FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Software Testing and Analysis: Process, Principles, and Techniques

20.7 Improving the Process . . . 394
20.8 The Quality Team . . . 399

21 Integration and Component-based Software Testing 405
21.1 Overview . . . 405
21.2 Integration Testing Strategies . . . 408
21.3 Testing Components and Assemblies . . . 413

22 System, Acceptance, and Regression Testing 417
22.1 Overview . . . 417
22.2 System Testing . . . 418
22.3 Acceptance Testing . . . 421
22.4 Usability . . . 423
22.5 Regression Testing . . . 427
22.6 Regression Test Selection Techniques . . . 428
22.7 Test Case Prioritization and Selective Execution . . . 434

23 Automating Analysis and Test 439
23.1 Overview . . . 439
23.2 Automation and Planning . . . 441
23.3 Process Management . . . 441
23.4 Static Metrics . . . 443
23.5 Test Case Generation and Execution . . . 445
23.6 Static Analysis and Proof . . . 445
23.7 Cognitive Aids . . . 448
23.8 Version Control . . . 449
23.9 Debugging . . . 449
23.10 Choosing and Integrating Tools . . . 451

24 Documenting Analysis and Test 455
24.1 Overview . . . 455
24.2 Organizing Documents . . . 456
24.3 Test Strategy Document . . . 458
24.4 Analysis and Test Plan . . . 458
24.5 Test Design Specification Documents . . . 460
24.6 Test and Analysis Reports . . . 462

Bibliography 467

Index 479

Chapter 21

Integration and Component-based Software Testing

Problems arise in integration even of well-designed modules and components. Integration testing aims to uncover interaction and compatibility problems as early as possible. This chapter presents integration testing strategies, including the increasingly important problem of testing integration with commercial off-the-shelf (COTS) components, libraries, and frameworks.

Required Background

- Chapter 4
  Basic concepts of quality process, goals, and activities are important for understanding this chapter.
- Chapter 17
  Scaffolding is a key cost element of integration testing. Some knowledge about scaffolding design and implementation is important to fully understand an essential dimension of integration testing.

21.1 Overview

The traditional *V* model introduced in Chapter 2 divides testing into four main levels of granularity: module, integration, system, and acceptance test. Module or unit test checks module behavior against specifications or expectations; integration test checks module compatibility; system and acceptance tests check behavior of the whole system with respect to specifications and user needs, respectively.

An effective integration test is built on a foundation of thorough module testing and inspection. Module test maximizes controllability and observability of an individual

unit, and is more effective in exercising the full range of module behaviors, rather than just those that are easy to trigger and observe in a particular context of other modules. While integration testing may to some extent act as a process check on module testing (i.e., faults revealed during integration test can be taken as a signal of unsatisfactory unit testing), thorough integration testing cannot fully compensate for sloppiness at the module level. In fact, the quality of a system is limited by the quality of the modules and components from which it is built, and even apparently noncritical modules can have widespread effects. For example, in 2004 a buffer overflow vulnerability in a single, widely used library for reading Portable Network Graphics (PNG) files caused security vulnerabilities in Windows, Linux, and Mac OS X Web browsers and email clients.

On the other hand, some unintended side-effects of module faults may become apparent only in integration test (see sidebar on page 409), and even a module that satisfies its interface specification may be incompatible because of errors introduced in design decomposition. Integration tests therefore focus on checking compatibility between module interfaces.

Integration faults are ultimately caused by incomplete specifications or faulty implementations of interfaces, resource usage, or required properties. Unfortunately, it may be difficult or not cost-effective to anticipate and completely specify all module interactions. For example, it may be very difficult to anticipate interactions between remote and apparently unrelated modules through sharing a temporary hidden file that just happens to be given the same name by two modules, particularly if the name clash appears rarely and only in some installation configurations. Some of the possible manifestations of incomplete specifications and faulty implementations are summarized in Table 21.1.

The official investigation of the Ariane 5 accident that led to the loss of the rocket on July 4, 1996 concluded that the accident was caused by incompatibility of a software module with the Ariane 5 requirements. The software module was in charge of computing the horizontal bias, a value related to the horizontal velocity sensed by the platform that is calculated as an indicator of alignment precision. The module had functioned correctly for Ariane 4 rockets, which were smaller than the Ariane 5, and thus had a substantially lower horizontal velocity. It produced an overflow when integrated into the Ariane 5 software. The overflow started a series of events that terminated with self-destruction of the launcher. The problem was not revealed during testing because of incomplete specifications:

> The specification of the inertial reference system and the tests performed at equipment level did not specifically include the Ariane 5 trajectory data. Consequently the realignment function was not tested under simulated Ariane 5 flight conditions, and the design error was not discovered. [From the official investigation report]

As with most software problems, integration problems may be attacked at many levels. Good design and programming practice and suitable choice of design and programming environment can reduce or even eliminate some classes of integration problems. For example, in applications demanding management of complex, shared

Integration fault	Example
Inconsistent interpretation of parameters or values Each module's interpretation may be reasonable, but they are incompatible.	Unit mismatch: A mix of metric and British measures (meters and yards) is believed to have led to loss of the Mars Climate Orbiter in September 1999.
Violations of value domains or of capacity or size limits Implicit assumptions on ranges of values or sizes.	Buffer overflow, in which an implicit (unchecked) capacity bound imposed by one module is violated by another, has become notorious as a security vulnerability. For example, some versions of the Apache 2 Web server between 2.0.35 and 2.0.50 could overflow a buffer while expanding environment variables during configuration file parsing.
Side-effects on parameters or resources	A module often uses resources that are not explicitly mentioned in its interface. Integration problems arise when these implicit effects of one module interfere with those of another. For example, using a temporary file "tmp" may be invisible until integration with another module that also attempts to use a temporary file "tmp" in the same directory of scratch files.
Missing or misunderstood functionality Underspecification of functionality may lead to incorrect assumptions about expected results.	Counting hits on Web sites may be done in many different ways: per unique IP address, per hit, including or excluding spiders, and so on. Problems arise if the interpretation assumed in the counting module differs from that of its clients.
Nonfunctional problems	Nonfunctional properties like performance are typically specified explicitly only when they are expected to be an issue. Even when performance is not explicitly specified, we expect that software provides results in a reasonable time. Interference between modules may reduce performance below an acceptable threshold.
Dynamic mismatches Many languages and frameworks allow for dynamic binding. Problems may be caused by failures in matchings when modules are integrated.	Polymorphic calls may be dynamically bound to incompatible methods, as discussed in Chapter 15.

This core taxonomy can be extended to effectively classify important or frequently occurring integration faults in particular domains.

Table 21.1: Integration faults.

structures, choosing a language with automatic storage management and garbage collection greatly reduces memory disposal errors such as dangling pointers and redundant deallocations ("double frees").

Even if the programming language choice is determined by other factors, many errors can be avoided by choosing patterns and enforcing coding standards across the entire code base; the standards can be designed in such a way that violations are easy to detect manually or with tools. For example, many projects using C or C++ require use of "safe" alternatives to unchecked procedures, such as requiring strncpy or strlcpy (string copy procedures less vulnerable to buffer overflow) in place of strcpy. Checking for the mere presence of strcpy is much easier (and more easily automated) than checking for its safe use. These measures do not eliminate the possibility of error, but integration testing is more effective when focused on finding faults that slip through these design measures.

21.2 Integration Testing Strategies

Integration testing proceeds incrementally with assembly of modules into successively larger subsystems. Incremental testing is preferred, first, to provide the earliest possible feedback on integration problems. In addition, controlling and observing the behavior of an integrated collection of modules grows in complexity with the number of modules and the complexity of their interactions. Complex interactions may hide faults, and failures that are manifested may propagate across many modules, making fault localization difficult. Therefore it is worthwhile to thoroughly test a small collection of modules before adding more.

A strategy for integration testing of successive partial subsystems is driven by the order in which modules are constructed (the build plan), which is an aspect of the system architecture. The build plan, in turn, is driven partly by the needs of test. Design and integration testing are so tightly coupled that in many companies the integration and the testing groups are merged in a single group in charge of both design and test integration.

