing. Myers [1978a] compared black-box testing, a combination of black-box and glass-box testing, and three-person code walkthroughs. The experiment was performed using 59 highly experienced programmers test- ing the same product. All three techniques were equally effective in finding faults, but code walkthroughs proved to be less cost effective than the other two techniques. Hwang [1981]

compared black-box testing, glass-box testing, and code reading by one person. All three techniques were found to be equally effective, with each technique having its own strengths and weaknesses.

A major experiment was conducted by Basili and Selby [1987]. The techniques com- pared were the same as in Hwang’s experiment: black-box testing, glass-box testing, and one-person code reading. The subjects were 32 professional programmers and 42 advanced students. Each tested three products, using each testing technique once. Fractional facto- rial design [Basili and Weiss, 1984] was used to compensate for the different ways the products were tested by different participants; no participant tested the same product in more than one way. Different results were obtained from the two groups of participants. The professional programmers detected more faults with code reading than with the other two techniques, and the fault detection rate was faster. Two groups of advanced students participated. In one group, no significant difference was found among the three techniques; in the other, code reading and black-box testing were equally good and both outperformed glass-box testing. However, the rate at which students detected faults was the same for all techniques. Overall, code reading led to the detection of more interface faults than the other two techniques, whereas black-box testing was most successful at finding control faults.

In Basili and Selby’s experiment, code inspection was at least as successful at detecting faults as glass-box and black-box testing. Most subsequent experiments have shown that black-box testing and glass-box testing are more efficient or more effective than inspections [Runeson et al., 2006]. However, some studies have shown that test cases and inspections tend to find different kinds of faults. In other words, the two techniques are complementary, and both need to be utilized on every software product.

A development technique that makes use of this conclusion is the Cleanroom software development technique.

Một số nghiên cứu đã so sánh các chiến lược để thử nghiệm đơn vị. Myers [1978a] đã so sánh kiểm thử hộp đen, sự kết hợp giữa kiểm thử hộp đen và hộp thủy tinh, và hướng dẫn mã ba người. Thử nghiệm được thực hiện bằng cách sử dụng 59 lập trình viên có kinh nghiệm cao đang thử nghiệm cùng một sản phẩm. Cả ba kỹ thuật đều có hiệu quả như nhau trong việc tìm lỗi, nhưng mã hướng dẫn được chứng minh là ít hiệu quả hơn so với hai kỹ thuật còn lại. Hwang [1981]

so sánh kiểm thử hộp đen, kiểm thử hộp thủy tinh và đọc mã bởi một người. Cả ba kỹ thuật đều có hiệu quả như nhau, với mỗi kỹ thuật đều có điểm mạnh và điểm yếu riêng.

Một thí nghiệm lớn được thực hiện bởi Basili và Selby [1987]. Các kỹ thuật được so sánh giống như trong thí nghiệm của Hwang: kiểm thử hộp đen, kiểm thử hộp thủy tinh và đọc mã một người. Các đối tượng là 32 lập trình viên chuyên nghiệp và 42 sinh viên tiên tiến. Mỗi người đã thử nghiệm ba sản phẩm, sử dụng mỗi kỹ thuật thử nghiệm một lần. Thiết kế thừa số phân số [Basili và Weiss, 1984] được sử dụng để bù đắp cho những cách khác nhau mà sản phẩm được thử nghiệm bởi những người tham gia khác nhau; không có người tham gia nào thử nghiệm cùng một sản phẩm theo nhiều cách. Kết quả khác nhau thu được từ hai nhóm người tham gia. Các lập trình viên chuyên nghiệp phát hiện nhiều lỗi hơn khi đọc mã so với hai kỹ thuật còn lại và tốc độ phát hiện lỗi nhanh hơn. Hai nhóm học sinh tiên tiến tham gia. Trong một nhóm, không có sự khác biệt đáng kể nào được tìm thấy giữa ba kỹ thuật; mặt khác, đọc mã và kiểm tra hộp đen đều tốt như nhau và cả hai đều vượt trội so với kiểm tra hộp thủy tinh. Tuy nhiên, tốc độ học sinh phát hiện lỗi là như nhau đối với tất cả các kỹ thuật. Nhìn chung, việc đọc mã dẫn đến việc phát hiện nhiều lỗi giao diện hơn so với hai kỹ thuật còn lại, trong khi kiểm thử hộp đen thành công nhất trong việc tìm ra các lỗi điều khiển.

Trong thí nghiệm của Basili và Selby, kiểm tra mã ít nhất cũng thành công trong việc phát hiện lỗi như kiểm tra hộp thủy tinh và hộp đen. Hầu hết các thí nghiệm sau đó đã chỉ ra rằng kiểm thử hộp đen và kiểm thử hộp thủy tinh hiệu quả hơn hoặc hiệu quả hơn kiểm tra [Runeson và cộng sự, 2006]. Tuy nhiên, một số nghiên cứu đã chỉ ra rằng các trường hợp thử nghiệm và kiểm tra có xu hướng tìm ra các loại lỗi khác nhau. Nói cách khác, hai kỹ thuật này bổ sung cho nhau và cả hai đều cần được sử dụng trên mọi sản phẩm phần mềm.

Một kỹ thuật phát triển sử dụng kết luận này là kỹ thuật phát triển phần mềm Phòng sạch.

* 1. Cleanroom 15.16 Phòng sạch

The **Cleanroom** technique [Linger, 1994] is a combination of a number of different soft- ware development techniques, including an incremental life-cycle model, formal tech- niques for analysis and design, and non-execution-based unit-testing techniques, such as code reading [Mills, Dyer, and Linger, 1987] and code walkthroughs and inspections (Sec- tion 15.14). A critical aspect of Cleanroom is that a code artifact is not compiled until it has passed inspection. That is, a code artifact should be compiled only after non-execution- based testing has been successfully completed.

The technique has had a number of great successes. For example, a prototype auto- mated documentation system was developed for the U.S. Naval Underwater Systems Center using Cleanroom [Trammel, Binder, and Snyder, 1992]. Altogether 18 faults were detected while the design underwent “functional verification,” a review process in which correctness-proving techniques are employed (Section 6.5). Informal proofs such as the one presented in Section 6.5.1 were used as much as possible; full mathematical proofs were developed only when participants were unsure of the correctness of the portion of the design being inspected. Another 19 faults were detected during walkthroughs of the 1820 lines of FoxBASE code; when the code was then compiled, there were no compilation errors. Furthermore, there were no failures at execution time. This is an additional indica- tion of the power of non-execution-based testing techniques.

This certainly is an impressive result. But, as has been pointed out, results that apply to small-scale software products cannot necessarily be scaled up to large-scale software. In the case of Cleanroom, however, results for larger products also are impressive. The relevant metric is the **testing fault rate**, that is, the total number of faults detected per KLOC (thousand lines of code), a relatively common metric in the software industry. Yet, there is a critical difference in the way this metric is computed when Cleanroom is used as opposed to traditional development techniques.

As pointed out in Section 6.6, when traditional development techniques are used, a code artifact is tested informally by its programmer while it is being developed and there- after it is tested methodically by the SQA group. Faults detected by the programmer while developing the code are not recorded. However, from the time the artifact leaves the private workspace of the programmer and is handed over to the SQA group for execution-based and non-execution-based testing, a tally is kept of the number of faults detected. In con- trast, when Cleanroom is used, “testing faults” are counted from the time of compilation. Fault counting then continues through execution-based testing. In other words, when tradi- tional development techniques are used, faults detected informally by the programmer do not count toward the testing fault rate. When Cleanroom is used, faults detected during the inspections and other non-execution-based testing procedures that precede compilation are recorded, but they do not count toward the testing fault rate.

A report on 17 Cleanroom products appears in [Linger, 1994]. For example, Cleanroom was used to develop the 350,000-line Ericsson Telecom OS32 operating system. The prod- uct was developed in 18 months by a team of 70. The testing fault rate was only 1.0 fault per KLOC. Another product was the prototype automated documentation system described previously; the testing fault rate was 0.0 faults per KLOC for the 1820-line program. The 17 products together total nearly 1 million lines of code. The weighted average testing fault rate was 2.3 faults per KLOC, which Linger describes as a remarkable quality achievement. That praise certainly is no exaggeration.

Kỹ thuật Phòng sạch [Linger, 1994] là sự kết hợp của một số kỹ thuật phát triển phần mềm khác nhau, bao gồm mô hình vòng đời gia tăng, các kỹ thuật chính thức để phân tích và thiết kế, và các kỹ thuật kiểm thử đơn vị không dựa trên thực thi, chẳng hạn như đọc mã [Mills, Dyer và Linger, 1987] và hướng dẫn và kiểm tra mã (Phần 15.14). Một khía cạnh quan trọng của Phòng sạch là một tạo phẩm mã không được biên dịch cho đến khi nó vượt qua kiểm tra. Nghĩa là, một tạo phẩm mã chỉ nên được biên dịch sau khi thử nghiệm không dựa trên thực thi đã được hoàn tất thành công.

Kỹ thuật này đã có một số thành công lớn. Ví dụ, một nguyên mẫu hệ thống tài liệu tự động đã được phát triển cho Trung tâm Hệ thống Dưới nước của Hải quân Hoa Kỳ sử dụng Phòng sạch [Trammel, Binder, và Snyder, 1992]. Tổng cộng 18 lỗi đã được phát hiện trong khi thiết kế trải qua quá trình “xác minh chức năng”, một quy trình xem xét trong đó các kỹ thuật chứng minh tính đúng đắn được sử dụng (Phần 6.5). Các bằng chứng không chính thức như được trình bày trong Phần 6.5.1 đã được sử dụng nhiều nhất có thể; bằng chứng toán học đầy đủ chỉ được phát triển khi những người tham gia không chắc chắn về tính đúng đắn của phần thiết kế được kiểm tra. 19 lỗi khác đã được phát hiện trong quá trình xem qua 1820 dòng mã FoxBASE; khi mã được biên dịch, không có lỗi biên dịch. Hơn nữa, không có lỗi tại thời điểm thực hiện. Đây là một dấu hiệu bổ sung về sức mạnh của các kỹ thuật kiểm thử không dựa trên thực thi.

Đây chắc chắn là một kết quả ấn tượng. Nhưng, như đã được chỉ ra, các kết quả áp dụng cho các sản phẩm phần mềm quy mô nhỏ không nhất thiết phải được mở rộng thành phần mềm quy mô lớn. Tuy nhiên, trong trường hợp Phòng sạch, kết quả cho các sản phẩm lớn hơn cũng rất ấn tượng. Số liệu liên quan là tỷ lệ lỗi thử nghiệm, nghĩa là tổng số lỗi được phát hiện trên mỗi KLOC (nghìn dòng mã), một số liệu tương đối phổ biến trong ngành công nghiệp phần mềm. Tuy nhiên, có một sự khác biệt quan trọng trong cách tính số liệu này khi Cleanroom được sử dụng trái ngược với các kỹ thuật phát triển truyền thống.

Như đã chỉ ra trong Phần 6.6, khi các kỹ thuật phát triển truyền thống được sử dụng, một tạo phẩm mã được kiểm tra không chính thức bởi người lập trình của nó trong khi nó đang được phát triển và sau đó nó được kiểm tra một cách có phương pháp bởi nhóm SQA. Các lỗi do lập trình viên phát hiện trong khi phát triển mã không được ghi lại. Tuy nhiên, kể từ thời điểm phần mềm rời khỏi không gian làm việc riêng tư của lập trình viên và được bàn giao cho nhóm SQA để kiểm tra dựa trên thực thi và không dựa trên thực thi, số lượng lỗi được phát hiện sẽ được lưu giữ. Ngược lại, khi Cleanroom được sử dụng, “các lỗi kiểm tra” được tính từ thời điểm biên dịch. Việc đếm lỗi sau đó tiếp tục thông qua thử nghiệm dựa trên thực thi. Nói cách khác, khi các kỹ thuật phát triển truyền thống được sử dụng, các lỗi được lập trình viên phát hiện một cách không chính thức không được tính vào tỷ lệ lỗi thử nghiệm. Khi sử dụng Phòng sạch, các lỗi được phát hiện trong quá trình kiểm tra và các quy trình kiểm tra không dựa trên thực thi khác trước khi biên dịch sẽ được ghi lại, nhưng chúng không được tính vào tỷ lệ lỗi kiểm tra.

Một báo cáo về 17 sản phẩm phòng sạch xuất hiện trong [Linger, 1994]. Ví dụ: Phòng sạch được sử dụng để phát triển hệ điều hành 350.000 dòng của Ericsson Telecom OS32. Sản phẩm được phát triển trong 18 tháng bởi một nhóm gồm 70 người. Tỷ lệ lỗi thử nghiệm chỉ là 1,0 lỗi trên mỗi KLOC. Một sản phẩm khác là hệ thống tài liệu tự động nguyên mẫu đã được mô tả trước đây; tỷ lệ lỗi thử nghiệm là 0,0 lỗi trên mỗi KLOC cho chương trình 1820 dòng. Tổng cộng 17 sản phẩm có gần 1 triệu dòng mã. Tỷ lệ lỗi thử nghiệm trung bình có trọng số là 2,3 lỗi trên mỗi KLOC, mà Linger mô tả là một thành tích chất lượng đáng chú ý. Lời khen ngợi đó chắc chắn không phải là cường điệu.

* 1. Potential Problems When Testing Objects 15.17 Các vấn đề tiềm ẩn khi kiểm tra đối tượng

One of the many reasons put forward for using the object-oriented paradigm is that it reduces the need for testing. Reuse via inheritance is a major strength of the paradigm; once a class has been tested, the argument goes, there is no need to retest it. Furthermore, new methods defined within a subclass of such a tested class have to be tested, but inherited methods need no further testing.

In fact, both claims are only partially true. In addition, the testing of objects poses cer- tain problems that are specific to object orientation. These issues are discussed here.

To begin, it is necessary to clarify an issue regarding the testing of classes and of objects. As explained in Section 7.7, a class is an abstract data type that supports inheritance, and an object is an instance of a class. That is, a class has no concrete realization, whereas an object is a physical piece of code executing within a specific environment. Therefore, it is impos- sible to perform execution-based testing on a class; only non-execution-based testing, such as an inspection, can be done.

Information hiding and the fact that many methods consist of relatively few lines of code can have a significant impact on testing. First, consider a product developed using the

classical paradigm. Nowadays, such a product generally consists of modules of roughly 50 executable instructions. The interface between a module and the rest of the product is the argument list. Arguments are of two kinds, input arguments supplied to the module when it is invoked and output arguments returned by the module when it returns control to the calling module. Testing a module consists of supplying values to the input arguments and invoking the module and then comparing the values of the output arguments to the pre- dicted results of the test.

In contrast, a “typical” object contains perhaps 30 methods, many of which are rel- atively small, frequently just two or three executable statements [Wilde, Matthews, and Huitt, 1993]. These methods do not return a value to the caller but rather change the state of the object. That is, these methods modify attributes (state variables) of the object. The difficulty here is that, to test that the change of state has been performed correctly, it is nec- essary to send additional messages to the object. For example, consider the bank account object described in Section 1.9. The effect of method deposit is to increase the value of state variable accountBalance. However, as a consequence of information hiding, the only way to test whether a particular deposit method has been executed correctly is to invoke method determineBalance both before and after invoking method deposit and see how the bank balance changes.

The situation is worse if the object does not include methods that can be invoked to de- termine the values of all the state variables. One alternative is to include additional methods for this purpose, and then use conditional compilation to ensure that they are unavailable except for testing purposes (in C++, this can be implemented using #ifdef). The test plan (Section 9.6) should stipulate that the value of every state variable be accessible during testing. To satisfy this requirement, additional methods that return the values of the state variables may have to be added to the relevant classes during the design workflow. As a result, it is possible to test the effect of invoking a specific method of an object by querying the value of the applicable state variable.

Surprisingly enough, an inherited method still may have to be tested. That is, even if a method has been adequately tested, it may require thorough testing when inherited, unchanged, by a subclass. To see this latter point, consider the class hierarchy shown in Figure 15.17. Two methods are defined in the base class **RootedTreeClass**, namely, displayNodeContents and printRoutine, where method displayNodeContents uses method printRoutine.