Since incremental assemblies of modules are incomplete, one must often construct scaffolding — drivers, stubs, and various kinds of instrumentation — to effectively test them. This can be a major cost of integration testing, and it depends to a large extent on the order in which modules are assembled and tested.

big bang testing

One extreme approach is to avoid the cost of scaffolding by waiting until all modules are integrated, and testing them together — essentially merging integration testing into system testing. In this *big bang* approach, neither stubs nor drivers need be constructed, nor must the development be carefully planned to expose well-specified interfaces to each subsystem. These savings are more than offset by losses in observability, diagnosability, and feedback. Delaying integration testing hides faults whose effects do not always propagate outward to visible failures (violating the principle that failing always is better than failing sometimes) and impedes fault localization and diagnosis because the failures that are visible may be far removed from their causes. Requiring the whole system to be available before integration does not allow early test and feedback, and so faults that are detected are much more costly to repair. Big bang

Memory Leaks

Memory leaks are typical of program faults that often escape module testing. They may be detected in integration testing, but often escape further and are discovered only in actual system operation.

The Apache Web server, version 2.0.48, contained the following code for reacting to normal Web page requests that arrived on the secure (https) server port:

```
1  static void ssl_io_filter_disable(ap_filter_t *f)
2  {
3      bio_filter_in_ctx_t *inctx = f->ctx;
4      inctx->ssl = NULL;
5      inctx->filter_ctx->pssl = NULL;
6  }
```

This code fails to reclaim some dynamically allocated memory, causing the Web server to "leak" memory at run-time. Over a long period of use, or over a shorter period if the fault is exploited in a denial-of-service attack, this version of the Apache Web server will allocate and fail to reclaim more and more memory, eventually slowing to the point of unusability or simply crashing.

The fault is nearly impossible to see in this code. The memory that should be deallocated here is part of a structure defined and created elsewhere, in the SSL (secure sockets layer) subsystem, written and maintained by a different set of developers. Even reading the definition of the ap_filter_t structure, which occurs in a different part of the Apache Web server source code, doesn't help, since the ctx field is an opaque pointer (type void * in C) . The repair, applied in version 2.0.49 of the server, is:

```
1  static void ssl_io_filter_disable(SSLConnRec *sslconn, ap_filter_t *f)
2  {
3      bio_filter_in_ctx_t *inctx = f->ctx;
4      SSL_free(inctx->ssl);
5      sslconn->ssl = NULL;
6      inctx->ssl = NULL;
7      inctx->filter_ctx->pssl = NULL;
8  }
```

This memory leak illustrates several properties typical of integration faults. In principle, it stems from incomplete knowledge of the protocol required to interact with some other portion of the code, either because the specification is (inevitably) incomplete or because it is not humanly possible to remember everything. The problem is due at least in part to a weakness of the programming language — it would not have occurred in a language with automatic garbage collection, such as Java. Finally, although the fault would be very difficult to detect with conventional unit testing techniques, there do exist both static and dynamic analysis techniques that could have made early detection much more likely, as discussed in Chapter 18.

integration testing is less a rational strategy than an attempt to recover from a lack of planning; it is therefore also known as the *desperate tester* strategy.

structural integration test strategy

Among strategies for incrementally testing partially assembled systems, we can distinguish two main classes: *structural* and *feature oriented*. In a structural approach, modules are constructed, assembled, and tested together in an order based on hierarchical structure in the design. Structural approaches include bottom-up, top-down, and a combination sometimes referred to as sandwich or backbone strategy. Feature-oriented strategies derive the order of integration from characteristics of the application, and include threads and critical modules strategies.

top-down and bottom-up testing

Top-down and *bottom-up* strategies are classic alternatives in system construction and incremental integration testing as modules accumulate. They consist in sorting modules according to the use/include relation (see Chapter 15, page 286), and in starting testing from the top or from the bottom of the hierarchy, respectively.

A top-down integration strategy begins at the top of the uses hierarchy, including the interfaces exposed through a user interface or top-level application program interface (API). The need for drivers is reduced or eliminated while descending the hierarchy, since at each stage the already tested modules can be used as drivers while testing the next layer. For example, referring to the excerpt of the Chipmunk Web presence shown in Figure 21.1, we can start by integrating *CustomerCare* with *Customer*, while stubbing *Account* and *Order*. We could then add either *Account* or *Order* and *Package*, stubbing *Model* and *Component* in the last case. We would finally add *Model*, *Slot*, and *Component* in this order, without needing any driver.

Bottom-up integration similarly reduces the need to develop stubs, except for breaking circular relations. Referring again to the example in Figure 21.1, we can start bottom-up by integrating *Slot* with *Component*, using drivers for *Model* and *Order*. We can then incrementally add *Model* and *Order*. We can finally add either *Package* or *Account* and *Customer*, before integrating *CustomerCare*, without constructing stubs.

Top-down and bottom-up approaches to integration testing can be applied early in the development if paired with similar design strategies: If modules are delivered following the hierarchy, either top-down or bottom-up, they can be integrated and tested as soon as they are delivered, thus providing early feedback to the developers. Both approaches increase controllability and diagnosability, since failures are likely caused by interactions with the newly integrated modules.

In practice, software systems are rarely developed strictly top-down or bottom-up. Design and integration strategies are driven by other factors, like reuse of existing modules or commercial off-the-shelf (COTS) components, or the need to develop early prototypes for user feedback. Integration may combine elements of the two approaches, starting from both ends of the hierarchy and proceeding toward the middle. An early top-down approach may result from developing prototypes for early user feedback, while existing modules may be integrated bottom-up. This is known as the *sandwich* or *backbone* strategy. For example, referring once more to the small system of Figure 21.1, let us imagine reusing existing modules for *Model*, *Slot*, and *Component*, and developing *CustomerCare* and *Customer* as part of an early prototype. We can start integrating *CustomerCare* and *Customer* top down, while stubbing *Account* and *Order*. Meanwhile, we can integrate bottom-up *Model*, *Slot*, and *Component* with *Order*, using drivers for *Customer* and *Package*. We can then integrate *Account* with *Customer*,

sandwich or backbone

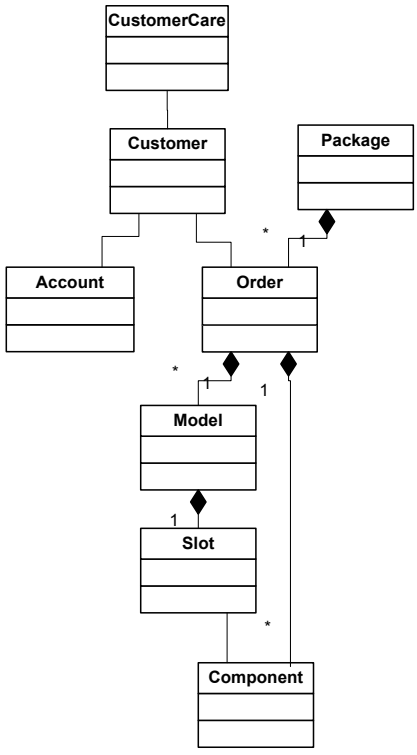

Figure 21.1: An excerpt of the class diagram of the Chipmunk Web presence. Modules are sorted from the top to the bottom according to the use/include relation. The topmost modules are not used or included in any other module, while the bottom-most modules do not include or use other modules.

and *Package* with *Order*, before finally integrating the whole prototype system.

The price of flexibility and adaptability in the sandwich strategy is complex planning and monitoring. While top-down and bottom-up are straightforward to plan and monitor, a sandwich approach requires extra coordination between development and test.

In contrast to structural integration testing strategies, feature-driven strategies select an order of integration that depends on the dynamic collaboration patterns among modules regardless of the static structure of the system. The *thread* integration testing strategy integrates modules according to system features. Test designers identify threads of execution that correspond to system features, and they incrementally test each thread. The thread integration strategy emphasizes module interplay for specific functionality.

thread testing

Referring to the Chipmunk Web presence, we can identify feature threads for assembling models, finalizing orders, completing payments, packaging and shipping, and so on. Feature thread integration fits well with software processes that emphasize incremental delivery of user-visible functionality. Even when threads do not correspond to usable end-user features, ordering integration by functional threads is a useful tactic to make flaws in integration externally visible.

Incremental delivery of usable features is not the only possible consideration in choosing the order in which functional threads are integrated and tested. Risk reduction is also a driving force in many software processes. *Critical module* integration testing focuses on modules that pose the greatest risk to the project. Modules are sorted and incrementally integrated according to the associated risk factor that characterizes the criticality of each module. Both external risks (such as safety) and project risks (such as schedule) can be considered.

critical module

A risk-based approach is particularly appropriate when the development team does not have extensive experience with some aspect of the system under development. Consider once more the Chipmunk Web presence. If Chipmunk has not previously constructed software that interacts directly with shipping services, those interface modules will be critical because of the inherent risks of interacting with externally provided subsystems, which may be inadequately documented or misunderstood and which may also change.

Feature-driven test strategies usually require more complex planning and management than structural strategies. Thus, we adopt them only when their advantages exceed the extra management costs. For small systems a structural strategy is usually sufficient, but for large systems feature-driven strategies are usually preferred. Often large projects require combinations of development strategies that do not fit any single test integration strategies. In these cases, quality managers would combine different strategies: top-down, bottom-up, and sandwich strategies for small subsystems, and a blend of threads and critical module strategies at a higher level.

21.3 Testing Components and Assemblies

Many software products are constructed, partly or wholly, from assemblies of prebuilt software components.[1] A key characteristic of software components is that the organization that develops a component is distinct from the (several) groups of developers who use it to construct systems. The component developers cannot completely anticipate the uses to which a component will be put, and the system developers have limited knowledge of the component. Testing components (by the component developers) and assemblies (by system developers) therefore brings some challenges and constraints that differ from testing other kinds of module.

Reusable components are often more dependable than software developed for a single application. More effort can be invested in improving the quality of a component when the cost is amortized across many applications. Moreover, when reusing a component that has been in use in other applications for some time, one obtains the benefit not only of test and analysis by component developers, but also of actual operational use.

The advantages of component reuse for quality are not automatic. They do not apply to code that was developed for a single application and then scavenged for use in another. The benefit of operational experience as a kind of *in vivo* testing, moreover, is obtained only to the extent that previous uses of the component are quite similar to the new use. These advantages are balanced against two considerable disadvantages. First, a component designed for wide reuse will usually be much more complex than a module designed for a single use; a rule of thumb is that the development effort (including analysis and test) for a widely usable component is at least twice that for a module that provides equivalent functionality for a single application. In addition, a reusable component is by definition developed without full knowledge of the environment in which it will be used, and it is exceptionally difficult to fully and clearly describe all the assumptions, dependencies, and limitations that might impinge upon its use in a particular application.

In general, a software component is characterized by a *contract* or *application program interface* (API) distinct from its implementation. Where a mature market has developed for components addressing a particular need, a single interface specification (e.g., SQL for database access or document object model (DOM) for access and traversal of XML data) can have several distinct implementations. The contract describes the component by specifying access points of the component, such as procedures (methods) and their parameters, possible exceptions, global variables, and input and output network connections. Even when the interface specification is bound to a single implementation, the logical distinction between interface and implementation is crucial to effective use and testing.

The interface specification of a component should provide all the information required for reusing the component, including so-called nonfunctional properties such as performance or capacity limits, in addition to functional behavior. All dependence of the component on the environment in which it executes should also be specified. In

[1] The term component is used loosely and often inconsistently in different contexts. Our working definition and related terms are explained in the sidebar on page 414.

Terminology for Components and Frameworks

Component A software component is a reusable unit of deployment and composition that is deployed and integrated multiple times and usually by different teams. Components are characterized by a *contract* or *interface* and may or may not have state.

Components are often confused with objects, and a component can be encapsulated by an object or a set of objects, but they typically differ in many respects:

- Components typically use persistent storage, while objects usually have only local state.
- Components may be accessed by an extensive set of communication mechanisms, while objects are activated through method calls.
- Components are usually larger grain subsystems than objects.

Component contract or interface The component contract describes the access points and parameters of the component, and specifies functional and nonfunctional behavior and any conditions required for using the component.

Framework A framework is a micro-architecture or a skeleton of an application, with hooks for attaching application-specific functionality or configuration-specific components. A framework can be seen as a circuit board with empty slots for components.

Frameworks and design patterns Patterns are logical design fragments, while frameworks are concrete elements of the application. Frameworks often implement patterns.

Component-based system A component-based system is a system built primarily by assembling software components (and perhaps a small amount of application-specific code) connected through a framework or ad hoc "glue code."

COTS The term commercial off-the-shelf, or COTS, indicates components developed for the sale to other organizations.

practice, few component specifications are complete in every detail, and even details that are specified precisely can easily be overlooked or misunderstood when embedded in a complex specification document.

The main problem facing test designers in the organization that produces a component is lack of information about the ways in which the component will be used. A component may be reused in many different contexts, including applications for which its functionality is an imperfect fit. A general component will typically provide many more features and options than are used by any particular application.

A good deal of functional and structural testing of a component, focused on finding and removing as many program faults as possible, can be oblivious to the context of actual use. As with system and acceptance testing of complete applications, it is then necessary to move to test suites that are more reflective of actual use. Testing with usage scenarios places a higher priority on finding faults most likely to be encountered in use and is needed to gain confidence that the component will be perceived by its users (that is, by developers who employ it as part of larger systems) as sufficiently dependable.

Test designers cannot anticipate all possible uses of a component under test, but they can design test suites for classes of use in the form of scenarios. Test scenarios are closely related to scenarios or use cases in requirements analysis and design.

Sometimes different classes of use are clearly evident in the component specification. For example, the W3 Document Object Model (DOM) specification has parts that deal exclusively with HTML markup and parts that deal with XML; these correspond to different uses to which a component implementing the DOM may be put. The DOM specification further provides two "views" of the component interface. In the flat view, all traversal and inspection operations are provided on node objects, without regard to subclass. In the structured view, each subclass of node offers traversal and inspection operations specific to that variety of node. For example, an Element node has methods to get and set attributes, but a Text node (which represents simple textual data within XML or HTML) does not.

Open Research Issues

Ensuring quality of components and of component-based systems remains a challenging problem and a topic of current research. One research thread considers how dynamic analysis of components and component-based systems in one environment can produce useful information for assessing likely suitability for using some of the same components in another environment (by characterizing the contexts in which a component has been used successfully). A related approach of characterizing a set of behaviors and recognizing changes or differences (whether or not those differences are failures) may be applicable in the increasingly important context of dynamically configurable and field-upgradable systems, which pose all the problems of component-based systems with the additional complication of performing integration in deployed systems rather than in the development environment. For these and other systems, self-monitoring and postdeployment testing in the field are likely to play an increasingly

important role in the future.

Software design for testability is an important factor in the cost and effectiveness of test and analysis, particularly for module and component integration. To some extent model-based testing (Chapter 14) is progress toward producing modules and components with well-specified and testable interfaces, but much remains to be done in characterizing and supporting testability. Design for testability should be an important factor in the evolution of architectural design approaches and notations, including architecture design languages.

Further Reading

The buffer overflow problem in libpng, which caused security vulnerabilities in major Windows, Linux, and Mac OS X Web browsers and e-mail clients, was discovered in 2004 and documented by the United States Computer Emergency Readiness Team (CERT) in Vulnerability Note VU#388984 [Uni04]. The full report on the famous Ariane 5 failure [Lio96] is available from several sources on the Web. The NASA report on loss of the Mars Climate Orbiter [Ste99] is also available on the Web. Leveson [Lev04] describes the role of software in the Ariane failure, loss of the Mars Climate Orbiter, and other spacecraft losses. Weyuker [Wey98] describes challenges of testing component-based systems.

Exercises

21.1. When developing a graphical editor, we used a COTS component for saving and reading files in XML format. During integration testing, the program failed when reading an empty file and when reading a file containing a syntax error.

Try to classify the corresponding faults according to the taxonomy described in Table 21.1.

21.2. The Chipmunk quality team decided to use both thread and critical module integration testing strategies for the Chipmunk Web presence. Envisage at least one situation in which thread integration should be preferred over critical module and one in which critical module testing should be preferred over thread, and motivate the choice.

21.3. Can a backbone testing strategy yield savings in the cost of producing test scaffolding, relative to other structural integration testing strategies? If so, how and under what conditions? If not, why not?

Chapter 22

System, Acceptance, and Regression Testing

System testing can be considered a final step in integration testing, but encompassing systemwide properties against a system specification. Acceptance testing abandons specifications in favor of users, and measures how the final system meets users' expectations. Regression testing checks for faults introduced during evolution.

Required Background

- Chapter 4

  The concepts of dependability, reliability, availability and mean time to failure are important for understanding the difference between system and acceptance testing.

- Chapter 17

  Generating reusable scaffolding and test cases is a foundation for regression testing. Some knowledge about the scaffolding and test case generation problem, though not strictly required, may be useful for understanding regression testing problems.

22.1 Overview

System, acceptance, and regression testing are all concerned with the behavior of a software system as a whole, but they differ in purpose.