Next consider subclass **BinaryTreeClass**. This subclass inherits method printRoutine from its base class **RootedTreeClass**. In addition, a new method, displayNodeContents, is defined that overrides the method defined in **RootedTreeClass**. This new method still uses printRoutine. In Java notation, BinaryTreeClass.displayNodeContents uses RootedTreeClass.printRoutine.

Now consider the subclass **BalancedBinaryTreeClass**. This subclass inherits method displayNodeContents from its superclass **BinaryTreeClass**. However, a new method printRoutine is defined that overrides the one defined in **RootedTreeClass**. When displayNodeContents uses printRoutine within the context of **Balanced- BinaryTreeClass**, the scope rules of C++ and Java specify that the local version of printRoutine is to be used. In Java notation, when method BinaryTreeClass.display- NodeContents is invoked within the lexical scope of **BalancedBinaryTreeClass**, it uses method BalancedBinaryTreeClass.printRoutine.

**FIGURE 15.17**

A Java implementation of a tree hierarchy.

**class RootedTreeClass**

{

…

**void** displayNodeContents (Node a);

**void** printRoutine (Node b);

//

// *method* displayNodeContents *uses method* printRoutine

//

…

}

**class BinaryTreeClass extends RootedTreeClass**

{

…

**void** displayNodeContents (Node a);

//

// *method* displayNodeContents *defined in this class uses*

// *method* printRoutine *inherited from* **ClassRootedTree**

//

…

}

**class BalancedBinaryTreeClass extends BinaryTreeClass**

{

…

**void** printRoutine (Node b);

//

// *method* displayNodeContents (*inherited from* BinaryTreeClass) *uses this*

// *local version of* printRoutine *within class* **BalancedBinaryTreeClass**

//

…

}

Therefore, the actual code (method printRoutine) executed when displayNodeContents is invoked within instantiations of **BinaryTreeClass** is different from what is executed when displayNodeContents is invoked within instantiations of **BalancedBinaryTreeClass**. This holds notwithstanding that the method displayNodeContents itself is inherited, un- changed, by **BalancedBinaryTreeClass** from **BinaryTreeClass**. Therefore, even if method displayNodeContents has been thoroughly tested within a **BinaryTreeClass** object, it has to be retested from scratch when reused within a **BalancedBinaryTreeClass** environment. To make matters even more complex, there are theoretical reasons why it needs to be retested with different test cases [Perry and Kaiser, 1990].

It must be pointed out immediately that these complications are no reason to abandon the object-oriented paradigm. First, they arise only through the interaction of methods (dis- playNodeContents and printRoutine in the example). Second, it is possible to determine when this retesting is needed [Harrold, McGregor, and Fitzpatrick, 1992].

Suppose an instantiation of a class has been thoroughly tested. Any new or redefined methods of a subclass then need to be tested, together with methods flagged for retesting

because of their interaction with other methods. In short, then, the claim that use of the object-oriented paradigm reduces the need for testing largely is true.

Some management implications of unit testing now are considered.

Một trong nhiều lý do được đưa ra để sử dụng mô hình hướng đối tượng là nó làm giảm nhu cầu kiểm tra. Tái sử dụng thông qua kế thừa là sức mạnh chính của mô hình; khi một lớp đã được kiểm tra, đối số sẽ tiếp tục, không cần phải kiểm tra lại nó. Hơn nữa, các phương thức mới được định nghĩa trong một lớp con của một lớp đã kiểm tra như vậy phải được kiểm tra, nhưng các phương thức kế thừa không cần kiểm tra thêm.

Trên thực tế, cả hai tuyên bố đều chỉ đúng một phần. Ngoài ra, việc kiểm tra các đối tượng đặt ra một số vấn đề cụ thể đối với hướng đối tượng. Những vấn đề này được thảo luận ở đây.

Để bắt đầu, cần phải làm rõ một vấn đề liên quan đến việc kiểm thử các lớp và các đối tượng. Như đã giải thích trong Phần 7.7, một lớp là một kiểu dữ liệu trừu tượng hỗ trợ tính kế thừa và một đối tượng là một thể hiện của một lớp. Nghĩa là, một lớp không có hiện thực cụ thể, trong khi một đối tượng là một đoạn mã thực thi trong một môi trường cụ thể. Do đó, không thể thực hiện kiểm thử dựa trên thực thi trên một lớp; chỉ có thể thực hiện thử nghiệm không dựa trên thực thi, chẳng hạn như kiểm tra.

Ẩn thông tin và thực tế là nhiều phương pháp bao gồm tương đối ít dòng mã có thể có tác động đáng kể đến quá trình thử nghiệm. Đầu tiên, hãy xem xét một sản phẩm được phát triển bằng cách sử dụng mô hình cổ điển. Ngày nay, một sản phẩm như vậy thường bao gồm các mô-đun khoảng 50 lệnh thực thi. Giao diện giữa một mô-đun và phần còn lại của sản phẩm là danh sách đối số. Các đối số có hai loại, các đối số đầu vào được cung cấp cho mô-đun khi nó được gọi và các đối số đầu ra được mô-đun trả về khi nó trả lại quyền điều khiển cho mô-đun đang gọi. Thử nghiệm một mô-đun bao gồm việc cung cấp các giá trị cho các đối số đầu vào và gọi mô-đun đó, sau đó so sánh các giá trị của các đối số đầu ra với kết quả dự đoán của thử nghiệm.

Ngược lại, một đối tượng “điển hình” có lẽ chứa 30 phương thức, nhiều phương thức trong số đó tương đối nhỏ, thường chỉ có hai hoặc ba câu lệnh thực thi được [Wilde, Matthews và Huitt, 1993]. Các phương thức này không trả lại giá trị cho người gọi mà thay đổi trạng thái của đối tượng. Nghĩa là, các phương thức này sửa đổi các thuộc tính (biến trạng thái) của đối tượng. Khó khăn ở đây là, để kiểm tra xem việc thay đổi trạng thái có được thực hiện đúng hay không, cần phải gửi các thông báo bổ sung cho đối tượng. Ví dụ, xem xét đối tượng tài khoản ngân hàng được mô tả trong Phần 1.9. Tác dụng của phương thức gửi tiền là tăng giá trị của biến trạng thái accountBalance. Tuy nhiên, do ẩn thông tin, cách duy nhất để kiểm tra xem một phương thức gửi tiền cụ thể đã được thực hiện chính xác hay chưa là gọi phương thức xác định Số dư cả trước và sau khi gọi phương thức gửi tiền và xem số dư ngân hàng thay đổi như thế nào.

Tình huống tồi tệ hơn nếu đối tượng không bao gồm các phương thức có thể được gọi để xác định giá trị của tất cả các biến trạng thái. Một cách khác là bao gồm các phương thức bổ sung cho mục đích này, sau đó sử dụng trình biên dịch có điều kiện để đảm bảo rằng chúng không khả dụng ngoại trừ mục đích thử nghiệm (trong C++, điều này có thể được thực hiện bằng cách sử dụng #ifdef). Kế hoạch kiểm tra (Phần 9.6) phải quy định rằng giá trị của mọi biến trạng thái có thể truy cập được trong quá trình kiểm tra. Để đáp ứng yêu cầu này, các phương thức bổ sung trả về giá trị của các biến trạng thái có thể phải được thêm vào các lớp liên quan trong quy trình thiết kế. Do đó, có thể kiểm tra hiệu quả của việc gọi một phương thức cụ thể của một đối tượng bằng cách truy vấn giá trị của biến trạng thái áp dụng.

Đáng ngạc nhiên là một phương pháp kế thừa vẫn có thể phải được thử nghiệm. Nghĩa là, ngay cả khi một phương thức đã được kiểm tra đầy đủ, nó có thể yêu cầu kiểm tra kỹ lưỡng khi được kế thừa, không thay đổi, bởi một lớp con. Để thấy điểm sau này, hãy xem xét hệ thống phân cấp lớp được hiển thị trong Hình 15.17. Hai phương thức được định nghĩa trong lớp cơ sở RootedTreeClass, cụ thể là displayNodeContents và printRoutine, trong đó phương thức displayNodeContents sử dụng phương thức printRoutine.

Tiếp theo xem xét phân lớp BinaryTreeClass. Lớp con này kế thừa phương thức printRoutine từ lớp cơ sở RootedTreeClass của nó. Ngoài ra, một phương thức mới, displayNodeContents, được định nghĩa sẽ ghi đè phương thức được định nghĩa trong RootedTreeClass. Phương thức mới này vẫn sử dụng printRoutine. Trong ký hiệu Java, BinaryTreeClass.displayNodeContents sử dụng RootedTreeClass.printRoutine.

Bây giờ hãy xem xét phân lớp BalancedBinaryTreeClass. Lớp con này kế thừa phương thức displayNodeContents từ lớp cha BinaryTreeClass của nó. Tuy nhiên, một phương thức mới printRoutine được định nghĩa sẽ ghi đè phương thức được định nghĩa trong RootedTreeClass. Khi displayNodeContents sử dụng printRoutine trong ngữ cảnh của Balanced- BinaryTreeClass, các quy tắc phạm vi của C++ và Java chỉ định rằng phiên bản cục bộ của printRoutine sẽ được sử dụng. Trong ký hiệu Java, khi phương thức BinaryTreeClass.display- NodeContents được gọi trong phạm vi từ vựng của BalancedBinaryTreeClass, nó sử dụng phương thức BalancedBinaryTreeClass.printRoutine.

Do đó, mã thực tế (phương thức printRoutine) được thực thi khi displayNodeContents được gọi trong phần khởi tạo của BinaryTreeClass khác với mã được thực thi khi displayNodeContents được gọi trong phần khởi tạo của BalancedBinaryTreeClass. Điều này cho dù chính phương thức displayNodeContents được kế thừa, không thay đổi, bởi BalancedBinaryTreeClass từ BinaryTreeClass. Do đó, ngay cả khi phương thức displayNodeContents đã được kiểm tra kỹ lưỡng trong đối tượng BinaryTreeClass, thì phương pháp này vẫn phải được kiểm tra lại từ đầu khi được sử dụng lại trong môi trường BalancedBinaryTreeClass. Để làm cho vấn đề trở nên phức tạp hơn, có những lý do lý thuyết giải thích tại sao nó cần được kiểm thử lại với các trường hợp kiểm thử khác nhau [Perry và Kaiser, 1990].

Cần phải chỉ ra ngay rằng những rắc rối này không phải là lý do để từ bỏ mô hình hướng đối tượng. Đầu tiên, chúng chỉ phát sinh thông qua sự tương tác của các phương thức (ví dụ như displayNodeContents và printRoutine). Thứ hai, có thể xác định khi nào cần kiểm tra lại [Harrold, McGregor, và Fitzpatrick, 1992].

Giả sử việc khởi tạo một lớp đã được kiểm tra kỹ lưỡng. Sau đó, bất kỳ phương thức mới hoặc được xác định lại nào của một lớp con cần phải được kiểm tra, cùng với các phương thức được gắn cờ để kiểm tra lại do tương tác của chúng với các phương thức khác. Nói tóm lại, tuyên bố rằng việc sử dụng mô hình hướng đối tượng làm giảm nhu cầu kiểm tra phần lớn là đúng.

Một số ý nghĩa quản lý của thử nghiệm đơn vị hiện được xem xét.

* 1. Management Aspects of Unit Testing 15.18 Các khía cạnh quản lý của kiểm thử đơn vị

An important decision that must be made during the development of every code artifact is how much time, and therefore money, to spend on testing that artifact. As with so many other economic issues in software engineering, cost–benefit analysis (Section 5.2) can play a useful role. For example, the decision as to whether the cost of correctness proving ex- ceeds the benefit of the assurance that a specific product satisfies its specifications can be decided on the basis of cost–benefit analysis. Cost–benefit analysis also can be used to compare the cost of running additional test cases against the cost of failure of the delivered product caused by inadequate testing.

There is another approach for determining whether testing of a specific code artifact should continue or whether it is likely that virtually all the faults have been removed. The techniques of reliability analysis can be used to provide statistical estimates of how many faults remain. A variety of different techniques have been proposed for determining statistical estimates of the number of remaining faults. The basic idea underlying these techniques is the following: Suppose a code artifact is tested for 1 week. On Monday, 23 faults are found and seven more are found on Tuesday. On Wednesday, five more faults are found, two on Thursday, and none on Friday. Because the rate of fault detection decreases steadily from 23 faults per day to none, it seems likely that most faults have been found, and testing of that code artifact could be halted. Determining the probability that there are no more faults in the code requires a level of mathematical statistics beyond that required for readers of this book. Details therefore are not given here; the reader interested in reliability analysis should consult Grady [1992].

Một quyết định quan trọng phải được đưa ra trong quá trình phát triển mọi mã tạo phẩm là bao nhiêu thời gian, và do đó là tiền bạc, để chi cho việc thử nghiệm tạo phẩm đó. Cũng như rất nhiều vấn đề kinh tế khác trong công nghệ phần mềm, phân tích chi phí-lợi ích (Phần 5.2) có thể đóng một vai trò hữu ích. Ví dụ, quyết định liệu chi phí chứng minh tính đúng đắn có vượt quá lợi ích của việc đảm bảo rằng một sản phẩm cụ thể đáp ứng các thông số kỹ thuật của nó hay không có thể được quyết định trên cơ sở phân tích chi phí-lợi ích. Phân tích lợi ích chi phí cũng có thể được sử dụng để so sánh chi phí chạy các trường hợp thử nghiệm bổ sung với chi phí thất bại của sản phẩm được giao do thử nghiệm không đầy đủ.

Có một cách tiếp cận khác để xác định liệu có nên tiếp tục kiểm tra một tạo phẩm mã cụ thể hay liệu có khả năng là hầu như tất cả các lỗi đã được loại bỏ hay chưa. Các kỹ thuật phân tích độ tin cậy có thể được sử dụng để đưa ra các ước tính thống kê về số lượng lỗi còn lại. Nhiều kỹ thuật khác nhau đã được đề xuất để xác định các ước tính thống kê về số lỗi còn lại. Ý tưởng cơ bản của các kỹ thuật này là như sau: Giả sử một tạo phẩm mã được thử nghiệm trong 1 tuần. Vào thứ Hai, 23 lỗi được tìm thấy và bảy lỗi nữa được tìm thấy vào thứ Ba. Vào thứ Tư, năm lỗi nữa được tìm thấy, hai lỗi vào thứ Năm và không có lỗi nào vào thứ Sáu. Bởi vì tỷ lệ phát hiện lỗi giảm đều đặn từ 23 lỗi mỗi ngày xuống không có lỗi nào, có vẻ như hầu hết các lỗi đã được tìm thấy và việc kiểm tra tạo phẩm mã đó có thể bị tạm dừng. Việc xác định xác suất không còn lỗi trong mã yêu cầu trình độ thống kê toán học cao hơn yêu cầu đối với người đọc cuốn sách này. Do đó, chi tiết không được đưa ra ở đây; người đọc quan tâm đến phân tích độ tin cậy nên tham khảo Grady [1992].

* 1. **When to Reimplement Rather**

**than Debug a Code Artifact** **15.19 Khi Nào Nên Thực Hiện Lại**

**hơn Gỡ lỗi mã tạo tác**

When a member of the SQA group detects a failure (erroneous output), as stated previously, the code artifact must be returned to the original programmer for **debugging**, that is, detec- tion of the fault and correction of the code. On some occasions, it is preferable for the code artifact to be thrown away and redesigned and recoded from scratch, either by the original programmer or by another, possibly more senior, member of the development team.

To see why this may be necessary, consider Figure 15.18. The graph shows the coun- terintuitive concept that the probability of the existence of more faults in a code artifact is proportional to the number of faults already found in that code artifact [Myers, 1979]. To see why this should be so, consider two code artifacts, a1 and a2. Suppose that both code artifacts are approximately the same length and both have been tested for the same number of hours. Suppose further that only 2 faults were detected in a1, but 48 faults were detected in a2. It is likely that more faults remain to be rooted out of a2 than out of a1. Furthermore, additional testing and debugging of a2 is likely to be a lengthy process, and the suspicion that a2 is still not perfect will remain. In both the short run and the long run, it is preferable to discard a2, redesign it, and then recode it.