System testing is a check of consistency between the software system and its specification (it is a *verification* activity). Like unit and integration testing, system testing is primarily aimed at uncovering faults, but unlike testing activities at finer granularity levels, system testing focuses on system-level properties. System testing together with acceptance testing also serves an important role in assessing whether a product can be

System, Acceptance, and Regression Testing

System test	Acceptance test	Regression test
Checks against requirements specifications	Checks suitability for user needs	Rechecks test cases passed by previous production versions
Performed by development test group	Performed by test group with user involvement	Performed by development test group
Verifies correctness and completion of the product	Validates usefulness and satisfaction with the product	Guards against unintended changes

released to customers, which is distinct from its role in exposing faults to be removed to improve the product.

Flaws in specifications and in development, as well as changes in users' expectations, may result in products that do not fully meet users' needs despite passing system tests. Acceptance testing, as its name implies, is a *validation* activity aimed primarily at the acceptability of the product, and it includes judgments of actual usefulness and usability rather than conformance to a requirements specification.

Regression testing is specialized to the problem of efficiently checking for unintended effects of software changes. New functionality and modification of existing code may introduce unexpected interactions and lead latent faults to produce failures not experienced in previous releases.

22.2 System Testing

The essential characteristics of system testing are that it is comprehensive, based on a specification of observable behavior, and independent of design and implementation decisions. System testing can be considered the culmination of integration testing, and passing all system tests is tantamount to being complete and free of known bugs. The system test suite may share some test cases with test suites used in integration and even unit testing, particularly when a thread-based or spiral model of development has been taken and subsystem correctness has been tested primarily through externally visible features and behavior. However, the essential characteristic of independence implies that test cases developed in close coordination with design and implementation may be unsuitable. The overlap, if any, should result from using system test cases early, rather than reusing unit and integration test cases in the system test suite.

Independence in system testing avoids repeating software design errors in test design. This danger exists to some extent at all stages of development, but always in trade for some advantage in designing effective test cases based on familiarity with the software design and its potential pitfalls. The balance between these considerations shifts at different levels of granularity, and it is essential that independence take priority at some level to obtain a credible assessment of quality.

In some organizations, responsibility for test design and execution shifts at a discrete point from the development team to an independent verification and validation team that is organizationally isolated from developers. More often the shift in empha-

sis is gradual, without a corresponding shift in responsible personnel.

Particularly when system test designers are developers or attached to the development team, the most effective way to ensure that the system test suite is not unduly influenced by design decisions is to design most system test cases as early as possible. Even in agile development processes, in which requirements engineering is tightly interwoven with development, it is considered good practice to design test cases for a new feature before implementing the feature. When the time between specifying a feature and implementing it is longer, early design of system tests facilitates risk-driven strategies that expose critical behaviors to system test cases as early as possible, avoiding unpleasant surprises as deployment nears.

For example, in the (imaginary) Chipmunk development of Web-based purchasing, some questions were raised during requirements specification regarding the point at which a price change becomes effective. For example, if an item's catalog price is raised or lowered between the time it is added to the shopping cart and the time of actual purchase, which price is the customer charged? The requirement was clarified and documented with a set of use cases in which outcomes of various interleavings of customer actions and price changes were specified, and each of these scenarios became a system test case specification. Moreover, since this was recognized as a critical property with many opportunities for failure, the system architecture and build-plan for the Chipmunk Web presence was structured with interfaces that could be artificially driven through various scenarios early in development, and with several of the system test scenarios simulated in earlier integration tests.

The appropriate notions of thoroughness in system testing are with respect to the system specification and potential usage scenarios, rather than code or design. Each feature or specified behavior of the system should be accounted for in one or several test cases. In addition to facilitating design for test, designing system test cases together with the system requirements specification document helps expose ambiguity and refine specifications.

The set of feature tests passed by the current partial implementation is often used as a gauge of progress. Interpreting a count of failing feature-based system tests is discussed in Chapter 20, Section 20.6.

Additional test cases can be devised during development to check for observable symptoms of failures that were not anticipated in the initial system specification. They may also be based on failures observed and reported by actual users, either in acceptance testing or from previous versions of a system. These are in addition to a thorough specification-based test suite, so they do not compromise independence of the quality assessment.

Some system properties, including performance properties like latency between an event and system response and reliability properties like mean time between failures, are inherently global. While one certainly should aim to provide estimates of these properties as early as practical, they are vulnerable to unplanned interactions among parts of a complex system and its environment. The importance of such global properties is therefore magnified in system testing.

Global properties like performance, security, and safety are difficult to specify precisely and operationally, and they depend not only on many parts of the system under test, but also on its environment and use. For example, U.S. HIPAA regulations governing privacy of medical records require appropriate administrative, technical, and physical safeguards to protect the privacy of health information, further specified as follows:

> Implementation specification: safeguards. A covered entity must reasonably safeguard protected health information from any intentional or unintentional use or disclosure that is in violation of the standards, implementation specifications or other requirements of this subpart. [Uni00, sec. 164.530(c)(2)]

It is unlikely that any precise operational specification can fully capture the HIPAA requirement as it applies to an automated medical records system. One must consider the whole context of use, including, for example, which personnel have access to the system and how unauthorized personnel are prevented from gaining access.

Some global properties may be defined operationally, but parameterized by use. For example, a hard-real-time system must meet deadlines, but cannot do so in a completely arbitrary environment; its performance specification is parameterized by event frequency and minimum inter-arrival times. An e-commerce system may be expected to provide a certain level of responsiveness up to a certain number of transactions per second and to degrade gracefully up to a second rate. A key step is identifying the "operational envelope" of the system, and testing both near the edges of that envelope (to assess compliance with specified goals) and well beyond it (to ensure the system degrades or fails gracefully). Defining borderline and extreme cases is logically part of requirements engineering, but as with precise specification of features, test design often reveals gaps and ambiguities.

Not all global properties will be amenable to dynamic testing at all, at least in the conventional sense. One may specify a number of properties that a secure computer system should have, and some of these may be amenable to testing. Others can be addressed only through inspection and analysis techniques, and ultimately one does not trust the security of a system at least until an adversarial team has tried and failed to subvert it. Similarly, there is no set of test cases that can establish software safety, in part because safety is a property of a larger system and environment of which the software is only part. Rather, one must consider the safety of the overall system, and assess aspects of the software that are critical to that overall assessment. Some but not all of those claims may be amenable to testing.

Testing global system properties may require extensive simulation of the execution environment. Creating accurate models of the operational environment requires substantial human resources, and executing them can require substantial time and machine resources. Usually this implies that "stress" testing is a separate activity from frequent repetition of feature tests. For example, a large suite of system test cases might well run each night or several times a week, but a substantial stress test to measure robust performance under heavy load might take hours to set up and days or weeks to run.

A test case that can be run automatically with few human or machine resources should generally focus on one purpose: to make diagnosis of failed test executions as

Unit, Integration, and System Testing

	Unit Test	Integration Test	System Test
Test cases derived from	module specifications	architecture and design specifications	requirements specification
Visibility required	all the details of the code	some details of the code, mainly interfaces	no details of the code
Scaffolding required	Potentially complex, to simulate the activation environment (drivers), the modules called by the module under test (stubs) and test oracles	Depends on architecture and integration order. Modules and subsystems can be incrementally integrated to reduce need for drivers and stubs.	Mostly limited to test oracles, since the whole system should not require additional drivers or stubs to be executed. Sometimes includes a simulated execution environment (e.g., for embedded systems).
Focus on	behavior of individual modules	module integration and interaction	system functionality

clear and simple as possible. Stress testing alters this: If a test case takes an hour to set up and a day to run, then one had best glean as much information as possible from its results. This includes monitoring for faults that should, in principle, have been found and eliminated in unit and integration testing, but which become easier to recognize in a stress test (and which, for the same reason, are likely to become visible to users). For example, several embedded system products ranging from laser printers to tablet computers have been shipped with slow memory leaks that became noticeable only after hours or days of continuous use. In the case of the tablet PC whose character recognition module gradually consumed all system memory, one must wonder about the extent of stress testing the software was subjected to.

22.3 Acceptance Testing

The purpose of acceptance testing is to guide a decision as to whether the product in its current state should be released. The decision can be based on measures of the product or process. Measures of the product are typically some inference of dependability based on statistical testing. Measures of the process are ultimately based on comparison to experience with previous products.

Although system and acceptance testing are closely tied in many organizations, fundamental differences exist between searching for faults and measuring quality. Even when the two activities overlap to some extent, it is essential to be clear about the

distinction, in order to avoid drawing unjustified conclusions.

Quantitative goals for dependability, including reliability, availability, and mean time between failures, were introduced in Chapter 4. These are essentially statistical measures and depend on a statistically valid approach to drawing a representative sample of test executions from a population of program behaviors. Systematic testing, which includes all of the testing techniques presented heretofore in this book, does not draw statistically representative samples. Their purpose is not to fail at a "typical" rate, but to exhibit as many failures as possible. They are thus unsuitable for statistical testing.

The first requirement for valid statistical testing is a precise definition of what is being measured and for what population. If system operation involves transactions, each of which consists of several operations, a failure rate of one operation in a thousand is quite different from a failure rate of one transaction in a thousand. In addition, the failure rate may vary depending on the mix of transaction types, or the failure rate may be higher when one million transactions occur in an hour than when the same transactions are spread across a day. Statistical modeling therefore necessarily involves construction of a model of usage, and the results are relative to that model.

Suppose, for example, that a typical session using the Chipmunk Web sales facility consists of 50 interactions, the last of which is a single operation in which the credit card is charged and the order recorded. Suppose the Chipmunk software always operates flawlessly up to the point that a credit card is to be charged, but on half the attempts it charges the wrong amount. What is the reliability of the system? If we count the fraction of individual interactions that are correctly carried out, we conclude that only one operation in 100 fails, so the system is 99% reliable. If we instead count entire sessions, then it is only 50% reliable, since half the sessions result in an improper credit card charge.

operational profile

Statistical models of usage, or *operational profiles*, may be available from measurement of actual use of prior, similar systems. For example, use of a current telephone handset may be a reasonably good model of how a new handset will be used. Good models may also be obtained in embedded systems whose environment is primarily made up of predictable devices rather than unpredictable humans. In other cases one cannot justify high confidence in a model, but one can limit the uncertainty to a small number of parameters. One can perform sensitivity testing to determine which parameters are critical. Sensitivity testing consists of repeating statistical tests while systematically varying parameters to note the effect of each parameter on the output. A particular parameter may have little effect on outcomes over the entire range of plausible values, or there may be an effect that varies smoothly over the range. If the effect of a given parameter is either large or varies discontinuously (e.g., performance falls precipitously when system load crosses some threshold), then one may need to make distinct predictions for different value ranges.

sensitivity testing

A second problem faced by statistical testing, particularly for reliability, is that it may take a very great deal of testing to obtain evidence of a sufficient level of reliability. Consider that a system that executes once per second, with a failure rate of one execution in a million, or 99.9999% reliability, fails about 31 times each year; this may require a great testing effort and still not be adequate if each failure could result in death or a lawsuit. For critical systems, one may insist on software failure rates that are

an insignificant fraction of total failures. For many other systems, statistical measures of reliability may simply not be worth the trouble.

A less formal, but frequently used approach to acceptance testing is testing with users. An early version of the product is delivered to a sample of users who provide feedback on failures and usability. Such tests are often called *alpha* and *beta* tests. The two terms distinguish between testing phases. Often the early or alpha phases are performed within the developing organization, while the later or beta phases are performed at users' sites.

alpha and beta test

In alpha and beta testing, the user sample determines the operational profile. A good sample of users should include representatives of each distinct category of users, grouped by operational profile and significance. Suppose, for example, Chipmunk plans to provide Web-based sales facilities to dealers, industrial customers, and individuals. A good sample should include both users from each of those three categories and a range of usage in each category. In the industrial user category, large customers who frequently issue complex orders as well as small companies who typically order a small number of units should be represented, as the difference in their usage may lead to different failure rates. We may weigh differently the frequency of failure reports from dealers and from direct customers, to reflect either the expected mix of usage in the full population or the difference in consequence of failure.

22.4 Usability

A usable product is quickly learned, allows users to work efficiently, and is pleasant to use. Usability involves objective criteria such as the time and number of operations required to perform tasks and the frequency of user error, in addition to the overall, subjective satisfaction of users.

For test and analysis, it is useful to distinguish attributes that are uniquely associated with usability from other aspects of software quality (dependability, performance, security, etc.). Other software qualities may be necessary for usability; for example, a program that often fails to satisfy its functional requirements or that presents security holes is likely to suffer poor usability as a consequence. Distinguishing primary usability properties from other software qualities allows responsibility for each class of properties to be allocated to the most appropriate personnel, at the most cost-effective points in the project schedule.

Even if usability is largely based on user perception and thus is validated based on user feedback, it can be verified early in the design and through the whole software life cycle. The process of verifying and validating usability includes the following main steps:

Inspecting specifications with usability checklists. Inspection provides early feedback on usability.

Testing early prototypes with end users to explore their mental model (*exploratory test*), evaluate alternatives (*comparison test*), and validate software usability. A prototype for early assessment of usability may not include any functioning soft-

ware; a *cardboard prototype* may be as simple as a sequence of static images presented to users by the usability tester.

Testing incremental releases with both usability experts and end users to monitor progress and anticipate usability problems.

System and acceptance testing that includes expert-based inspection and testing, user-based testing, comparison testing against competitors, and analysis and checks often done automatically, such as a check of link connectivity and verification of browser compatibility.

User-based testing (i.e., testing with representatives of the actual end-user population) is particularly important for validating software usability. It can be applied at different stages, from early prototyping through incremental releases of the final system, and can be used with different goals: exploring the mental model of the user, evaluating design alternatives, and validating against established usability requirements and standards.

exploratory testing

The purpose of *exploratory testing* is to investigate the mental model of end users. It consists of asking users about their approach to interactions with the system. For example, during an exploratory test for the Chipmunk Web presence, we may provide users with a generic interface for choosing the model they would like to buy, in order to understand how users will interact with the system. A generic interface could present information about all laptop computer characteristics uniformly to see which are examined first by the sample users, and thereby to determine the set of characteristics that should belong to the summary in the menu list of laptops. Exploratory test is usually performed early in design, especially when designing a system for a new target population.

The purpose of *comparison testing* is evaluating options. It consists of observing user reactions to alternative interaction patterns. During comparison test we can, for example, provide users with different facilities to assemble the desired Chipmunk laptop configuration, and to identify patterns that facilitate users' interactions. Comparison test is usually applied when the general interaction patterns are clear and need to be refined. It can substitute for exploratory testing if initial knowledge about target users is sufficient to construct a range of alternatives, or otherwise follows exploratory testing.

The purpose of *validation testing* is assessing overall usability. It includes identifying difficulties and obstacles that users encounter while interacting with the system, as well as measuring characteristics such as error rate and time to perform a task.

A well-executed design and organization of usability testing can produce results that are objective and accurately predict usability in the target user population. The usability test design includes selecting suitable representatives of the target users and organizing sessions that guide the test toward interpretable results. A common approach is divided into preparation, execution, and analysis phases. During the preparation phase, test designers define the objectives of the session, identify the items to be tested, select a representative population of end users, and plan the required actions. During execution, users are monitored as they execute the planned actions in a controlled envi-

ronment. During analysis, results are evaluated, and changes to the software interfaces or new testing sessions are planned, if required.

Each phase must be carefully executed to ensure success of the testing session. User time is a valuable and limited resource. Well-focused test objectives should not be too narrow, to avoid useless waste of resources, nor too wide, to avoid scattering resources without obtaining useful data. Focusing on specific interactions is usually more effective than attempting to assess the usability of a whole program at once. For example, the Chipmunk usability test team independently assesses interactions for catalog browsing, order definition and purchase, and repair service.

The larger the population sample, the more precise the results, but the cost of very large samples is prohibitive; selecting a small but representative sample is therefore critical. A good practice is to identify homogeneous classes of users and select a set of representatives from each class. Classes of users depend on the kind of application to be tested and may be categorized by role, social characteristics, age, and so on. A typical compromise between cost and accuracy for a well-designed test session is five users from a unique class of homogeneous users, four users from each of two classes, or three users for each of three or more classes. Questionnaires should be prepared for the selected users to verify their membership in their respective classes. Some approaches also assign a weight to each class, according to their importance to the business. For example, Chipmunk can identify three main classes of users: individual, business, and education customers. Each of the main classes is further divided. Individual customers are distinguished by education level; business customers by role; and academic customers by size of the institution. Altogether, six putatively homogeneous classes are obtained: Individual customers with and without at least a bachelor degree, managers and staff of commercial customers, and customers at small and large education institutions.

Users are asked to execute a planned set of actions that are identified as typical uses of the tested feature. For example, the Chipmunk usability assessment team may ask users to configure a product, modify the configuration to take advantage of some special offers, and place an order with overnight delivery.