Khi một thành viên của nhóm SQA phát hiện lỗi (đầu ra sai), như đã nêu trước đó, tạo phẩm mã phải được trả lại cho lập trình viên ban đầu để gỡ lỗi, nghĩa là phát hiện lỗi và sửa mã. Trong một số trường hợp, tốt hơn là loại bỏ tạo phẩm mã và thiết kế lại và mã hóa lại từ đầu, bởi lập trình viên ban đầu hoặc bởi một thành viên khác, có thể cao cấp hơn, của nhóm phát triển.

Để biết tại sao điều này có thể cần thiết, hãy xem Hình 15.18. Biểu đồ cho thấy khái niệm phản trực giác rằng xác suất tồn tại nhiều lỗi hơn trong một mã tạo tác tỷ lệ thuận với số lượng lỗi đã được tìm thấy trong mã tạo tác đó [Myers, 1979]. Để biết tại sao lại như vậy, hãy xem xét hai tạo phẩm mã, a1 và a2. Giả sử rằng cả hai tạo tác mã đều có độ dài xấp xỉ như nhau và cả hai đã được thử nghiệm trong cùng số giờ. Giả sử thêm rằng chỉ có 2 lỗi được phát hiện trong a1, nhưng 48 lỗi được phát hiện trong a2. Có khả năng là vẫn còn nhiều lỗi phát sinh từ a2 hơn là từ a1. Hơn nữa, việc kiểm tra và sửa lỗi bổ sung của a2 có thể là một quá trình dài và sự nghi ngờ rằng a2 vẫn chưa hoàn hảo sẽ vẫn còn. Trong cả ngắn hạn và dài hạn, tốt hơn là loại bỏ a2, thiết kế lại và sau đó mã hóa lại.

**FIGURE 15.18**

Graph showing that the probability that faults are still to be found is proportional to the number of faults already detected.

1

Probability of existence of additional faults

0

Number of faults already found

The distribution of faults in modules certainly is not uniform. Myers [1979] cites the example of faults found by users in OS/370. It was found that 47 percent of the faults were associated with only 4 percent of the modules. Current research shows that the nonuniform distribution of faults in modules has continued. For example, Andersson and Runeson [2007] examined three telecommunications products that were developed using the iterative-and- incremental model. For the first project, they found that 20 percent of the modules contained 63 percent of the faults; for the second and third projects, 20 percent of the modules contained 70 percent of the faults.

An earlier study by Endres [1975] regarding internal tests of DOS/VS (Release 28) at IBM Laboratories, Böblingen, Germany, showed similar nonuniformity. Of the total of 512 faults detected in 202 modules, only 1 fault was detected in each of 112 of the modules. On the other hand, some modules were found to have 14, 15, 19, and 28 faults, respectively. Endres points out that the latter three modules were three of the largest modules in the product, each comprising over 3000 lines of DOS macro assembler language. However, the module with 14 faults was a relatively small module previously known to be very unstable. This type of module is a prime candidate for being discarded and recoded.

The way for management to cope with this sort of situation is to predetermine the maximum number of faults permitted during development of a given code artifact; when that maximum is reached, the code artifact must be thrown away and then redesigned and recoded, preferably by an experienced software professional. This maximum varies from application domain to application domain and from code artifact to code artifact. After all, the maximum permitted number of faults detected in a code artifact that reads a record from a database and checks the validity of the part number should be far smaller than the number of faults in a complex code artifact from a tank weapons control system that must coordinate data from a variety of sensors and direct the aim of the main gun toward the intended target. One way to decide on the maximum fault figure for a specific code artifact is to examine fault data on similar code artifacts that have required corrective maintenance. But, whatever estimation technique is used, management must ensure that the code artifact is scrapped if that figure is exceeded (but see Just in Case You Wanted to Know Box 15.7).

Sự phân bổ lỗi trong các mô-đun chắc chắn không đồng đều. Myers [1979] trích dẫn ví dụ về lỗi do người dùng tìm thấy trong OS/370. Người ta thấy rằng 47 phần trăm lỗi chỉ liên quan đến 4 phần trăm mô-đun. Nghiên cứu hiện tại cho thấy rằng sự phân bố lỗi không đồng đều trong các mô-đun vẫn tiếp tục. Ví dụ, Andersson và Runeson [2007] đã kiểm tra ba sản phẩm viễn thông được phát triển bằng cách sử dụng mô hình lặp đi lặp lại và tăng dần. Đối với dự án đầu tiên, họ phát hiện ra rằng 20% ​​mô-đun chứa 63% lỗi; đối với dự án thứ hai và thứ ba, 20% số mô-đun chứa 70% lỗi.

Một nghiên cứu trước đó của Endres [1975] liên quan đến các thử nghiệm nội bộ của DOS/VS (Phiên bản 28) tại Phòng thí nghiệm IBM, Böblingen, Đức, cho thấy sự không đồng nhất tương tự. Trong tổng số 512 lỗi được phát hiện trong 202 mô-đun, chỉ có 1 lỗi được phát hiện trong mỗi 112 mô-đun. Mặt khác, một số mô-đun được phát hiện có lỗi lần lượt là 14, 15, 19 và 28. Endres chỉ ra rằng ba mô-đun sau là ba trong số các mô-đun lớn nhất trong sản phẩm, mỗi mô-đun bao gồm hơn 3000 dòng ngôn ngữ trình biên dịch chương trình macro DOS. Tuy nhiên, mô-đun có 14 lỗi là một mô-đun tương đối nhỏ trước đây được biết là rất không ổn định. Loại mô-đun này là ứng cử viên chính cho việc bị loại bỏ và mã hóa lại.

Cách để ban quản lý đối phó với loại tình huống này là xác định trước số lượng lỗi tối đa được phép trong quá trình phát triển một tạo phẩm mã nhất định; khi đạt đến mức tối đa đó, phần mềm tạo mã phải được loại bỏ, sau đó được thiết kế lại và mã hóa lại, tốt nhất là bởi một chuyên gia phần mềm có kinh nghiệm. Mức tối đa này thay đổi từ miền ứng dụng này sang miền ứng dụng khác và từ tạo phẩm mã này sang tạo tác mã khác. Xét cho cùng, số lỗi tối đa được phép phát hiện trong một mã tạo tác đọc bản ghi từ cơ sở dữ liệu và kiểm tra tính hợp lệ của số bộ phận phải nhỏ hơn nhiều so với số lỗi trong một tạo phẩm mã phức tạp từ hệ thống điều khiển vũ khí xe tăng. phải phối hợp dữ liệu từ nhiều loại cảm biến và hướng mục tiêu của súng chính về phía mục tiêu đã định. Một cách để quyết định con số lỗi tối đa cho một tạo phẩm mã cụ thể là kiểm tra dữ liệu lỗi trên các tạo phẩm mã tương tự đã yêu cầu bảo trì khắc phục. Tuy nhiên, bất kể kỹ thuật ước tính nào được sử dụng, ban quản lý phải đảm bảo rằng phần mềm tạo mã bị loại bỏ nếu vượt quá con số đó (nhưng hãy xem Đề phòng trường hợp bạn muốn biết Hộp 15.7).



**Just in Case You Wanted to Know**

Chapter 15 *Implementation* **535**

**Box 15.7**

The discussion regarding the maximum permitted number of faults detected during devel- opment of a code artifact means precisely that: the maximum number permitted *during de- velopment*. The maximum permitted number of faults detected after the product has been delivered to the client should be *zero* for all code artifacts of all products. That is, it should be the aim of every software engineer to deliver fault-free code to the client.

Cuộc thảo luận về số lượng lỗi tối đa được phép phát hiện trong quá trình phát triển tạo phẩm mã có nghĩa chính xác là: số lượng lỗi tối đa được phép trong quá trình phát triển. Số lượng lỗi tối đa được phép phát hiện sau khi sản phẩm đã được giao cho khách hàng phải bằng 0 đối với tất cả các tạo phẩm mã của tất cả các sản phẩm. Nghĩa là, mục tiêu của mọi kỹ sư phần mềm là cung cấp mã không có lỗi cho khách hàng.

Cuộc thảo luận về số lượng lỗi tối đa được phép phát hiện trong quá trình phát triển tạo phẩm mã có nghĩa chính xác là: số lượng lỗi tối đa được phép trong quá trình phát triển. Số lượng lỗi tối đa được phép phát hiện sau khi sản phẩm đã được giao cho khách hàng phải bằng 0 đối với tất cả các tạo phẩm mã của tất cả các sản phẩm. Nghĩa là, mục tiêu của mọi kỹ sư phần mềm là cung cấp mã không có lỗi cho khách hàng.

* 1. Integration Testing 15.20 Thử nghiệm tích hợp

Each new code artifact must be tested when it is added to what has already been integrated; this is termed **integration testing**. The key point here is first to test the new code artifact as described in Sections 15.10 through 15.14 (unit testing) and then to check that the rest of the partial product continues to behave as it did before the new code artifact was integrated into it.

When the product has a graphical user interface, special issues can arise with regard to integration testing. In general, testing a product usually can be simplified by storing the input data for a test case in a file. The product then is executed, and the relevant data submitted to it. With the aid of a CASE tool, the whole process can be automated; that is, a set of test cases is set up, together with the expected outcome of each case. The CASE tool runs each test case, compares the actual results with the expected results, and reports to the user on each case. The test cases then are stored for use in regression testing whenever the product is modified. SilkTest is an example of a tool of this kind.

However, when a product incorporates a graphical user interface, this approach does not work. Specifically, test data for pulling down a menu or clicking on a mouse button cannot be stored in a file in the same way as conventional test data. At the same time, it is time consum- ing and boring to test a GUI manually. The solution to this problem is to use a special CASE tool that keeps a record of mouse clicks, key presses, and so on. The GUI is tested once manu- ally so that the CASE tool can set up the test file. Thereafter, this file is used in subsequent tests. A number of CASE tools support testing GUIs, including QARun and XRunner.

When the integration process is complete, the product as a whole is tested; this is termed **product testing**. When the developers are confident about the correctness of every aspect of the product, it is handed over to the client for **acceptance testing**. These two forms of testing are now described in more detail.

Mỗi tạo tác mã mới phải được kiểm tra khi nó được thêm vào những gì đã được tích hợp; điều này được gọi là thử nghiệm tích hợp. Điểm mấu chốt ở đây trước tiên là kiểm tra tạo phẩm mã mới như được mô tả trong Phần 15.10 đến 15.14 (kiểm tra đơn vị) và sau đó kiểm tra xem phần còn lại của sản phẩm một phần có tiếp tục hoạt động như trước khi tạo phẩm mã mới được tích hợp vào không.

Khi sản phẩm có giao diện người dùng đồ họa, các vấn đề đặc biệt có thể phát sinh liên quan đến thử nghiệm tích hợp. Nói chung, việc kiểm thử một sản phẩm thường có thể được đơn giản hóa bằng cách lưu trữ dữ liệu đầu vào cho một trường hợp kiểm thử trong một tệp. Sau đó, sản phẩm được thực thi và dữ liệu liên quan được gửi tới sản phẩm. Với sự trợ giúp của công cụ CASE, toàn bộ quy trình có thể được tự động hóa; nghĩa là, một tập hợp các trường hợp thử nghiệm được thiết lập cùng với kết quả mong đợi của từng trường hợp. Công cụ CASE chạy từng trường hợp thử nghiệm, so sánh kết quả thực tế với kết quả mong đợi và báo cáo cho người dùng về từng trường hợp. Các trường hợp thử nghiệm sau đó được lưu trữ để sử dụng trong thử nghiệm hồi quy bất cứ khi nào sản phẩm được sửa đổi. SilkTest là một ví dụ về một công cụ thuộc loại này.

Tuy nhiên, khi một sản phẩm kết hợp giao diện người dùng đồ họa, phương pháp này không hoạt động. Cụ thể, dữ liệu kiểm tra để kéo xuống menu hoặc nhấp vào nút chuột không thể được lưu trữ trong một tệp giống như dữ liệu kiểm tra thông thường. Đồng thời, việc kiểm tra GUI theo cách thủ công sẽ tốn thời gian và nhàm chán. Giải pháp cho vấn đề này là sử dụng một công cụ CASE đặc biệt lưu giữ bản ghi các lần bấm chuột, bấm phím, v.v. GUI được kiểm tra thủ công một lần để công cụ CASE có thể thiết lập tệp kiểm tra. Sau đó, tệp này được sử dụng trong các thử nghiệm tiếp theo. Một số công cụ CASE hỗ trợ GUI kiểm tra, bao gồm QARun và XRunner.

Khi quá trình tích hợp hoàn tất, toàn bộ sản phẩm sẽ được kiểm tra; điều này được gọi là thử nghiệm sản phẩm. Khi các nhà phát triển tự tin về tính chính xác của mọi khía cạnh của sản phẩm, nó sẽ được bàn giao cho khách hàng để kiểm tra chấp nhận. Hai hình thức thử nghiệm này hiện được mô tả chi tiết hơn.

* 1. Product Testing 15.21 Thử nghiệm sản phẩm

The fact that the last code artifact has been integrated successfully into the product does not mean that the task of the developers is complete. The SQA group still must perform a number of testing tasks to ascertain that the product will be successful. There are two main types of software, commercial off-the-shelf (COTS) software (Section 1.11) and custom software. The aim of COTS product testing is to ensure that the product as a whole is free of faults. When the product testing is complete, the product undergoes alpha and beta testing, as described in Section 3.7. That is, preliminary versions are shipped to selected prospective buyers of the product to get feedback, particularly regarding residual faults overlooked by the SQA team.

Custom software, on the other hand, undergoes somewhat different product testing. The SQA group performs a number of testing tasks to be certain that the product will not fail its acceptance test, the final hurdle that the custom software development team must overcome.

The failure of a product to pass its acceptance test almost always is a poor reflection on the management capabilities of the development organization. The client may conclude that the developers are incompetent, which all but guarantees that the client will do everything to avoid employing those developers again. Worse, the client may believe that the developers are dishonest and deliberately handed over substandard software to finish the contract and be paid as quickly as possible. If the client genuinely believes this and tells other potential clients, then the developers face a major public relations problem. It is up to the SQA group to make sure the product passes the acceptance test with flying colors.

To ensure a successful acceptance test, the SQA group must test the product using tests that the SQA group believes closely approximate the forthcoming acceptance tests:

* Black-box test cases for the product as a whole must be run. Up to now, test cases have been set up on an artifact-by-artifact or class-by-class basis, ensuring that each code artifact or class individually satisfies its specifications.
* The robustness of the product as a whole must be tested. Again, the robustness of individ- ual code artifacts and classes was tested during integration; now productwide robustness is the issue for which test cases must be set up and run. In addition, the product must be subjected to **stress testing**, that is, making sure that it behaves correctly when operat- ing under a peak load, such as all terminals trying to log on at the same time or customers operating all the automated teller machines simultaneously. The product also must be sub- jected to **volume testing**, for example, making sure that it can handle large input files.
* The SQA group must check that the product satisfies all its constraints. For example, if the specifications state that the response time for 95 percent of queries when the product is working under full load must be under 3 seconds, then it is the responsibility of the SQA group to verify that this indeed is the case. There is no question that the client will check constraints during acceptance testing; and if the product fails to meet a major con- straint, then the development organization will lose a considerable amount of credibility. Similarly, storage constraints and security constraints must be checked.
* The SQA group must review all documentation to be handed over to the client together with the code. The SQA group must check that the documentation conforms to the stan- dards laid down in the SPMP. In addition, the documentation must be checked against the product. For instance, the SQA group has to determine that the user manual indeed reflects the correct way of using the product and that the product functions as specified in the user manual.

Once the SQA group assures management that the product can handle anything the acceptance testers can throw at it, the product (that is, the code plus all the documentation) is handed to the client organization for acceptance testing.

Thực tế là tạo tác mã cuối cùng đã được tích hợp thành công vào sản phẩm không có nghĩa là nhiệm vụ của các nhà phát triển đã hoàn thành. Nhóm SQA vẫn phải thực hiện một số nhiệm vụ thử nghiệm để chắc chắn rằng sản phẩm sẽ thành công. Có hai loại phần mềm chính, phần mềm thương mại có sẵn (COTS) (Phần 1.11) và phần mềm tùy chỉnh. Mục đích của thử nghiệm sản phẩm COTS là để đảm bảo rằng toàn bộ sản phẩm không có lỗi. Khi thử nghiệm sản phẩm hoàn tất, sản phẩm sẽ trải qua thử nghiệm alpha và beta, như được mô tả trong Phần 3.7. Nghĩa là, các phiên bản sơ bộ được chuyển đến những người mua tiềm năng đã chọn của sản phẩm để nhận phản hồi, đặc biệt là về các lỗi còn sót lại mà nhóm SQA đã bỏ qua.

Mặt khác, phần mềm tùy chỉnh trải qua thử nghiệm sản phẩm hơi khác. Nhóm SQA thực hiện một số nhiệm vụ thử nghiệm để chắc chắn rằng sản phẩm sẽ không vượt qua thử nghiệm chấp nhận, rào cản cuối cùng mà nhóm phát triển phần mềm tùy chỉnh phải vượt qua.

Việc một sản phẩm không vượt qua được bài kiểm tra chấp nhận hầu như luôn phản ánh kém về khả năng quản lý của tổ chức phát triển. Khách hàng có thể kết luận rằng các nhà phát triển không đủ năng lực, điều này đảm bảo rằng khách hàng sẽ làm mọi cách để tránh sử dụng lại những nhà phát triển đó. Tệ hơn nữa, khách hàng có thể tin rằng nhà phát triển không trung thực và cố tình bàn giao phần mềm kém chất lượng để hoàn thành hợp đồng và được thanh toán càng nhanh càng tốt. Nếu khách hàng thực sự tin vào điều này và nói với các khách hàng tiềm năng khác, thì các nhà phát triển phải đối mặt với một vấn đề lớn về quan hệ công chúng. Nhóm SQA phụ thuộc vào việc đảm bảo sản phẩm vượt qua bài kiểm tra nghiệm thu với màu sắc bay bổng.

Để đảm bảo kiểm tra chấp nhận thành công, nhóm SQA phải kiểm tra sản phẩm bằng các kiểm tra mà nhóm SQA tin rằng gần đúng với các kiểm tra chấp nhận sắp tới:

• Các ca kiểm thử hộp đen cho toàn bộ sản phẩm phải được chạy. Cho đến nay, các trường hợp thử nghiệm đã được thiết lập trên cơ sở từng thành phần tạo tác hoặc từng lớp, đảm bảo rằng mỗi tạo phẩm mã hoặc lớp riêng lẻ đáp ứng các thông số kỹ thuật của nó.

• Độ chắc chắn của toàn bộ sản phẩm phải được kiểm tra. Một lần nữa, tính mạnh mẽ của các tạo phẩm và lớp mã riêng lẻ đã được kiểm tra trong quá trình tích hợp; giờ đây, độ bền trên toàn sản phẩm là vấn đề mà các trường hợp thử nghiệm phải được thiết lập và chạy. Ngoài ra, sản phẩm phải được kiểm tra căng thẳng, nghĩa là đảm bảo rằng sản phẩm hoạt động chính xác khi hoạt động ở mức tải cao nhất, chẳng hạn như tất cả các thiết bị đầu cuối cố gắng đăng nhập cùng một lúc hoặc khách hàng vận hành tất cả các máy rút tiền tự động đồng thời. Ví dụ, sản phẩm cũng phải được kiểm tra khối lượng, đảm bảo rằng nó có thể xử lý các tệp đầu vào lớn.

• Nhóm SQA phải kiểm tra xem sản phẩm có thỏa mãn tất cả các ràng buộc của nó không. Ví dụ: nếu thông số kỹ thuật nêu rõ rằng thời gian phản hồi cho 95 phần trăm truy vấn khi sản phẩm hoạt động ở chế độ đầy tải phải dưới 3 giây, thì nhóm SQA có trách nhiệm xác minh rằng đây thực sự là trường hợp. Không có câu hỏi nào về việc khách hàng sẽ kiểm tra các ràng buộc trong quá trình kiểm tra chấp nhận; và nếu sản phẩm không đáp ứng được một hạn chế lớn, thì tổ chức phát triển sẽ mất đi một mức độ tín nhiệm đáng kể. Tương tự, các ràng buộc lưu trữ và ràng buộc bảo mật phải được kiểm tra.

• Nhóm SQA phải xem xét tất cả các tài liệu sẽ được bàn giao cho khách hàng cùng với mã. Nhóm SQA phải kiểm tra xem tài liệu có phù hợp với các tiêu chuẩn được đặt ra trong SPMP không. Ngoài ra, tài liệu phải được kiểm tra đối với sản phẩm. Chẳng hạn, nhóm SQA phải xác định rằng hướng dẫn sử dụng thực sự phản ánh đúng cách sử dụng sản phẩm và sản phẩm hoạt động như được chỉ định trong hướng dẫn sử dụng.

Sau khi nhóm SQA đảm bảo với ban quản lý rằng sản phẩm có thể xử lý mọi thứ mà người kiểm tra chấp nhận có thể ném vào nó, thì sản phẩm (tức là mã cộng với tất cả tài liệu) sẽ được chuyển cho tổ chức khách hàng để kiểm tra chấp nhận.

* 1. Acceptance Testing 15.22 Thử nghiệm nghiệm thu

The purpose of acceptance testing is for the client to determine whether the product in- deed satisfies its specifications as claimed by the developer. Acceptance testing is done by either the client organization, the SQA group in the presence of client representatives, or an independent SQA group hired by the client for this purpose. Acceptance testing natu- rally includes correctness testing, but in addition, it is necessary to test performance and

Chapter 15 *Implementation* **537**

robustness. The four major components of acceptance testing—testing correctness, robust- ness, performance, and documentation—are exactly what is done by the developer during product testing; this is not surprising, because product testing is a comprehensive rehearsal for the acceptance test.

A key aspect of acceptance testing is that it must be performed on actual data rather than on test data. No matter how well test cases are set up, by their very nature, they are artificial. More important, test data should be a true reflection of the corresponding actual data, but in practice, this is not always the case. For example, the member of the specifica- tion team responsible for characterizing the actual data may perform this task incorrectly. Alternatively, even if the data are specified correctly, the SQA group member who uses that data specification may misunderstand or misinterpret it. The resulting test cases are not a true reflection of the actual data, leading to an inadequately tested product. For these reasons, acceptance testing must be performed on actual data. Furthermore, because the development team endeavors to ensure that the product testing duplicates every aspect of the acceptance testing, as much of the product testing as possible should also be performed on actual data.

When a new product is to replace an existing one, the specification document almost always includes a clause to the effect that the new product must be installed to run in par- allel with the existing product. The reason is that there is a very real possibility that the new product may be faulty in some way. The existing product works correctly but is inad- equate in some respects. If the existing product is replaced by a new product that works incorrectly, then the client is in trouble. Therefore, both products must run in parallel until the client is satisfied that the new product can take over the functions of the existing prod- uct. Successful parallel running concludes acceptance testing, and the existing product can be retired.

When the product has passed its acceptance test, the task of the developers is complete.

Any changes now made to that product constitute postdelivery maintenance.

Mục đích của thử nghiệm chấp nhận là để khách hàng xác định xem sản phẩm có thực sự đáp ứng các thông số kỹ thuật của nó như yêu cầu của nhà phát triển hay không. Kiểm tra chấp nhận được thực hiện bởi tổ chức khách hàng, nhóm SQA với sự có mặt của đại diện khách hàng hoặc nhóm SQA độc lập do khách hàng thuê cho mục đích này. Kiểm tra chấp nhận đương nhiên bao gồm kiểm tra tính đúng đắn, nhưng ngoài ra, cần phải kiểm tra hiệu suất và

Chương 15 Thực hiện 537

sự cường tráng. Bốn thành phần chính của kiểm thử chấp nhận—kiểm thử tính chính xác, độ bền, hiệu suất và tài liệu—chính xác là những gì nhà phát triển thực hiện trong quá trình kiểm thử sản phẩm; điều này không có gì đáng ngạc nhiên, bởi vì thử nghiệm sản phẩm là một cuộc diễn tập toàn diện cho thử nghiệm chấp nhận.

Một khía cạnh quan trọng của thử nghiệm chấp nhận là nó phải được thực hiện trên dữ liệu thực chứ không phải trên dữ liệu thử nghiệm. Cho dù các trường hợp thử nghiệm được thiết lập tốt đến đâu, về bản chất, chúng đều là nhân tạo. Quan trọng hơn, dữ liệu thử nghiệm phải phản ánh trung thực dữ liệu thực tế tương ứng, nhưng trên thực tế, điều này không phải lúc nào cũng đúng. Ví dụ, thành viên của nhóm đặc tả chịu trách nhiệm mô tả dữ liệu thực tế có thể thực hiện nhiệm vụ này không chính xác. Ngoài ra, ngay cả khi dữ liệu được chỉ định chính xác, thành viên nhóm SQA sử dụng đặc tả dữ liệu đó có thể hiểu sai hoặc diễn giải sai. Các trường hợp thử nghiệm kết quả không phản ánh đúng dữ liệu thực tế, dẫn đến sản phẩm được thử nghiệm không đầy đủ. Vì những lý do này, thử nghiệm chấp nhận phải được thực hiện trên dữ liệu thực tế. Hơn nữa, vì nhóm phát triển nỗ lực để đảm bảo rằng thử nghiệm sản phẩm trùng lặp với mọi khía cạnh của thử nghiệm chấp nhận, nên càng nhiều thử nghiệm sản phẩm càng tốt cũng nên được thực hiện trên dữ liệu thực tế.

Khi một sản phẩm mới thay thế một sản phẩm hiện có, tài liệu đặc điểm kỹ thuật hầu như luôn bao gồm một điều khoản rằng sản phẩm mới phải được cài đặt để chạy song song với sản phẩm hiện có. Lý do là có khả năng rất thực tế là sản phẩm mới có thể bị lỗi theo một cách nào đó. Sản phẩm hiện tại hoạt động chính xác nhưng không phù hợp ở một số khía cạnh. Nếu sản phẩm hiện tại được thay thế bằng một sản phẩm mới hoạt động không chính xác, thì khách hàng sẽ gặp rắc rối. Do đó, cả hai sản phẩm phải chạy song song cho đến khi khách hàng hài lòng rằng sản phẩm mới có thể tiếp quản các chức năng của sản phẩm hiện có. Quá trình chạy song song thành công sẽ kết thúc thử nghiệm chấp nhận và sản phẩm hiện có có thể ngừng hoạt động.

Khi sản phẩm đã vượt qua bài kiểm tra chấp nhận, nhiệm vụ của các nhà phát triển đã hoàn thành.

Bất kỳ thay đổi nào hiện được thực hiện đối với sản phẩm đó đều cấu thành bảo trì sau giao hàng.

###### Case Study



* 1. The Test Workflow: The MSG Foundation Case Study

The C++ and Java implementations of the MSG Foundation product (available for download at [www.mhhe.com/Schach](http://www.mhhe.com/Schach)) were tested against the black-box test cases of Figure 15.13 and 15.14, as well as the glass-box test cases of Problems 15.35 through 15.39.

15.23 Quy trình thử nghiệm: Nghiên cứu tình huống của Quỹ MSG

Việc triển khai C++ và Java của sản phẩm MSG Foundation (có sẵn để tải xuống tại www.mhhe.com/Schach) đã được thử nghiệm dựa trên các trường hợp thử nghiệm hộp đen của Hình 15.13 và 15.14, cũng như các trường hợp thử nghiệm hộp thủy tinh của Bài toán 15.35 đến 15.39

* 1. **CASE Tools for Implementation 15.24 Công cụ TÌNH HUỐNG để Thực hiện**

CASE tools to support implementation of code artifacts were described in some detail in Chapter 5. For integration, version-control tools, build tools, and configuration manage- ment tools are needed (Chapter 5). The reason is that code artifacts under test change

continually as a consequence of faults being detected and corrected, and these CASE tools are essential to ensure that the appropriate version of each artifact is compiled and linked. Commercially available configuration-control workbenches include PVCS and SourceSafe. Popular open-source configuration-control tools include CVS and Subversion.

In each chapter so far, CASE tools and workbenches specific to that workflow have been described. Now that all workflows of the development process have been described, it is appropriate to consider CASE tools for the process as a whole.

Các công cụ CASE để hỗ trợ triển khai các tạo phẩm mã đã được mô tả chi tiết trong Chương 5. Để tích hợp, các công cụ kiểm soát phiên bản, công cụ xây dựng và công cụ quản lý cấu hình là cần thiết (Chương 5). Lý do là các tạo phẩm mã trong quá trình thử nghiệm thay đổi

liên tục do lỗi được phát hiện và sửa chữa, đồng thời các công cụ CASE này rất cần thiết để đảm bảo rằng phiên bản phù hợp của từng cấu phần phần mềm được biên dịch và liên kết. Bàn làm việc kiểm soát cấu hình có sẵn trên thị trường bao gồm PVCS và SourceSafe. Các công cụ kiểm soát cấu hình nguồn mở phổ biến bao gồm CVS và Subversion.

Trong mỗi chương cho đến nay, các công cụ CASE và bàn làm việc cụ thể cho quy trình làm việc đó đã được mô tả. Bây giờ tất cả các quy trình công việc của quy trình phát triển đã được mô tả, nên xem xét các công cụ CASE cho toàn bộ quy trình là phù hợp

* + 1. CASE Tools for the Complete Software Process 15.24.1 Công cụ CASE cho Quy trình Phần mềm Hoàn chỉnh

There is a natural progression within CASE. As described in Section 5.7, the simplest CASE device is a single **tool**, such as an online interface checker or a build tool. Next, tools can be combined, leading to a **workbench** that supports one or two activities within the software process, such as configuration control or coding. However, such a workbench might not pro- vide management information even for the limited portion of the software process to which it is applicable, let alone for the project as a whole. Finally, an **environment** provides computer-aided support for most, if not all of, the process.

Ideally, every software development organization should utilize an environment. But the cost of an environment can be large—not just the package itself but the hardware on which to run it. For a smaller organization, a workbench, or perhaps just a set of tools, may suffice. But, if at all possible, an **integrated environment** should be utilized to support the development and maintenance effort.

Có một sự tiến triển tự nhiên trong CASE. Như được mô tả trong Phần 5.7, thiết bị CASE đơn giản nhất là một công cụ duy nhất, chẳng hạn như trình kiểm tra giao diện trực tuyến hoặc công cụ xây dựng. Tiếp theo, các công cụ có thể được kết hợp, dẫn đến một bàn làm việc hỗ trợ một hoặc hai hoạt động trong quy trình phần mềm, chẳng hạn như kiểm soát cấu hình hoặc viết mã. Tuy nhiên, một bàn làm việc như vậy có thể không cung cấp thông tin quản lý ngay cả đối với phần giới hạn của quy trình phần mềm mà nó được áp dụng, chứ đừng nói đến toàn bộ dự án. Cuối cùng, một môi trường cung cấp hỗ trợ có sự trợ giúp của máy tính cho hầu hết, nếu không muốn nói là tất cả, quy trình.

Lý tưởng nhất là mọi tổ chức phát triển phần mềm nên sử dụng một môi trường. Nhưng chi phí của một môi trường có thể lớn—không chỉ bản thân gói mà cả phần cứng để chạy nó. Đối với một tổ chức nhỏ hơn, một bàn làm việc hoặc có lẽ chỉ một bộ công cụ là đủ. Tuy nhiên, nếu có thể, nên sử dụng một môi trường tích hợp để hỗ trợ nỗ lực phát triển và bảo trì.

* + 1. Integrated Development Environments 15.24.2 Môi trường phát triển tích hợp

The most common meaning of the word *integrated* within the CASE context is in terms of **user interface integration**. That is, all the tools in the environment share a common user interface. The idea behind this is that, if all the tools have the same visual appearance, the user of one tool should have little difficulty in learning and using another tool in the en- vironment. This has been successfully achieved on the Macintosh, where most applications have a similar “look and feel.” Although this is the usual meaning, there are other types of integration as well.

The term **tool integration** means that all the tools communicate via the same data format. For example, in the UNIX Programmer’s Workbench, the UNIX pipe formalism assumes that all data are in the form of an ASCII stream. It therefore is easy to combine two tools by directing the output stream from one tool to the input stream of the other tool. Eclipse is an open-source environment for tool integration.