Users should perform tasks independently, without help or influence from the testing staff. User actions are recorded, and comments and impressions are collected with a post-activity questionnaire. Activity monitoring can be very simple, such as recording sequences of mouse clicks to perform each action. More sophisticated monitoring can include recording mouse or eye movements. Timing should also be recorded and may sometimes be used for driving the sessions (e.g., fixing a maximum time for the session or for each set of actions).

An important aspect of usability is accessibility to all users, including those with disabilities. Accessibility testing is legally required in some application domains. For example, some governments impose specific accessibility rules for Web applications of public institutions. The set of Web Content Accessibility Guidelines (WCAG) defined by the World Wide Web Consortium are becoming an important standard reference. The WCAG guidelines are summarized in the sidebar on page 426.

Web Content Accessibility Guidelines (WCAG)[a]

1. Provide equivalent alternatives to auditory and visual content that convey essentially the same function or purpose.
2. Ensure that text and graphics are understandable when viewed without color.
3. Mark up documents with the proper structural elements, controlling presentation with style sheets rather than presentation elements and attributes.
4. Use markup that facilitates pronunciation or interpretation of abbreviated or foreign text.
5. Ensure that tables have necessary markup to be transformed by accessible browsers and other user agents.
6. Ensure that pages are accessible even when newer technologies are not supported or are turned off.
7. Ensure that moving, blinking, scrolling, or auto-updating objects or pages may be paused or stopped.
8. Ensure that the user interface, including embedded user interface elements, follows principles of accessible design: device-independent access to functionality, keyboard operability, self-voicing, and so on.
9. Use features that enable activation of page elements via a variety of input devices.
10. Use interim accessibility so that assisting technologies and older browsers will operate correctly.
11. Where technologies outside of W3C specifications is used (e.g, Flash), provide alternative versions to ensure accessibility to standard user agents and assistive technologies (e.g., screen readers).
12. Provide context and orientation information to help users understand complex pages or elements.
13. Provide clear and consistent navigation mechanisms to increase the likelihood that a person will find what they are looking for at a site.
14. Ensure that documents are clear and simple, so they may be more easily understood.

[a]Excerpted and adapted from *Web Content Accessibility Guidelines 1.0*, W3C Recommendation 5-May 1999; used by permission. The current version is distributed by W3C at `http://www.w3.org/TR/WAI-WEBCONTENT`.

22.5 Regression Testing

When building a new version of a system (e.g., by removing faults, changing or adding functionality, porting the system to a new platform, or extending interoperability), we may also change existing functionality in unintended ways. Sometimes even small changes can produce unforeseen effects that lead to new failures. For example, a guard added to an array to fix an overflow problem may cause a failure when the array is used in other contexts, or porting the software to a new platform may expose a latent fault in creating and modifying temporary files.

When a new version of software no longer correctly provides functionality that should be preserved, we say that the new version *regresses* with respect to former versions. The *nonregression* of new versions (i.e., preservation of functionality), is a basic quality requirement. Disciplined design and development techniques, including precise specification and modularity that encapsulates independent design decisions, improves the likelihood of achieving nonregression. Testing activities that focus on regression problems are called *(non) regression testing*. Usually "non" is omitted and we commonly say *regression testing*.

retest all

A simple approach to regression testing consists of reexecuting all test cases designed for previous versions. Even this simple *retest all* approach may present nontrivial problems and costs. Former test cases may not be reexecutable on the new version without modification, and rerunning all test cases may be too expensive and unnecessary. A good quality test suite must be maintained across system versions.

test case maintenance

Changes in the new software version may impact the format of inputs and outputs, and test cases may not be executable without corresponding changes. Even simple modifications of the data structures, such as the addition of a field or small change of data types, may invalidate former test cases, or outputs comparable with the new ones. Moreover, some test cases may be *obsolete*, since they test features of the software that have been modified, substituted, or removed from the new version.

Scaffolding that interprets test case specifications, rather than fully concrete test data, can reduce the impact of input and output format changes on regression testing, as discussed in Chapter 17. Test case specifications and oracles that capture essential correctness properties, abstracting from arbitrary details of behavior, likewise reduce the likelihood that a large portion of a regression test suite will be invalidated by a minor change.

High-quality test suites can be maintained across versions by identifying and removing obsolete test cases, and by revealing and suitably marking redundant test cases. Redundant cases differ from obsolete, being executable but not important with respect to the considered testing criteria. For example, test cases that cover the same path are mutually redundant with respect to structural criteria, while test cases that match the same partition are mutually redundant with respect to functional criteria. Redundant test cases may be introduced in the test suites due to concurrent work of different test designers or to changes in the code. Redundant test cases do not reduce the overall effectiveness of tests, but impact on the cost-benefits trade-off: They are unlikely to reveal faults, but they augment the costs of test execution and maintenance. Obsolete test cases are removed because they are no longer useful, while redundant test cases are kept because they may become helpful in successive versions of the software.

Good test documentation is particularly important. As we will see in Chapter 24, test specifications define the features to be tested, the corresponding test cases, the inputs and expected outputs, as well as the execution conditions for all cases, while reporting documents indicate the results of the test executions, the open faults, and their relation to the test cases. This information is essential for tracking faults and for identifying test cases to be reexecuted after fault removal.

22.6 Regression Test Selection Techniques

Even when we can identify and eliminate obsolete test cases, the number of tests to be reexecuted may be large, especially for legacy software. Executing all test cases for large software products may require many hours or days of execution and may depend on scarce resources such as an expensive hardware test harness. For example, some mass market software systems must be tested for compatibility with hundreds of different hardware configurations and thousands of drivers. Many test cases may have been designed to exercise parts of the software that cannot be affected by the changes in the version under test. Test cases designed to check the behavior of the file management system of an operating system is unlikely to provide useful information when reexecuted after changes of the window manager. The cost of reexecuting a test suite can be reduced by selecting a subset of test cases to be reexecuted, omitting irrelevant test cases or prioritizing execution of subsets of the test suite by their relation to changes.

Test case prioritization orders frequency of test case execution, executing all of them eventually but reducing the frequency of those deemed least likely to reveal faults by some criterion. Alternate execution is a variant on prioritization for environments with frequent releases and small incremental changes; it selects a subset of regression test cases for each software version. Prioritization can be based on the specification and code-based regression test selection techniques described later in this chapter. In addition, test histories and fault-proneness models can be incorporated in prioritization schemes. For example, a test case that has previously revealed a fault in a module that has recently undergone change would receive a very high priority, while a test case that has never failed (yet) would receive a lower priority, particularly if it primarily concerns a feature that was not the focus of recent changes.

Regression test selection techniques are based on either code or specifications. Code-based selection techniques select a test case for execution if it exercises a portion of the code that has been modified. Specification-based criteria select a test case for execution if it is relevant to a portion of the specification that has been changed. Code-based regression test techniques can be supported by relatively simple tools. They work even when specifications are not properly maintained. However, like code-based test techniques in general, they do not scale well from unit testing to integration and system testing. In contrast, specification-based criteria scale well and are easier to apply to changes that cut across several modules. However, they are more challenging to automate and require carefully structured and well-maintained specifications.

Among code-based test selection techniques, control-based techniques rely on a record of program elements executed by each test case, which may be gathered from

an instrumented version of the program. The structure of the new and old versions of the program are compared, and test cases that exercise added, modified, or deleted elements are selected for reexecution. Different criteria are obtained depending on the program model on which the version comparison is based (e.g., control flow or data flow graph models).

control flow graph (CFG) regression test

Control flow graph (CFG) regression techniques are based on the differences between the CFGs of the new and old versions of the software. Let us consider, for example, the C function cgi_decode from Chapter 12. Figure 22.1 shows the original function as presented in Chapter 12, while Figure 22.2 shows a revison of the program. We refer to these two versions as 1.0 and 2.0, respectively. Version 2.0 adds code to fix a fault in interpreting hexadecimal sequences `'%xy'`. The fault was revealed by testing version 1.0 with input terminated by an erroneous subsequence `'%x'`, causing version 1.0 to read past the end of the input buffer and possibly overflow the output buffer. Version 2.0 contains a new branch to map the unterminated sequence to a question mark.

Let us consider all structural test cases derived for cgi_decode in Chapter 12, and assume we have recorded the paths exercised by the different test cases as shown in Figure 22.3. Recording paths executed by test cases can be done automatically with modest space and time overhead, since what must be captured is only the set of program elements exercised rather than the full history.

CFG regression testing techniques compare the annotated control flow graphs of the two program versions to identify a subset of test cases that traverse modified parts of the graphs. The graph nodes are annotated with corresponding program statements, so that comparison of the annotated CFGs detects not only new or missing nodes and arcs, but also nodes whose changed annotations correspond to small, but possibly relevant, changes in statements.