**Process integration** refers to an environment that supports one specific software process. A subset of this class of environment is the **technique-based environment** (but see Just in Case You Wanted to Know Box 15.8). An environment of this type sup- ports only a specific technique for developing software, rather than a complete process. Environments exist for a variety of the techniques discussed in this book, such as Gane and Sarsen’s structured systems analysis (Section 12.3), Jackson system development (Section 14.5), and Petri nets (Section 12.8). The majority of these environments pro- vide graphical support for analysis and design and incorporate a data dictionary. Some consistency checking usually is provided. Support for managing the development pro- cess frequently is incorporated into the environment. Many environments of this type are

Chapter 15 *Implementation* **539**



**Just in Case You Wanted to Know**

**Box 15.8**

In the literature, technique-based environments usually are called **method-based environments**. The rise of the object-oriented paradigm gave the word *method* a second meaning (in the software engineering context). The original meaning was a technique or an approach; this is how the word is used in the phrase *method-based environment*. The object- oriented meaning is an operation within an object or class. Unfortunately, it sometimes is not totally clear from the context which meaning is intended.

Accordingly, I have used the word *method* exclusively within the context of the object-oriented paradigm. Otherwise, I have employed the term *technique* or *approach*. For example, that is why the term *formal method* never appears in Chapter 12. Instead, I use the term *formal technique*. Similarly, in this chapter, I have used the term *technique-based environments*.

Trong tài liệu, môi trường dựa trên kỹ thuật thường được gọi là môi trường dựa trên phương pháp. Sự gia tăng của mô hình hướng đối tượng đã mang lại cho từ phương thức một ý nghĩa thứ hai (trong bối cảnh công nghệ phần mềm). Ý nghĩa ban đầu là một kỹ thuật hoặc một cách tiếp cận; đây là cách từ này được sử dụng trong môi trường dựa trên phương pháp cụm từ. Ý nghĩa hướng đối tượng là một hoạt động bên trong một đối tượng hoặc lớp. Thật không may, đôi khi nó không hoàn toàn rõ ràng từ ngữ cảnh mà ý nghĩa được dự định.

Theo đó, tôi đã sử dụng từ phương pháp độc quyền trong ngữ cảnh của mô hình hướng đối tượng. Mặt khác, tôi đã sử dụng thuật ngữ kỹ thuật hoặc cách tiếp cận. Ví dụ, đó là lý do tại sao thuật ngữ phương pháp hình thức không bao giờ xuất hiện trong Chương 12. Thay vào đó, tôi sử dụng thuật ngữ kỹ thuật hình thức. Tương tự, trong chương này, tôi đã sử dụng thuật ngữ môi trường dựa trên kỹ thuật.

commercially available, including Analyst/Designer and Rhapsody. Analyst/Designer is specific to Yourdon’s methodology [Yourdon, 1989], and Rhapsody supports Statecharts [Harel et al., 1990]. With regard to object-oriented methodologies, IBM Rational Rose supports the Unified Process [Jacobson, Booch, and Rumbaugh, 1999]. In addition, some older environments have been extended to support the object-oriented paradigm; Software through Pictures is an example of this type. Almost all object-oriented environments now support UML.

The emphasis in most technique-based environments is on the support and formaliza- tion of the manual operations for software development laid down by the technique. That is, these environments force users to utilize the technique step by step in the way intended by its author, while assisting the user by providing graphical tools, a data dictionary, and consistency checking. This computerized framework is a strength of technique-based envi- ronments in that users are forced to use a specific technique and use it correctly. But it can be a weakness as well. Unless the software process of the organization incorporates this specific technique, use of a technique-based environment can be counterproductive.

Ý nghĩa phổ biến nhất của từ tích hợp trong ngữ cảnh CASE là về mặt tích hợp giao diện người dùng. Nghĩa là, tất cả các công cụ trong môi trường đều có chung một giao diện người dùng. Ý tưởng đằng sau điều này là, nếu tất cả các công cụ đều có hình thức trực quan giống nhau, thì người dùng một công cụ sẽ gặp ít khó khăn trong việc học và sử dụng một công cụ khác trong môi trường. Điều này đã đạt được thành công trên Macintosh, nơi hầu hết các ứng dụng đều có “giao diện” tương tự. Mặc dù đây là ý nghĩa thông thường, nhưng cũng có những kiểu tích hợp khác.

Thuật ngữ tích hợp công cụ có nghĩa là tất cả các công cụ giao tiếp thông qua cùng một định dạng dữ liệu. Ví dụ, trong Bàn làm việc của Lập trình viên UNIX, chủ nghĩa hình thức ống UNIX giả định rằng tất cả dữ liệu đều ở dạng luồng ASCII. Do đó, có thể dễ dàng kết hợp hai công cụ bằng cách hướng luồng đầu ra từ một công cụ này sang luồng đầu vào của công cụ kia. Eclipse là một môi trường mã nguồn mở để tích hợp công cụ.

Tích hợp quy trình đề cập đến một môi trường hỗ trợ một quy trình phần mềm cụ thể. Một tập hợp con của lớp môi trường này là môi trường dựa trên kỹ thuật (nhưng xem Đề phòng trường hợp bạn muốn biết Hộp 15.8). Một môi trường kiểu này chỉ hỗ trợ một kỹ thuật cụ thể để phát triển phần mềm, hơn là một quy trình hoàn chỉnh. Môi trường tồn tại cho nhiều kỹ thuật được thảo luận trong cuốn sách này, chẳng hạn như phân tích hệ thống có cấu trúc của Gane và Sarsen (Phần 12.3), phát triển hệ thống Jackson (Phần 14.5) và mạng Petri (Phần 12.8). Phần lớn các môi trường này cung cấp hỗ trợ đồ họa cho phân tích và thiết kế và kết hợp từ điển dữ liệu. Một số kiểm tra tính nhất quán thường được cung cấp. Hỗ trợ cho việc quản lý quá trình phát triển thường xuyên được tích hợp vào môi trường. Nhiều môi trường thuộc loại này có sẵn trên thị trường, bao gồm Nhà phân tích/Nhà thiết kế và Rhapsody. Nhà phân tích/Nhà thiết kế dành riêng cho phương pháp của Yourdon [Yourdon, 1989], và Rhapsody hỗ trợ Statecharts [Harel et al., 1990]. Đối với các phương pháp hướng đối tượng, IBM Rational Rose hỗ trợ Quy trình hợp nhất [Jacobson, Booch và Rumbaugh, 1999]. Ngoài ra, một số môi trường cũ hơn đã được mở rộng để hỗ trợ mô hình hướng đối tượng; Phần mềm thông qua Hình ảnh là một ví dụ về loại này. Hầu như tất cả các môi trường hướng đối tượng hiện nay đều hỗ trợ UML.

Trọng tâm trong hầu hết các môi trường dựa trên kỹ thuật là hỗ trợ và chính thức hóa các thao tác thủ công để phát triển phần mềm do kỹ thuật đặt ra. Nghĩa là, những môi trường này buộc người dùng phải sử dụng kỹ thuật từng bước theo cách mà tác giả của nó dự định, đồng thời hỗ trợ người dùng bằng cách cung cấp các công cụ đồ họa, từ điển dữ liệu và kiểm tra tính nhất quán. Khung máy tính hóa này là một thế mạnh của môi trường dựa trên kỹ thuật trong đó người dùng buộc phải sử dụng một kỹ thuật cụ thể và sử dụng nó một cách chính xác. Nhưng nó cũng có thể là một điểm yếu. Trừ khi quy trình phần mềm của tổ chức kết hợp kỹ thuật cụ thể này, việc sử dụng môi trường dựa trên kỹ thuật có thể phản tác dụng

* + 1. Environments for Business Applications 15.24.3 Môi trường cho Ứng dụng Kinh doanh

An important class of environments is used for building business-oriented products. The emphasis is on ease of use, achieved in a number of ways. In particular, the environment incorporates a number of standard screens, and these can be modified endlessly via a user- friendly GUI generator. One popular feature of such environments is a code generator. The lowest level of abstraction of a product then is the detailed design. The detailed design is the input to a code generator that automatically generates code in a language such as C, C++, or Java. This automatically generated code is compiled; no “programming” of any kind is performed on it.

Languages for specifying the detailed design could well be the programming languages of the future. The level of abstraction of programming languages rose from the physical machine level of first- and second-generation languages to the abstract machine level of third- and fourth-generation languages. Today, the level of abstraction of environments of this type is the detailed design level, a portable level. Section 15.2 stated that one objec- tive in using a fourth-generation language is shorter code, and hence quicker development and easier postdelivery maintenance. The use of code generators takes these goals even further, in that the programmer has to provide fewer details to a code generator than to an

interpreter or compiler for a 4GL. Therefore, it is expected that use of business-oriented environments that support code generators will increase productivity.

A number of environments of this type are currently available, including Oracle Developer Suite. Bearing in mind the size of the market for business-oriented CASE environments, it is likely that many more environments of this type will be developed in future years.

Một lớp môi trường quan trọng được sử dụng để xây dựng các sản phẩm định hướng kinh doanh. Trọng tâm là tính dễ sử dụng, đạt được theo một số cách. Đặc biệt, môi trường kết hợp một số màn hình tiêu chuẩn và những màn hình này có thể được sửa đổi vô tận thông qua trình tạo GUI thân thiện với người dùng. Một tính năng phổ biến của các môi trường như vậy là trình tạo mã. Mức độ trừu tượng thấp nhất của một sản phẩm sau đó là thiết kế chi tiết. Thiết kế chi tiết là đầu vào cho trình tạo mã tự động tạo mã bằng ngôn ngữ như C, C++ hoặc Java. Mã được tạo tự động này được biên dịch; không có "lập trình" nào được thực hiện trên đó.

Các ngôn ngữ để chỉ định thiết kế chi tiết có thể là ngôn ngữ lập trình của tương lai. Mức độ trừu tượng của ngôn ngữ lập trình tăng từ cấp độ máy vật lý của ngôn ngữ thế hệ thứ nhất và thứ hai lên cấp độ máy trừu tượng của ngôn ngữ thế hệ thứ ba và thứ tư. Ngày nay, mức độ trừu tượng của các môi trường thuộc loại này là mức thiết kế chi tiết, mức di động. Mục 15.2 nói rằng một mục tiêu trong việc sử dụng ngôn ngữ thế hệ thứ tư là mã ngắn hơn, và do đó phát triển nhanh hơn và bảo trì sau khi giao hàng dễ dàng hơn. Việc sử dụng các trình tạo mã thậm chí còn đạt được những mục tiêu này hơn nữa, trong đó lập trình viên phải cung cấp ít chi tiết hơn cho trình tạo mã so với một

trình thông dịch hoặc trình biên dịch cho 4GL. Do đó, người ta hy vọng rằng việc sử dụng các môi trường định hướng kinh doanh hỗ trợ các trình tạo mã sẽ tăng năng suất.

Một số môi trường thuộc loại này hiện có sẵn, bao gồm Oracle Developer Suite. Lưu ý đến quy mô của thị trường cho các môi trường CASE định hướng kinh doanh, có khả năng nhiều môi trường kiểu này sẽ được phát triển trong những năm tới.

* + 1. Public Tool Infrastructures 15.24.4 Cơ sở hạ tầng công cụ công cộng

The European Strategic Programme for Research in Information Technology (ESPRIT) developed an infrastructure for supporting CASE tools. Despite its name, the **portable common tool environment (PCTE)** [Long and Morris, 1993] is *not* an environment. Instead, it is an infrastructure that provides the services needed by CASE tools, in much the same way that UNIX provides the operating system services needed by user products. (The word *common* in PCTE is in the sense of “public” or “not copyrighted.”)

PCTE has gained widespread acceptance. For example, PCTE and the C and Ada in- terfaces to PCTE were adopted as ISO/IEC Standard 13719 in 1995. Implementations of PCTE include those of Emeraude and IBM.

The hope is that, in the future, many more CASE tools will conform to the PCTE stan- dard and that PCTE itself will be implemented on a wider variety of computers. A tool that conforms to PCTE would run on any computer that supports PCTE. Accordingly, this should result in the widespread availability of a broad range of CASE tools. This, in turn, should lead to better software processes and better-quality software.

Chương trình Chiến lược Châu Âu về Nghiên cứu Công nghệ Thông tin (ESPRIT) đã phát triển cơ sở hạ tầng để hỗ trợ các công cụ CASE. Bất chấp cái tên của nó, môi trường công cụ chung di động (PCTE) [Long và Morris, 1993] không phải là một môi trường. Thay vào đó, nó là một cơ sở hạ tầng cung cấp các dịch vụ mà các công cụ CASE cần, giống như cách mà UNIX cung cấp các dịch vụ hệ điều hành mà các sản phẩm của người dùng cần. (Từ phổ biến trong PCTE có nghĩa là “công khai” hoặc “không có bản quyền.”)

PCTE đã được chấp nhận rộng rãi. Ví dụ, PCTE và các giao diện C và Ada cho PCTE đã được thông qua thành Tiêu chuẩn ISO/IEC 13719 vào năm 1995. Việc triển khai PCTE bao gồm các triển khai của Emeraude và IBM.

Hy vọng rằng, trong tương lai, nhiều công cụ CASE hơn sẽ phù hợp với tiêu chuẩn PCTE và bản thân PCTE sẽ được triển khai trên nhiều loại máy tính hơn. Một công cụ phù hợp với PCTE sẽ chạy trên bất kỳ máy tính nào hỗ trợ PCTE. Theo đó, điều này sẽ dẫn đến sự sẵn có rộng rãi của một loạt các công cụ CASE. Đổi lại, điều này sẽ dẫn đến các quy trình phần mềm tốt hơn và phần mềm có chất lượng tốt hơn.

* + 1. Potential Problems with Environments 15.24.5 Các vấn đề tiềm ẩn với môi trường

No one environment is ideal for all products and all organizations, any more than one programming language can be considered “the best.” Every environment has its strengths and its weaknesses, and choosing an inappropriate environment can be worse than using no environment at all. For example, as explained in Section 15.24.2, a technique-based envi- ronment essentially automates a manual process. If an organization chooses to use an envi- ronment that enforces a technique inappropriate for it as a whole or for a current software product under development, then use of that CASE environment is counterproductive.

A worse situation occurs when an organization chooses to ignore the advice of Section 5.12, that the use of a CASE environment should be firmly avoided until the organization has attained CMM level 3. Of course, every organization should use CASE tools, and there generally is little harm in using a workbench. However, an environment imposes an automated software process on an organization that uses it. If a good process is being used, that is, the organization is at level 3 or higher, then use of the environment assists in all aspects of software production by automating that process. But, if the organi- zation is at the crisis-driven level 1 or even at level 2, then no process as such is in place. Automation of this nonexistent process, that is, the introduction of a CASE environment (as opposed to a CASE tool or CASE workbench), can lead only to chaos.

Không có một môi trường nào là lý tưởng cho mọi sản phẩm và mọi tổ chức, bất kỳ ngôn ngữ lập trình nào cũng có thể được coi là “tốt nhất”. Môi trường nào cũng có điểm mạnh và điểm yếu của nó, và việc chọn một môi trường không phù hợp có thể còn tệ hơn là không sử dụng môi trường nào cả. Ví dụ, như đã giải thích trong Phần 15.24.2, một môi trường dựa trên kỹ thuật về cơ bản sẽ tự động hóa một quy trình thủ công. Nếu một tổ chức chọn sử dụng một môi trường thực thi một kỹ thuật không phù hợp với nó nói chung hoặc cho một sản phẩm phần mềm hiện tại đang được phát triển, thì việc sử dụng môi trường CASE đó là phản tác dụng.

Tình huống tồi tệ hơn xảy ra khi một tổ chức chọn bỏ qua lời khuyên trong Mục 5.12, rằng nên kiên quyết tránh sử dụng môi trường CASE cho đến khi tổ chức đạt được CMM cấp 3. Tất nhiên, mọi tổ chức nên sử dụng các công cụ CASE và thường có ít tác hại trong việc sử dụng bàn làm việc. Tuy nhiên, một môi trường áp đặt một quy trình phần mềm tự động lên một tổ chức sử dụng nó. Nếu một quy trình tốt đang được sử dụng, nghĩa là tổ chức ở cấp độ 3 hoặc cao hơn, thì việc sử dụng môi trường sẽ hỗ trợ tất cả các khía cạnh của quá trình sản xuất phần mềm bằng cách tự động hóa quy trình đó. Tuy nhiên, nếu tổ chức ở cấp độ 1 hoặc thậm chí cấp độ 2 do khủng hoảng dẫn dắt, thì không có quy trình nào như vậy được áp dụng. Tự động hóa quy trình không tồn tại này, nghĩa là giới thiệu môi trường CASE (trái ngược với công cụ CASE hoặc bàn làm việc CASE), chỉ có thể dẫn đến sự hỗn loạn.

* 1. CASE Tools for the Test Workflow 15.25 Công cụ CASE cho Quy trình công việc kiểm tra

Numerous CASE tools are available to support the different types of testing that are performed during the implementation workflow. First consider unit testing. The XUnit testing frame- works, including JUnit for Java and CppUnit for C++, are a set of open-source automated

tools for unit testing; that is, they are utilized to test each class in turn. A set of test cases is prepared, and the tool checks that each of the messages sent to the class results in the expected answer being returned. Commercial tools of this type are produced by many ven- dors, including Parasoft.

We now turn to integration testing. Examples of commercial tools that support auto- mated integration testing (as well as unit testing) include SilkTest and IBM Rational Func- tional Tester. It is common for tools of this kind to pool the unit-testing test cases and utilize the resulting set of test cases for integration testing and regression testing.

During the test workflow, it is essential for management to know the status of all defects. In particular, it is vital to know which defects have been detected but have not yet been cor- rected. The best-known defect-tracking tool is Bugzilla, an open-source product.

Returning to Figure 1.6 yet again, it is vital to detect coding faults as soon as possible. One way to achieve this is to use a CASE tool to analyze the code, looking for common syntactic and semantic faults, or constructs that could lead to problems later. Examples of such tools include *lint* (for C—see Section 8.11.4), IBM Rational Purify, Sun’s Jackpot Source Code Metrics, and three Microsoft tools: PREfix, PREfast, and SLAM.

The Hyades project (otherwise known as the Eclipse test and performance tools proj- ect) is an open-source integrated test, trace, and monitoring environment that currently can be used with Java and C++. It has facilities for a variety of different testing tools. As more and more tool vendors adapt their tools to work under Eclipse, users will be able to select from a wider choice of testing tools, all of which will work in conjunction with one another.

Nhiều công cụ CASE có sẵn để hỗ trợ các loại thử nghiệm khác nhau được thực hiện trong quy trình triển khai. Đầu tiên hãy xem xét thử nghiệm đơn vị. Khung thử nghiệm XUnit, bao gồm JUnit cho Java và CppUnit cho C++, là một tập hợp mã nguồn mở tự động hóa.

công cụ để kiểm tra đơn vị; nghĩa là, chúng được sử dụng để kiểm tra lần lượt từng lớp. Một tập hợp các trường hợp kiểm tra đã được chuẩn bị và công cụ này sẽ kiểm tra xem mỗi thông báo được gửi đến lớp có trả về câu trả lời mong đợi hay không. Các công cụ thương mại thuộc loại này được sản xuất bởi nhiều nhà cung cấp, bao gồm cả Parasoft.

Bây giờ chúng ta chuyển sang thử nghiệm tích hợp. Ví dụ về các công cụ thương mại hỗ trợ kiểm thử tích hợp tự động (cũng như kiểm thử đơn vị) bao gồm SilkTest và IBM Rational Function Tester. Các công cụ loại này thường tập hợp các trường hợp thử nghiệm kiểm tra đơn vị và sử dụng tập hợp các trường hợp thử nghiệm kết quả để thử nghiệm tích hợp và thử nghiệm hồi quy.

Trong quy trình kiểm thử, quản lý cần biết trạng thái của tất cả các lỗi. Đặc biệt, điều quan trọng là phải biết những khiếm khuyết nào đã được phát hiện nhưng vẫn chưa được sửa chữa. Công cụ theo dõi lỗi nổi tiếng nhất là Bugzilla, một sản phẩm mã nguồn mở.

Quay trở lại Hình 1.6 một lần nữa, điều quan trọng là phát hiện các lỗi mã hóa càng sớm càng tốt. Một cách để đạt được điều này là sử dụng công cụ CASE để phân tích mã, tìm kiếm các lỗi ngữ nghĩa và cú pháp phổ biến hoặc các cấu trúc có thể dẫn đến sự cố sau này. Ví dụ về các công cụ như vậy bao gồm lint (đối với C—xem Phần 8.11.4), IBM Rational Purify, Sun’s Jackpot Source Code Metrics và ba công cụ của Microsoft: PREfix, PREfast và SLAM.

Dự án Hyades (còn được gọi là dự án các công cụ hiệu suất và kiểm tra Eclipse) là một môi trường kiểm tra, theo dõi và giám sát tích hợp mã nguồn mở hiện có thể được sử dụng với Java và C++. Nó có cơ sở vật chất cho nhiều công cụ kiểm tra khác nhau. Khi ngày càng có nhiều nhà cung cấp công cụ điều chỉnh các công cụ của họ để hoạt động trong Eclipse, người dùng sẽ có thể chọn từ nhiều lựa chọn công cụ kiểm tra hơn, tất cả các công cụ này sẽ hoạt động cùng với nhau.

* 1. Metrics for the Implementation Workflow 15.26 Số liệu cho quy trình thực hiện

A number of different complexity metrics for the implementation workflow are discussed in Section 15.13.2, including lines of code and McCabe’s cyclomatic complexity.

From a testing viewpoint, the relevant metrics include the total number of test cases and the number of test cases that resulted in a failure. The usual fault statistics must be main- tained for code inspections. The total number of faults is important, because if the number of faults detected in a code artifact exceeds a predetermined maximum, then that code artifact must be redesigned and recoded, as discussed in Section 15.19. In addition, detailed statistics need to be kept regarding the types of faults detected. Typical fault types include misunderstanding the design, lack of initialization, and inconsistent use of variables. The fault data can be incorporated into the checklists to be used during code inspections of future products.

A number of metrics specific to the object-oriented paradigm have been put forward, for example, the height of the inheritance tree [Chidamber and Kemerer, 1994]. Many of these metrics have been questioned on both theoretical and experimental grounds [Binkley and Schach, 1996; 1997]. Furthermore, Alshayeb and Li [2003] have shown that, whereas object-oriented metrics can relatively accurately predict the number of lines of code added, changed, and deleted in agile processes, they are of little use in predicting the same mea- sures in a framework–based process (see Section 8.5.2). It remains to be shown that there is a need for specifically object-oriented metrics, as opposed to classical metrics that can be applied equally to object-oriented software.

Một số thước đo độ phức tạp khác nhau cho quy trình triển khai được thảo luận trong Phần 15.13.2, bao gồm các dòng mã và độ phức tạp chu trình của McCabe.

Từ quan điểm thử nghiệm, các số liệu liên quan bao gồm tổng số trường hợp thử nghiệm và số trường hợp thử nghiệm dẫn đến lỗi. Các số liệu thống kê lỗi thông thường phải được duy trì để kiểm tra mã. Tổng số lỗi rất quan trọng, bởi vì nếu số lượng lỗi được phát hiện trong một mã tạo tác vượt quá mức tối đa được xác định trước, thì tạo tác mã đó phải được thiết kế lại và mã hóa lại, như đã thảo luận trong Phần 15.19. Ngoài ra, cần lưu giữ số liệu thống kê chi tiết về các loại lỗi được phát hiện. Các loại lỗi điển hình bao gồm hiểu sai thiết kế, thiếu khởi tạo và sử dụng các biến không nhất quán. Dữ liệu lỗi có thể được đưa vào danh sách kiểm tra để sử dụng trong quá trình kiểm tra mã sản phẩm trong tương lai.

Một số thước đo cụ thể cho mô hình hướng đối tượng đã được đưa ra, ví dụ, chiều cao của cây thừa kế [Chidamber và Kemerer, 1994]. Nhiều số liệu trong số này đã bị đặt câu hỏi trên cả cơ sở lý thuyết và thực nghiệm [Binkley và Schach, 1996; 1997]. Hơn nữa, Alshayeb và Li [2003] đã chỉ ra rằng, trong khi các phép đo hướng đối tượng có thể dự đoán tương đối chính xác số dòng mã được thêm, thay đổi và xóa trong các quy trình nhanh, chúng ít được sử dụng trong việc dự đoán các phép đo tương tự trong một quy trình dựa trên khuôn khổ (xem Phần 8.5.2). Vẫn còn phải chứng minh rằng cần có các phép đo hướng đối tượng cụ thể, trái ngược với các phép đo cổ điển có thể được áp dụng như nhau cho phần mềm hướng đối tượng.

* 1. Challenges of the Implementation Workflow 15.27 Những thách thức của Quy trình thực hiện

Paradoxically, a major challenge of the implementation workflow has to be met in the workflows that precede it. As explained in Chapter 8, code reuse is an effective way of reducing software development cost and delivery time. However, it is hard to achieve code reuse if it is attempted as late as the implementation workflow.

For example, suppose the decision is made to implement a product in language L. Now, after half the code artifacts have been implemented and tested, management decides to utilize package P for the graphical user interfaces of the software product. No matter how powerful the routines of P may be, if they are implemented in a language that is hard to interface with L, then they cannot be reused in the software product.

Even if language interoperability is not an issue, there is little point in trying to reuse an existing code artifact unless the item to be reused fits the design exactly. More work may be needed to modify the existing code artifact than to create a new code artifact from scratch.

Code reuse therefore has to be built into a software product from the very beginning. Reuse has to be a user requirement as well as a constraint of the specification document. The software project management plan (Section 9.4) must incorporate reuse. Also, the design document must state which code artifacts are to be implemented and which are to be reused.

So, as stated at the beginning of this section, even though code reuse is an important challenge of implementation, code reuse has to be incorporated into the requirements, anal- ysis, and design workflows.

From a purely technical viewpoint, the implementation workflow is relatively straight- forward. If the requirements, analysis, and design workflows were carried out satisfactorily, the task of implementation should pose few problems to competent programmers. However, management of integration is of critical importance; the challenges of the implementation workflow are to be found in this area.

Typical make-or-break issues include use of the appropriate CASE tools (Section 15.24), test planning once the specifications have been signed off on by the client (Sec- tion 9.6), ensuring that changes to the design are communicated to all relevant personnel (Section 15.6.5), and deciding when to stop testing and deliver the product to the client (Section 6.1.2).

Nghịch lý thay, một thách thức lớn của quy trình triển khai phải được đáp ứng trong các quy trình công việc trước nó. Như đã giải thích trong Chương 8, sử dụng lại mã là một cách hiệu quả để giảm chi phí phát triển phần mềm và thời gian chuyển giao. Tuy nhiên, thật khó để tái sử dụng mã nếu nó được cố gắng thực hiện muộn nhất trong quy trình triển khai.

Ví dụ: giả sử quyết định được đưa ra để triển khai sản phẩm bằng ngôn ngữ L. Bây giờ, sau khi một nửa mã tạo phẩm đã được triển khai và kiểm tra, ban quản lý quyết định sử dụng gói P cho giao diện người dùng đồ họa của sản phẩm phần mềm. Cho dù các thói quen của P có thể mạnh đến mức nào, nếu chúng được triển khai bằng ngôn ngữ khó giao tiếp với L, thì chúng không thể được sử dụng lại trong sản phẩm phần mềm.

Ngay cả khi khả năng tương tác ngôn ngữ không phải là vấn đề, thì việc cố gắng sử dụng lại một tạo phẩm mã hiện có cũng chẳng ích gì trừ khi mục được sử dụng lại phù hợp chính xác với thiết kế. Có thể cần nhiều công việc hơn để sửa đổi tạo phẩm mã hiện có hơn là tạo một tạo phẩm mã mới từ đầu.

Do đó, việc sử dụng lại mã phải được tích hợp vào một sản phẩm phần mềm ngay từ đầu. Tái sử dụng phải là một yêu cầu của người dùng cũng như một ràng buộc của tài liệu đặc tả. Kế hoạch quản lý dự án phần mềm (Phần 9.4) phải kết hợp việc sử dụng lại. Ngoài ra, tài liệu thiết kế phải nêu rõ mã tạo phẩm nào sẽ được triển khai và mã nào sẽ được sử dụng lại.

Vì vậy, như đã nêu ở phần đầu của phần này, mặc dù việc sử dụng lại mã là một thách thức quan trọng trong quá trình triển khai, nhưng việc sử dụng lại mã phải được đưa vào quy trình công việc yêu cầu, phân tích và thiết kế.

Từ quan điểm kỹ thuật thuần túy, quy trình triển khai tương đối đơn giản. Nếu các quy trình yêu cầu, phân tích và thiết kế được thực hiện thỏa đáng, thì nhiệm vụ thực hiện sẽ đặt ra một số vấn đề cho các lập trình viên có năng lực. Tuy nhiên, quản lý tích hợp có tầm quan trọng đặc biệt; những thách thức của quy trình thực hiện sẽ được tìm thấy trong lĩnh vực này.

Các vấn đề quyết định điển hình bao gồm việc sử dụng các công cụ CASE thích hợp (Phần 15.24), lập kế hoạch kiểm tra sau khi các thông số kỹ thuật đã được khách hàng phê duyệt (Phần 9.6), đảm bảo rằng các thay đổi đối với thiết kế được truyền đạt tới tất cả những người có liên quan nhân sự (Phần 15.6.5) và quyết định khi nào ngừng thử nghiệm và giao sản phẩm cho khách hàng (Phần 6.1.2).

**Chapter Review**

This chapter presents various issues relating to the implementation of a product by a team. These include choice of programming language (Section 15.1). The issue of fourth-generation languages is discussed in some detail in Section 15.2. Good programming practice is described in Section 15.3, and the need for practical coding standards is presented in Section 15.4. Then, comments are made regarding code reuse (Section 15.5). Implementation and integration activities must be carried out in parallel (Section 15.6). Top-down, bottom-up, and sandwich integration are described and compared (Sections 15.6.1 through 15.6.3). Integration of object-oriented products is discussed in Section 15.6.4, and management of integration in Section 15.6.5. The implementation workflow is presented in Section 15.7 and applied to the MSG Foundation case study in Section 15.8. Next, implementation aspects of the test workflow are presented (Section 15.9). Test cases must be selected systematically (Section 15.10). Various black- box, glass-box, and non-execution-based unit-testing techniques are described (Sections 15.11, 15.13, and 15.14, respectively) and then compared (Section 15.15). Black-box testing of the MSG Foundation

**FIGURE 15.19**

|  |  |
| --- | --- |
| Implementation workflow | Section 15.8, Appendix H, Appendix I |
| Black-box test cases | Section 15.12 |
| Test workflow | Section 15.23 |

Overview of the MSG Foundation

case study for Chapter 15.

case study is presented in Section 15.12. The Cleanroom technique is described in Section 15.16. Test- ing objects is discussed in Section 15.17, followed by a discussion of the managerial implications of unit testing (Section 15.18). Another problem is when to reimplement rather than debug a code artifact (Section 15.19). Integration testing is described in Section 15.20, product testing in Section 15.21, and acceptance testing in Section 15.22. The test workflow for the MSG Foundation case study is outlined in Section 15.23. CASE tools for the implementation workflow are described in Section 15.24. In more detail, CASE tools for the complete process are discussed in Section 15.24.1 and integrated develop- ment environments in Section 15.24.2. Environments for business applications are presented in Section

15.24.3. Section 15.24.4 is devoted to public tool infrastructures. Next, potential problems with environ- ments are discussed (Section 15.24.5). Now CASE tools for the test workflow are described (Section 15.25). Metrics for the implementation workflow are discussed in Section 15.26. The chapter concludes with an analysis of the challenges of the implementation workflow (Section 15.27).

An overview of the MSG Foundation case study for Chapter 15 appears in Figure 15.19.

**For Further Reading**

The attitudes of 43 organizations to 4GLs are reported in [Guimaraes, 1985]. Klepper and Bock [1995] describes how McDonnell Douglas obtained higher productivity with 4GLs than with 3GLs. Some of the dangers of end-user programming are presented in [Harrison, 2004]. A wide variety of papers on end-user programming appear in the November 2004 issue of the *Communications of the ACM*. Localization techniques to assist end users in debugging spreadsheets are described in [Ruthruff, Burnett, and Rothermel, 2006].

Excellent books on good programming practice include [Kernighan and Plauger, 1974] and [Mc- Connell, 1993].