The CFG for version 2.0 of cgi_decode is given in Figure 22.4. Differences between version 2.0 and 1.0 are indicated in gray. In the example, we have new nodes, arcs and paths. In general, some nodes or arcs may be missing (e.g., when part of the program is removed in the new version), and some other nodes may differ only in the annotations (e.g., when we modify a condition in the new version).

CFG criteria select all test cases that exercise paths through changed portions of the CFG, including CFG structure changes and node annotations. In the example, we would select all test cases that pass through node D and proceed toward node G and all test cases that reach node L, that is, all test cases except $TC1$. In this example, the criterion is not very effective in reducing the size of the test suite because modified statements affect almost all paths.

If we consider only the corrective modification (nodes X and Y), the criterion is more effective. The modification affects only the paths that traverse the edge between D and G, so the CFG regression testing criterion would select only test cases traversing those nodes (i.e., $TC2$, $TC3$, $TC4$, $TC5$, $TC8$ and $TC9$). In this case the size of the test suite to be reexecuted includes two-thirds of the test cases of the original test suite.

In general, the CFG regression testing criterion is effective only when the changes affect a relatively small subset of the paths of the original program, as in the latter case. It becomes almost useless when the changes affect most paths, as in version 2.0.

Data flow (DF) regression testing techniques select test cases for new and modi-

```
1  #include "hex_values.h"
2  /**  Translate a string from the CGI encoding to plain ascii text.
3  *     '+' becomes space, %xx becomes byte with hex value xx,
4  *     other alphanumeric characters map to themselves.
5  *    Returns 0 for success, positive for erroneous input
6  *          1 = bad hexadecimal digit
7  */
8  int  cgi_decode(char *encoded, char *decoded) {
9     char *eptr = encoded;
10    char *dptr = decoded;
11    int ok=0;
12    while (*eptr) {
13       char c;
14       c = *eptr;
15       if (c == '+') {          /* Case 1:  '+' maps to blank */
16          *dptr = ' ';
17       } else if (c == '%') { /* Case 2:  '%xx' is hex for character xx */
18          int   digit_high = Hex_Values[*(++eptr)]; /* note illegal => -1 */
19          int   digit_low  = Hex_Values[*(++eptr)];
20          if ( digit_high == -1 || digit_low == -1 ) {
21             /* *dptr='?'; */
22             ok=1; /* Bad return code */
23          } else {
24             *dptr = 16* digit_high + digit_low;
25          }
26       } else {   /* Case 3: Other characters map to themselves */
27          *dptr = *eptr;
28       }
29       ++dptr;
30       ++eptr;
31    }
32    *dptr = '\0';                    /* Null terminator for string */
33    return ok;
34 }
```

Figure 22.1: C function cgi_decode *version 1.0. The C function* cgi_decode *translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface of most Web servers. Repeated from Figure 12.1 in Chapter 12.*

```
1  #include "hex_values.h"
2  /**   Translate a string from the CGI encoding to plain ascii text.
3  *     '+' becomes space, %xx becomes byte with hex value xx,
4  *     other alphanumeric characters map to themselves, illegal to '?'.
5  *    Returns 0 for success, positive for erroneous input
6  *          1 = bad hex digit, non-ascii char, or premature end.
7  */
8  int  cgi_decode(char *encoded, char *decoded) {
9     char *eptr = encoded;
10    char *dptr = decoded;
11    int ok=0;
12    while (*eptr) {
13       char c;
14       c = *eptr;
15       if (c == '+') {          /* Case 1:  '+' maps to blank */
16          *dptr = ' ';
17       } else if (c == '%') { /* Case 2:  '%xx' is hex for character xx */
18          if (! ( *(eptr + 1)   && *(eptr + 2) )) {   /* \%xx must precede EOL */
19             ok = 1; return;
20          }
21          /* OK, we know the xx are there, now decode them */
22          int   digit_high = Hex_Values[*(++eptr)]; /* note illegal => -1 */
23          int   digit_low  = Hex_Values[*(++eptr)];
24          if (    digit_high == -1 || digit_low == -1 ) {
25             /* *dptr='?'; */
26             ok=1; /* Bad return code */
27          } else {
28             *dptr = 16* digit_high + digit_low;
29          }
30       } else {   /* Case 3: Other characters map to themselves */
31          *dptr = *eptr;
32       }
33       if (! isascii(*dptr)) { /* Produce only legal ascii */
34          *dptr = '?';
35          ok = 1;
36       }
37       ++dptr;
38       ++eptr;
39    }
40    *dptr = '\0';       /* Null terminator for string */
41    return ok;
42 }
```

Figure 22.2: Version 2.0 of the C function cgi_decode *adds a control on hexadecimal escape sequences to reveal incorrect escape sequences at the end of the input string and a new branch to deal with non-ASCII characters.*

Id	Test case	Path
TC1	" "	A B M
TC2	"test+case%1Dadequacy"	A B C D F L ... B M
TC3	"adequate+test%0Dexecution%7U"	A B C D F L ... B M
TC4	"%3D"	A B C D G H L B M
TC5	"%A"	A B C D G I L B M
TC6	"a+b"	A B C D F L B C E L B C D F L B M
TC7	"test"	A B C D F L B C D F L B C D F L B C D F L B M
TC8	"+%0D+%4J"	A B C E L B C D G I L ... B M
TC9	"first+test%9Ktest%K9"	A B C D F L ... B M

Figure 22.3: Paths covered by the structural test cases derived for version 1.0 of function cgi_decode. Paths are given referring to the nodes of the control flow graph of Figure 22.4.

data flow (DF) regression test

fied pairs of definitions with uses (DU pairs, cf. Sections 6.1, page 77 and 13.2, page 236). DF regression selection techniques reexecute test cases that, when executed on the original program, exercise DU pairs that were deleted or modified in the revised program. Test cases that executed a conditional statement whose predicate was altered are also selected, since the changed predicate could alter some old definition-use associations. Figure 22.5 shows the new definitions and uses introduced by modifications to cgi_decode.[1] These new definitions and uses introduce new DU pairs and remove others.

In contrast to code-based techniques, specification-based test selection techniques do not require recording the control flow paths executed by tests. Regression test cases can be identified from correspondence between test cases and specification items. For example, when using category partition, test cases correspond to sets of choices, while in finite state machine model-based approaches, test cases cover states and transitions. Where test case specifications and test data are generated automatically from a specification or model, generation can simply be repeated each time the specification or model changes.

Code-based regression test selection criteria can be adapted for model-based regression test selection. Consider, for example, the control flow graph derived from the *process shipping order* specification in Chapter 14. We add the following item to that specification:

Restricted countries: A set of restricted destination countries is maintained, based on current trade restrictions. If the shipping address contains a restricted destination country, only credit card payments are accepted for that order, and shipping

[1] When dealing with arrays, we follow the criteria discussed in Chapter 13: A change of an array value is a definition of the array and a use of the index. A use of an array value is a use of both the array and the index.