Probably the most important early work on execution-based testing is [Myers, 1979]. A compre- hensive source of information on testing in general is [Beizer, 1990]. Functional testing is described in [Howden, 1987]. Black-box testing is described in detail in [Beizer, 1995]. The design of black-box test cases is presented in [Yamaura, 1998]. The relationship between the various coverage measures of structural testing and software quality is discussed in [Horgan, London, and Lyu, 1994]. A formal approach to glass-box testing is described in [Stocks and Carrington, 1996]. Elbaum, Malishevsky, and Rothermel [2002] discuss setting test case priorities. Generation of synthetic workloads for stress testing is presented in [Krishnamurthy, Rolia, and Majumdar, 2006]. A comprehensive list of unit- testing strategies appears in [Juristo, Moreno, Vegas, and Solari, 2006]. Geographically and tempo- rally distributed code reviews are presented in [Meyer, 2008].

Cleanroom is described in [Linger, 1994]. The use of Cleanroom during postdelivery mainte- nance is presented in [Sherer, Kouchakdjian, and Arnold, 1996]. A criticism of Cleanroom is given in [Beizer, 1997].

A good introduction to software reliability is [Musa and Everett, 1990]. In addition, the proceed- ings of the annual International Symposium on Software Reliability Engineering contain a wide variety of articles on software reliability.

The proceedings of the International Symposia on Software Testing and Analysis cover a particu- larly broad range of testing issues.

A survey of different approaches to the testing of objects can be found in [Turner, 1994]. Two impor- tant papers on the subject are [Perry and Kaiser, 1990] and [Harrold, McGregor, and Fitzpatrick, 1992].

[Beizer, 1995], mentioned previously, also covers black-box testing of object-oriented software. With regard to the object-oriented paradigm, Jorgensen and Erickson [1994] describe the integration testing of object-oriented software.

With regard to metrics for implementation, McCabe’s cyclomatic complexity was first presented in [McCabe, 1976]. Extensions of the metric to design appear in [McCabe and Butler, 1989]. Articles questioning the validity of cyclomatic complexity include [Shepperd and Ince, 1994]. The validity of object-oriented metrics is discussed in [Alshayeb and Li, 2003]. The relative inability of object- oriented metrics to detect high-impact faults is described in [Zhou and Leung, 2006].

Selection of test data for integration testing appears in [Harrold and Soffa, 1991]. The generation of test cases for testing GUIs is described in [Memon, Pollack, and Soffa, 2001].

Every 2 or 3 years, ACM SIGSOFT and SIGPLAN sponsor a Symposium on Practical Software Development Environments. The proceedings provide information on a broad spectrum of toolkits and environments. Also useful are the proceedings of the annual International Workshops on Computer- Aided Software Engineering.

With regard to PCTE, [Long and Morris, 1993] contains a number of information sources on that topic.

Chương này trình bày các vấn đề khác nhau liên quan đến việc triển khai sản phẩm của một nhóm. Chúng bao gồm lựa chọn ngôn ngữ lập trình (Phần 15.1). Vấn đề ngôn ngữ thế hệ thứ tư được thảo luận chi tiết trong Phần 15.2. Thực hành lập trình tốt được mô tả trong Phần 15.3 và nhu cầu về các tiêu chuẩn viết mã thực tế được trình bày trong Phần 15.4. Sau đó, các nhận xét được đưa ra liên quan đến việc sử dụng lại mã (Phần 15.5). Các hoạt động triển khai và tích hợp phải được tiến hành song song (Mục 15.6). Tích hợp từ trên xuống, từ dưới lên và bánh sandwich được mô tả và so sánh (Phần 15.6.1 đến 15.6.3). Tích hợp các sản phẩm hướng đối tượng được thảo luận trong Phần 15.6.4 và quản lý tích hợp trong Phần 15.6.5. Quy trình triển khai được trình bày trong Phần 15.7 và được áp dụng cho nghiên cứu tình huống của MSG Foundation trong Phần 15.8. Tiếp theo, các khía cạnh triển khai của quy trình thử nghiệm được trình bày (Phần 15.9). Các ca kiểm thử phải được chọn một cách có hệ thống (Mục 15.10). Các kỹ thuật kiểm thử đơn vị dựa trên hộp đen, hộp thủy tinh và không dựa trên thực thi khác nhau được mô tả (Phần 15.11, 15.13 và 15.14 tương ứng) và sau đó được so sánh (Phần 15.15). Thử nghiệm hộp đen của nghiên cứu trường hợp MSG Foundation được trình bày trong Phần 15.12. Kỹ thuật Phòng sạch được mô tả trong Phần 15.16. Các đối tượng thử nghiệm được thảo luận trong Phần 15.17, tiếp theo là phần thảo luận về ý nghĩa quản lý của thử nghiệm đơn vị (Phần 15.18). Một vấn đề khác là khi nào thì thực hiện lại thay vì gỡ lỗi một phần mềm mã (Phần 15.19). Thử nghiệm tích hợp được mô tả trong Phần 15.20, thử nghiệm sản phẩm trong Phần 15.21 và thử nghiệm chấp nhận trong Phần 15.22. Quy trình thử nghiệm cho nghiên cứu điển hình của Tổ chức MSG được nêu trong Phần 15.23. Các công cụ CASE cho quy trình triển khai được mô tả trong Phần 15.24. Chi tiết hơn, các công cụ CASE cho quy trình hoàn chỉnh được thảo luận trong Phần 15.24.1 và các môi trường phát triển tích hợp trong Phần 15.24.2. Môi trường cho các ứng dụng kinh doanh được trình bày trong Phần

15.24.3. Mục 15.24.4 dành cho cơ sở hạ tầng công cụ công cộng. Tiếp theo, các vấn đề tiềm ẩn về môi trường sẽ được thảo luận (Phần 15.24.5). Bây giờ các công cụ CASE cho quy trình thử nghiệm được mô tả (Phần 15.25). Các số liệu cho quy trình triển khai được thảo luận trong Phần 15.26. Chương này kết thúc với một phân tích về những thách thức của quy trình thực hiện (Phần 15.27).

Tổng quan về nghiên cứu điển hình của Tổ chức MSG cho Chương 15 xuất hiện trong Hình 15.19

Thái độ của 43 tổ chức đối với 4GL được báo cáo trong [Guimaraes, 1985]. Klepper và Bock [1995] mô tả cách McDonnell Douglas đạt được năng suất cao hơn với 4GL so với 3GL. Một số mối nguy hiểm của lập trình người dùng cuối được trình bày trong [Harrison, 2004]. Rất nhiều bài viết về lập trình người dùng cuối xuất hiện trong số ra tháng 11 năm 2004 của tạp chí Communications of the ACM. Các kỹ thuật bản địa hóa để hỗ trợ người dùng cuối gỡ lỗi bảng tính được mô tả trong [Ruthruff, Burnett và Rothermel, 2006].

Những cuốn sách hay về thực hành lập trình tốt bao gồm [Kernighan và Plauger, 1974] và [Mc-Connell, 1993].

Có lẽ công việc ban đầu quan trọng nhất về thử nghiệm dựa trên thực thi là [Myers, 1979]. Một nguồn thông tin toàn diện về thử nghiệm nói chung là [Beizer, 1990]. Kiểm thử chức năng được mô tả trong [Howden, 1987]. Kiểm thử hộp đen được mô tả chi tiết trong [Beizer, 1995]. Thiết kế các ca kiểm thử hộp đen được trình bày trong [Yamaura, 1998]. Mối quan hệ giữa các biện pháp bao phủ khác nhau của kiểm thử cấu trúc và chất lượng phần mềm được thảo luận trong [Horgan, London, và Lyu, 1994]. Một cách tiếp cận chính thức đối với thử nghiệm hộp thủy tinh được mô tả trong [Stocks và Carrington, 1996]. Elbaum, Malishevsky và Rothermel [2002] thảo luận về việc thiết lập các ưu tiên cho trường hợp thử nghiệm. Việc tạo khối lượng công việc tổng hợp cho kiểm thử căng thẳng được trình bày trong [Krishnamurthy, Rolia, và Majumdar, 2006]. Một danh sách đầy đủ các chiến lược kiểm thử đơn vị xuất hiện trong [Juristo, Moreno, Vegas và Solari, 2006]. Đánh giá mã phân tán theo thời gian và địa lý được trình bày trong [Meyer, 2008].

Phòng sạch được mô tả trong [Linger, 1994]. Việc sử dụng Phòng sạch trong quá trình bảo trì sau giao hàng được trình bày trong [Sherer, Kouchakdjian, và Arnold, 1996]. Một lời chỉ trích về Phòng sạch được đưa ra trong [Beizer, 1997].

Một giới thiệu tốt về độ tin cậy của phần mềm là [Musa và Everett, 1990]. Ngoài ra, thủ tục tố tụng của Hội nghị chuyên đề quốc tế hàng năm về Kỹ thuật độ tin cậy của phần mềm chứa rất nhiều bài viết về độ tin cậy của phần mềm.

Thủ tục tố tụng của Hội nghị chuyên đề quốc tế về kiểm thử và phân tích phần mềm bao gồm một loạt các vấn đề kiểm thử đặc biệt.

Có thể tìm thấy một cuộc khảo sát về các cách tiếp cận khác nhau đối với việc thử nghiệm các đối tượng trong [Turner, 1994]. Hai bài báo quan trọng về chủ đề này là [Perry và Kaiser, 1990] và [Harrold, McGregor, và Fitzpatrick, 1992].

[Beizer, 1995], đã được đề cập trước đây, cũng đề cập đến kiểm thử hộp đen của phần mềm hướng đối tượng. Liên quan đến mô hình hướng đối tượng, Jorgensen và Erickson [1994] mô tả thử nghiệm tích hợp của phần mềm hướng đối tượng.

Liên quan đến các thước đo để thực hiện, độ phức tạp chu trình của McCabe lần đầu tiên được trình bày trong [McCabe, 1976]. Các phần mở rộng của thước đo để thiết kế xuất hiện trong [McCabe và Butler, 1989]. Các bài báo đặt câu hỏi về tính hợp lệ của độ phức tạp chu trình bao gồm [Shepperd và Ince, 1994]. Hiệu lực của các thước đo hướng đối tượng được thảo luận trong [Alshayeb và Li, 2003]. Sự bất lực tương đối của các phép đo hướng đối tượng để phát hiện các lỗi có tác động cao được mô tả trong [Zhou và Leung, 2006].

Lựa chọn dữ liệu thử nghiệm cho thử nghiệm tích hợp xuất hiện trong [Harrold và Soffa, 1991]. Việc tạo các ca kiểm thử để kiểm thử GUI được mô tả trong [Memon, Pollack, and Soffa, 2001].

Cứ sau 2 hoặc 3 năm, ACM SIGSOFT và SIGPLAN tài trợ cho Hội nghị chuyên đề về Môi trường phát triển phần mềm thực tế. Quá trình tố tụng cung cấp thông tin về một loạt các bộ công cụ và môi trường. Kỷ yếu của các Hội thảo Quốc tế hàng năm về Công nghệ Phần mềm Hỗ trợ Máy tính cũng hữu ích.

Liên quan đến PCTE, [Long và Morris, 1993] có một số nguồn thông tin về chủ đề đó..

**Key Terms** acceptance testing *535*

all-definition-use-path coverage *526*

behavioral testing *517*

black-box testing *517*

bottom-up integration *513* boundary value analysis *521* branch coverage *526*

Cleanroom *529*

code artifact *516*

coding standards *509*

complexity *527*

component *516*

consistent variable names *504*

cyclomatic complexity *527*

data-driven testing *517*

debugging *533*

defensive programming *512*

driver *511*

end-user programming *503*

environment *538*

equivalence class *521*

execution-based testing *516*

first-generation language *501*

fourth-generation language (4GL) *501*

functional analysis *523*

functional testing *517*

glass-box testing *517*

good programming practice *504*

Hungarian Naming Conventions *505*

implementation workflow *516*

input/output-driven testing *517*

integrated environment *538*

integration *510*

integration testing *535* linear code sequences *526* logic artifact *511*

logic-driven testing *517*

meaningful variable names *504*

method-based environment *539*

non-execution-based testing *516*

nonprocedural *502*

operational artifact *511*

path coverage *526*

path-oriented testing *517*

portable common tool environment (PCTE) *540*

procedural *502*

process integration *538*

product testing *535*

programming-in-the-many *498*

prologue comments *506*

reliable *520*

sandwich integration *514*

second-generation language *501*

self-documenting code *505*

statement coverage *526*

static method *515*

stress testing *536*

structural test *526*

structural testing *517*

structured testing *528*

stub *510*

technique-based environment *538*

test case selection *527* testing fault rate *530* testing to code *517*

testing to specifications *517*

third-generation language *501*

tool *538*

tool integration *538*

top-down integration *511* user interface integration *538* valid *520*

unit testing *516*

volume testing *536*

white-box testing *517*

workbench *538*

Chapter 15 *Implementation* **545**

**Problems**

* 1. Your instructor has asked you to implement the Chocoholics Anonymous product (Appendix A). Which language would you choose for implementing the product, and why? Of the various languages available to you, list their benefits and their costs. Do not attempt to attach dollar values to your answers.
  2. Repeat Problem 15.1 for the elevator problem (Section 12.7.1).
  3. Repeat Problem 15.1 for the automated library circulation system (Problem 8.7).
  4. Repeat Problem 15.1 for the product that determines whether a bank statement is correct (Problem 8.8).
  5. Repeat Problem 15.1 for the automated teller machine (Problem 8.9).
  6. Add prologue comments to a code artifact that you have recently implemented.
  7. How do coding standards for a one-person software production company differ from those in organizations with 300 software professionals?
  8. How do coding standards for a software company that develops and maintains software for intensive-care units differ from those in an organization that develops and maintains account- ing products?
  9. Consider the statement

<*condition* 1> && <*condition* 2>

As stated at the end of Section 15.3, in Java and C++ the semantics of the && operator are such that if <*condition* 1> is false, then <*condition* 2> is not evaluated. What is the technical term for this?

* 1. Consider the statement

<*condition* 1> **and** <*condition* 2>

In what programming languages is <*condition* 2> evaluated even if <*condition* 1> is false?

* 1. Why does deep nesting of **if**-statements frequently lead to code that can be difficult to read?
  2. Why has it been suggested that modules ideally should consist of between 35 and 50 state- ments?
  3. Why should backward **goto** statements be avoided, whereas a forward **goto** may be used for error handling?
  4. Set up black-box test cases for Naur’s text-processing problem (Section 6.5.2). For each test case, state what is being tested and the expected outcome of that test case.
  5. Using your solution to Problem 6.14 (or code distributed by your instructor), set up statement coverage test cases. For each test case, state what is being tested and the expected outcome of that test case.
  6. Repeat Problem 15.15 for branch coverage.
  7. Repeat Problem 15.15 for all-definition-use-path coverage.
  8. Repeat Problem 15.15 for path coverage.
  9. Repeat Problem 15.15 for linear code sequences.
  10. Draw a flowchart of your solution to Problem 6.14 (or code distributed by your instructor). Determine its cyclomatic complexity. If you are unable to determine the number of branches, consider the flowchart as a directed graph. Determine the number of edges *e*, nodes *n*, and connected components *c*. (Each method constitutes a connected component.) The cyclomatic complexity *M* is then given by the formula [McCabe, 1976]

*M* = *e* − *n* + 2*c*

**546** Part B *The Workflows of the Software Life Cycle*

* 1. You are the owner and sole employee of One-Person Software Company. You bought the pro- gramming workbench described in Section 5.8. List its five capabilities in order of importance to you, giving reasons.
  2. You are now the vice-president for software technology of Very Big Software Company; there are 17,500 employees in your organization. How do you rank the capabilities of the program- ming workbench described in Section 5.8? Explain any differences between your answer to this problem and that of Problem 15.21.
  3. As SQA manager for a software development organization, you are responsible for determin- ing the maximum number of faults that may be found in a given code artifact during testing. If this maximum is exceeded, then the code artifact must be redesigned and recoded. What criteria would you use to determine the maximum for a given code artifact?
  4. Explain the difference between logic artifacts and operational artifacts.
  5. Defensive programming is good software engineering practice. At the same time, it can pre- vent operational artifacts from being tested thoroughly enough for reuse purposes. How can this apparent contradiction be resolved?
  6. What are the similarities between product testing and acceptance testing? What are the major differences?
  7. What is the role of the SQA group during implementation?
  8. You are the owner and sole employee of One-Person Software Company. You decide that to be competitive you must buy CASE tools. You therefore apply for a bank loan for $15,000. Your bank manager asks you for a statement no more than one page in length (preferably shorter) explaining in lay terms why you need CASE tools. Write the statement.
  9. The newly appointed vice-president for software development of Ye Olde Fashioned Software Corporation has hired you to help her change the way the company develops software. There are 650 employees, all writing COBOL 85 code without the assistance of any CASE tools (COBOL 85 conforms to the 1985 COBOL standard; it is not object-oriented). Write a memo to the vice-president stating what sort of CASE equipment the company should purchase. Justify your choice.
  10. You and a friend decide to start Personal Computer Software Programs ’R Us, developing software for personal computers on personal computers. Then a distant cousin dies, leaving you $1 million on condition that you spend the money on a business-oriented environment and the hardware needed to run it and that you keep the environment for at least 5 years. What do you do, and why?
  11. You are a computer science professor at an excellent small liberal arts college. Programming assignments for computer science courses are done on a network of 35 personal computers. Your dean asks you whether to use the limited software budget to buy CASE tools, bearing in mind that, unless some sort of site license can be obtained, 35 copies of every CASE tool have to be purchased. What do you advise?
  12. You have just been elected mayor of a major city. You discover that no CASE tools are being used to develop software for the city. What do you do?
  13. (Term Project) Draw up black-box test cases for the product you specified in Problem 12.20 or 13.22. For each test case, state what is being tested and the expected outcome of that test case.
  14. (Term Project) Implement and integrate the Chocoholics Anonymous product (Appendix A). Use the programming language specified by your instructor. Your instructor will tell you whether to build a Web-based user interface, a graphical user interface, or a text-based user interface. Remember to utilize the black-box test cases you developed in Problem 15.33 for testing your code.

Chapter 15 *Implementation* **547**

* 1. (Case Study) Download a copy of the implementation of the MSG Foundation product de- scribed in Section 15.8. Draw up statement coverage test cases for the product. For each test case, state what is being tested and the expected outcome of that test case.
  2. (Case Study) Repeat Problem 15.35 for branch coverage.
  3. (Case Study) Repeat Problem 15.35 for all-definition-use-path coverage.
  4. (Case Study) Repeat Problem 15.35 for path coverage.
  5. (Case Study) Repeat Problem 15.35 for linear code sequences.
  6. (Case Study) Starting with the detailed design of Problem 14.16, code the MSG Foundation case study in an object-oriented language other than C++ or Java.
  7. (Case Study) Recode the MSG Foundation case study (Section 15.8) in pure C, with no C++ features. Although C does not support inheritance, object-based concepts such as encapsu- lation and information hiding can be achieved relatively easily. How would you implement polymorphism and dynamic binding?
  8. (Case Study) To what extent is the documentation of the code of the implementation of Section

15.8 inadequate? Make any necessary additions.

* 1. (Readings in Software Engineering) Your instructor will distribute copies of [Meyer, 2008]. What are your views on geographically and temporally distributed code reviews?

**References** [Alshayeb and Li, 2003] M. ALSHAYEB, AND W. LI, “An Empirical Validation of Object-Oriented Met- rics in Two Different Iterative Software Processes,” *IEEE Transactions on Software Engineering* **29** (November 2003), pp. 1043–49.

[Andersson and Runeson, 2007] C. ANDERSSON AND P. RUNESON, “A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems,” *IEEE Transactions on Software Engineering* **33** (May 2007), pp. 273–86.

[Basili and Hutchens, 1983] V. R. BASILI AND D. H. HUTCHENS, “An Empirical Study of a Syntac- tic Complexity Family,” *IEEE Transactions on Software Engineering* **SE-9** (November 1983), pp. 664–72.

[Basili and Selby, 1987] V. R. BASILI AND R. W. SELBY, “Comparing the Effectiveness of Software Testing Strategies,” *IEEE Transactions on Software Engineering* **SE-13** (December 1987), pp. 1278–96.

[Basili and Weiss, 1984] V. R. BASILI AND D. M. WEISS, “A Methodology for Collecting Valid Soft- ware Engineering Data,” *IEEE Transactions on Software Engineering* **SE-10** (November 1984), pp. 728–38.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques,* 2nd ed., Van Nostrand Reinhold, New York, 1990. [Beizer, 1995] B. BEIZER, *Black-Box Testing: Techniques for Functional Testing of Software and Sys-*

*tems,* John Wiley and Sons, New York, 1995.

[Beizer, 1997] B. BEIZER, “Cleanroom Process Model: A Critical Examination,” *IEEE Software* **14**

(March–April 1997), pp. 14–16.

[Binkley and Schach, 1996] A. B. BINKLEY AND S. R. SCHACH, “A Comparison of Sixteen Qual- ity Metrics for Object-Oriented Design,” *Information Processing Letters* **57** (No. 6, June 1996), pp. 271–75.

[Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, “Toward a Unified Approach to Object- Oriented Coupling,” *Proceedings of the 35th Annual ACM Southeast Conference*, Murfreesboro, TN, April 2–4, ACM, 1997, pp. 91–97.

[Borland, 2002] BORLAND, “Press Release: Borland Unveils C++ Application Development Strategy for 2002,” [www.borland.com/news/press\_releases/2002/01\_28\_02\_cpp.strategy.html](http://www.borland.com/news/press_releases/2002/01_28_02_cpp.strategy.html),

January 28, 2002.

**548** Part B *The Workflows of the Software Life Cycle*

[Chidamber and Kemerer, 1994] S. R. CHIDAMBER AND C. F. KEMERER, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–93.

[Crossman, 1982] T. D. CROSSMAN, “Inspection Teams, Are They Worth It?” *Proceedings of the Sec- ond National Symposium on EDP Quality Assurance*, Chicago, ACM, November 1982.

[Date, 2003] C. J. DATE, *An Introduction to Database Systems*, 8th ed., Addison-Wesley, Reading, MA, 2003.

[Dunn, 1984] R. H. DUNN, *Software Defect Removal*, McGraw-Hill, New York, 1984.

[Elbaum, Malishevsky, and Rothermel, 2002] S. ELBAUM, A. G. MALISHEVSKY, AND G. ROTHERMEL, “Test Case Prioritization: A Family of Empirical Studies,” *IEEE Transactions on Software Engi- neering* **28** (February 2002), pp. 159–82.

[Endres, 1975] A. ENDRES, “An Analysis of Errors and Their Causes in System Programs,” *IEEE Transactions on Software Engineering* **SE-1** (June 1975), pp. 140–49.

[Grady, 1992] R. B. GRADY, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Guimaraes, 1985] T. GUIMARAES, “A Study of Application Program Development Techniques,”

*Communications of the ACM* **28** (May 1985), pp. 494–99.

[Harel et al., 1990] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A.

SHTULL-TRAURING, AND M. TRAKHTENBROT, “STATEMATE: A Working Environment for the De- velopment of Complex Reactive Systems,” *IEEE Transactions on Software Engineering* **16** (April 1990), pp. 403–14.

[Harrison, 2004] W. HARRISON, “The Dangers of End-User Programming,” *IEEE Software* **21** (July– August 2004), pp. 5–7.

[Harrold and Soffa, 1991] M. J. HARROLD AND M. L. SOFFA, “Selecting and Using Data for Integra- tion Testing,” *IEEE Software* **8** (1991), pp. 58–65.

[Harrold, McGregor, and Fitzpatrick, 1992] M. J. HARROLD, J. D. MCGREGOR, AND K. J. FITZPATRICK, “Incremental Testing of Object-Oriented Class Structures,” *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992, IEEE, pp. 68–80.

[Horgan, London, and Lyu, 1994] J. R. HORGAN, S. LONDON, AND M. R. LYU, “Achieving Software Quality with Testing Coverage Measures,” *IEEE Computer* **27** ( 1994), pp. 60–69.

[Howden, 1987] W. E. HOWDEN, *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

[Hwang, 1981] S.-S. V. HWANG, “An Empirical Study in Functional Testing, Structural Testing, and Code Reading Inspection,” Scholarly Paper 362, Department of Computer Science, University of Maryland, College Park, 1981.

[Jacobson, Booch, and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jorgensen and Erickson, 1994] P. C. JORGENSEN AND C. ERICKSON, “Object-Oriented Integration Testing,” *Communications of the ACM* **37** (September 1994), pp. 30–38.

[Juristo, Moreno, Vegas, and Solari, 2006] N. JURISTO, A. M. MORENO, S. VEGAS, AND M. SOLARI,

“In Search of What We Experimentally Know about Unit Testing,” *IEEE Software* **23** (November– December 2006), pp. 72–80.

[Kernighan and Plauger, 1974] B. W. KERNIGHAN AND P. J. PLAUGER, *The Elements of Programming Style*, McGraw-Hill, New York, 1974.

[Klepper and Bock, 1995] R. KLEPPER AND D. BOCK, “Third and Fourth Generation Productivity Dif- ferences,” *Communications of the ACM* **38** (September 1995), pp. 69–79.

[Klunder, 1988] D. KLUNDER, “Hungarian Naming Conventions,” Technical Report, Microsoft Cor- poration, Redmond, WA, January 1988.

Chapter 15 *Implementation* **549**

[Krishnamurthy, Rolia, and Majumdar, 2006] D. KRISHNAMURTHY, J. A. ROLIA, AND S. MAJUMDAR, “A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems,” *IEEE Transactions on Software Engineering* **32** (November 2006), pp. 868–82.

[Linger, 1994] R. C. LINGER, “Cleanroom Process Model,” *IEEE Software* **11** (March 1994), pp. 50–58.

[Long and Morris, 1993] F. LONG AND E. MORRIS, “An Overview of PCTE: A Basis for a Portable Common Tool Environment,” Technical Report CMU/SEI–93–TR–1, Software Engineering Insti- tute, Carnegie Mellon University, Pittsburgh, January 1993.

[Martin, 1985] J. MARTIN, *Fourth-Generation Languages,* Vols. 1, 2, and 3, Prentice Hall, Englewood Cliffs, NJ, 1985.

[McCabe, 1976] T. J. MCCABE, “A Complexity Measure,” *IEEE Transactions on Software Engineer- ing* **SE-2** (December 1976), pp. 308–20.

[McCabe and Butler, 1989] T. J. MCCABE AND C. W. BUTLER, “Design Complexity Measurement and Testing,” *Communications of the ACM* **32** (December 1989), pp. 1415–25.

[McConnell, 1993] S. MCCONNELL, *Code Complete: A Practical Handbook of Software Construc- tion*, Microsoft Press, Redmond, WA, 1993.

[Memon, Pollack, and Soffa, 2001] A. M. MEMON, M. E. POLLACK, AND M. L. SOFFA, “Hierarchical GUI Test Case Generation Using Automated Planning,” *IEEE Transactions on Software Engineering* **27** (February 2001), pp. 144–55.

[Meyer, 2008] B. MEYER, “Design and Code Reviews in the Age of the Internet,” *Communications of the ACM* **51** (September 2008), pp. 66–71.

[Mills, Dyer, and Linger, 1987] H. D. MILLS, M. DYER, AND R. C. LINGER, “Cleanroom Software Engineering,” *IEEE Software* **4** (September 1987), pp. 19–25.

[Musa and Everett, 1990] J. D. MUSA AND W. W. EVERETT, “Software-Reliability Engineering: Tech- nology for the 1990s,” *IEEE Software* **7** (November 1990), pp. 36–43.

[Musa, Iannino, and Okumoto, 1987] J. D. MUSA, A. IANNINO, AND K. OKUMOTO, *Software Reliabil- ity: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

[Myers, 1976] G. J. MYERS, *Software Reliability: Principles and Practices*, Wiley-Interscience, New York, 1976.

[Myers, 1978a] G. J. MYERS, “A Controlled Experiment in Program Testing and Code Walkthroughs/ Inspections,” *Communications of the ACM* **21** (September 1978), pp. 760–68.

[Myers, 1979] G. J. MYERS, *The Art of Software Testing*, John Wiley and Sons, New York, 1979. [Perry and Kaiser, 1990] D. E. PERRY AND G. E. KAISER, “Adequate Testing and Object-Oriented

Programming,” *Journal of Object-Oriented Programming* **2** (January–February 1990), pp. 13–19.

[Rapps and Weyuker, 1985] S. RAPPS AND E. J. WEYUKER, “Selecting Software Test Data Using Data Flow Information,” *IEEE Transactions on Software Engineering* **SE-11** (April 1985), pp. 367–75.

[Runeson et al., 2006] P. RUNESON, C. ANDERSSON, T. THELIN, A. ANDREWS, AND T. BERLING,

“What Do We Know about Defect Detection Methods?” *IEEE Software* **23** (May–June 2006), pp. 82–90.

[Ruthruff, Burnett, and Rothermel, 2006] J. R. RUTHRUFF, M. BURNETT, AND G. ROTHERMEL, “Interactive Fault Localization Techniques in a Spreadsheet Environment,” *IEEE Transactions on Software Engineering* **32** (April 2006), pp. 213–39.

[Sammet, 1978] J. E. SAMMET, “The Early History of COBOL,” *Proceedings of the History of Pro- gramming Languages Conference*, Los Angeles, ACM, 1978, pp. 199–276.

[Shepperd and Ince, 1994] M. SHEPPERD AND D. C. INCE, “A Critique of Three Metrics,” *Journal of Systems and Software* **26** (September 1994), pp. 197–210.

**550** Part B *The Workflows of the Software Life Cycle*

[Sherer, Kouchakdjian, and Arnold, 1996] S. W. SHERER, A. KOUCHAKDJIAN, AND P. G. ARNOLD, “Expe- rience Using Cleanroom Software Engineering,” *IEEE Software* **13** (May 1996), pp. 69–76.

[Stocks and Carrington, 1996] P. STOCKS AND D. CARRINGTON, “A Framework for Specification-Based Testing,” *IEEE Transactions on Software Engineering* **22** (November 1996), pp. 777–93.

[Takahashi and Kamayachi, 1985] M. TAKAHASHI AND Y. KAMAYACHI, “An Empirical Study of a Model for Program Error Prediction,” *Proceedings of the Eighth International Conference on Software Engineering*, London, IEEE, 1985, pp. 330–36.

[Trammel, Binder, and Snyder, 1992] C. J. TRAMMEL, L. H. BINDER, AND C. E. SNYDER, “The

Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering,” *ACM Transactions on Software Engineering and Methodology* **1** (January 1992), pp. 81–94.

[Turner, 1994] C. D. TURNER, “State-Based Testing: A New Method for the Testing of Object- Oriented Programs,” Ph.D. thesis, Computer Science Division, University of Durham, Durham, UK, November 1994.

[Walsh, 1979] T. J. WALSH, “A Software Reliability Study Using a Complexity Measure,” *Proceed- ings of the AFIPS National Computer Conference*, New York, AFIPS, 1979, pp. 761–68.

[Watson and McCabe, 1996] A. H. WATSON AND T. J. MCCABE, “Structured Testing: A Testing Meth- odology Using the Cyclomatic Complexity Metric,” NIST Special Publication 500–235, Com- puter Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

[Weyuker, 1988] E. J. WEYUKER, “An Empirical Study of the Complexity of Data Flow Testing,” *Pro- ceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, IEEE, July 1988, pp. 188–95.

[Wilde, Matthews, and Huitt, 1993] N. WILDE, P. MATTHEWS, AND R. HUITT, “Maintaining Object- Oriented Software,” *IEEE Software* **10** (January 1993), pp. 75–80.

[Woodward, Hedley, and Hennell, 1980] M. R. WOODWARD, D. HEDLEY, AND M. A. HENNELL, “Expe- rience with Path Analysis and Testing of Programs,” *IEEE Transactions on Software Engineering* **SE-6** (May 1980), pp. 278–86.

[Yamaura, 1998] T. YAMAURA, “How to Design Practical Test Cases,” *IEEE Software* **15** (November– December 1998), pp. 30–36.

[Yourdon, 1989] E. YOURDON, *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, NJ, 1989.

[Zhou and Leung, 2006] Y. ZHOU AND H. LEUNG, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults,” *IEEE Transactions on Software Engineering* **32** (October 2006), pp. 771–89.

Chapter

16