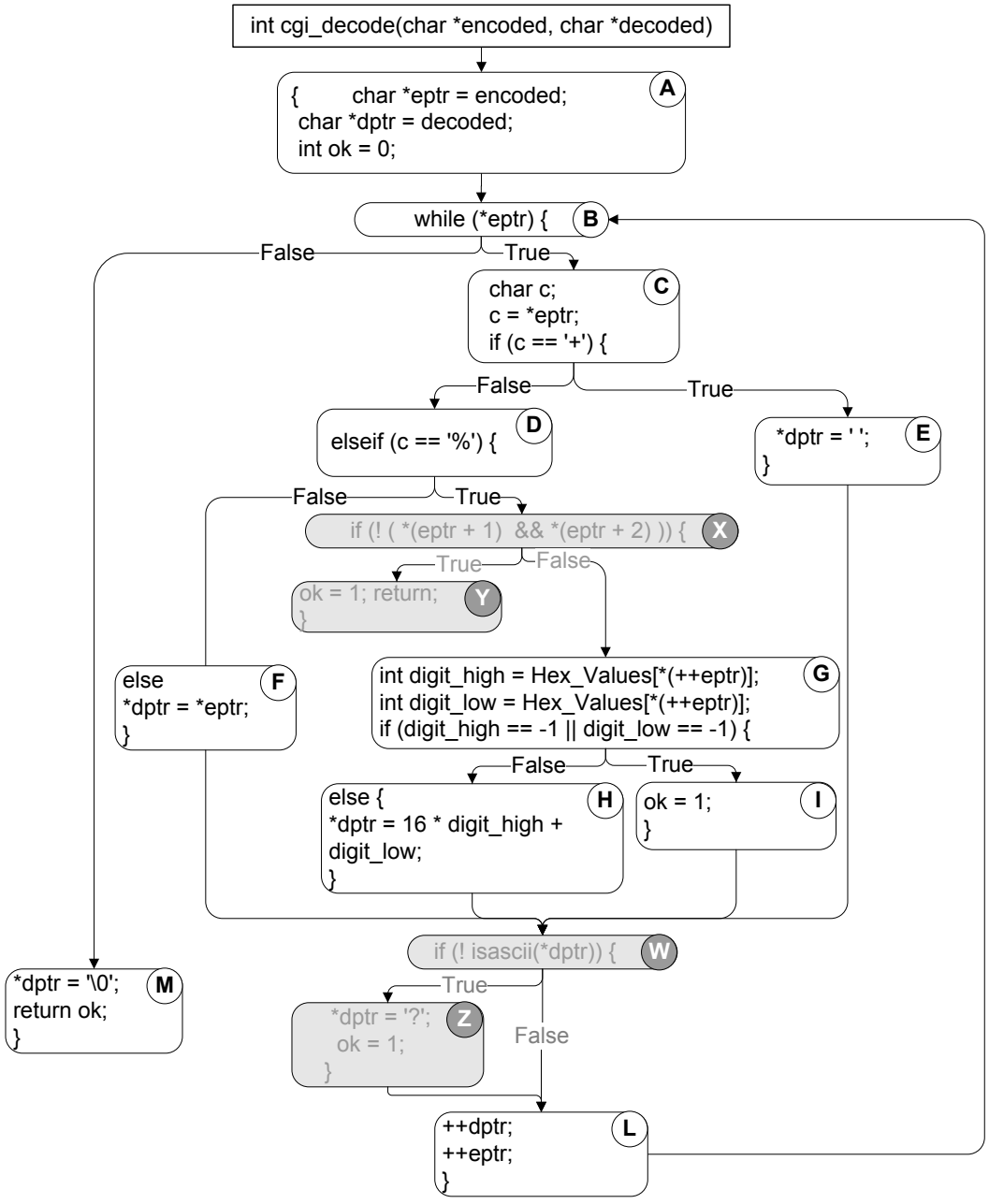

Figure 22.4: The control flow graph of function cgi_decode *version 2.0. Gray background indicates the changes from the former version.*

Variable	Definitions	Uses
*eptr		X
eptr		X
dptr	Z	W
dptr		Z W
ok	Y Z	

Figure 22.5: Definitions and uses introduced by changes in cgi_decode. *Labels refer to the nodes in the control flow graph of Figure 22.4.*

proceeds only after approval by a designated company officer responsible for checking that the goods ordered may be legally exported to that country.

The new requirement can be added to the flow graph model of the specification as illustrated in Figure 22.6.

We can identify regression test cases with the CFG criterion that selects all cases that correspond to international shipping addresses (i.e., test cases *TC-1* and *TC-5* from the following table). The table corresponds to the functional test cases derived using to the method described in Chapter 14 on page 259.

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	–	–	–	–
TC-3	Yes	–	–	–	–	–	–
TC-4	No	Dom	Air	–	–	–	–
TC-5	No	Int	Land	–	–	–	–
TC-6	No	–	–	Edu	Inv	–	–
TC-7	No	–	–	–	CC	Yes	–
TC-8	No	–	–	–	CC	–	No (abort)
TC-9	No	–	–	–	CC	–	No (no abort)

Models derived for testing can be used not only for selecting regression test cases, but also for generating test cases for the new code. In the preceding example, we can use the model not only to identify the test cases that should be reused, but also to generate new test cases for the new functionality, following the combinatorial approaches described in Chapter 11.

22.7 Test Case Prioritization and Selective Execution

Regression testing criteria may select a large portion of a test suite. When a regression test suite is too large, we must further reduce the set of test cases to be executed.

Random sampling is a simple way to reduce the size of the regression test suite. Better approaches prioritize test cases to reflect their predicted usefulness. In a con-

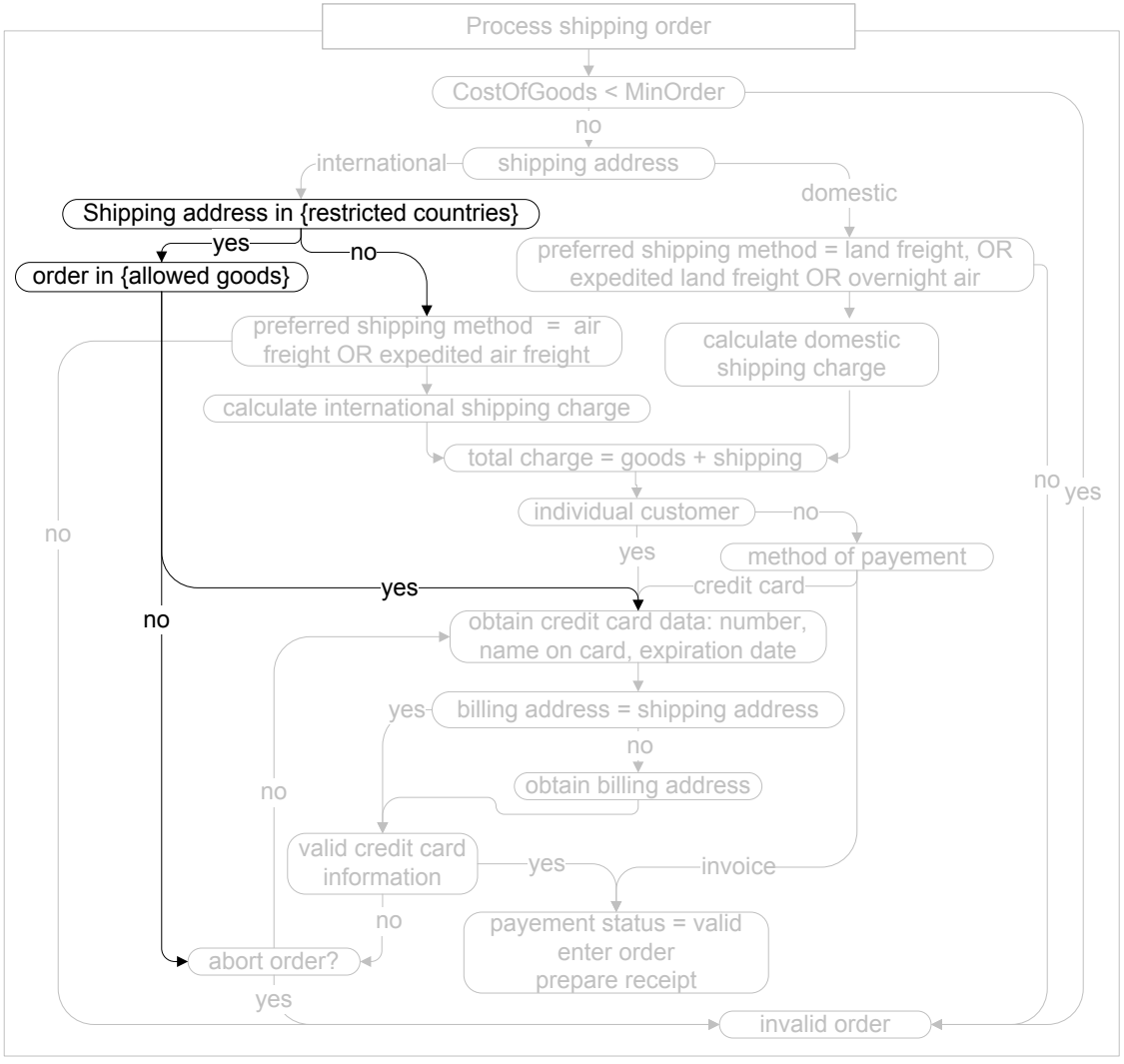

Figure 22.6: A flow graph model of the specification of the shipping order functionality presented in Chapter 14, augmented with the "restricted country" requirement. The changes in the flow graph are indicated in black.

tinuous cycle of retesting as the product evolves, high-priority test cases are selected more often than low-priority test cases. With a good selection strategy, all test cases are executed sooner or later, but the varying periods result in an efficient rotation in which the cases most likely to reveal faults are executed most frequently.

Priorities can be assigned in many ways. A simple priority scheme assigns priority according to the execution history: Recently executed test cases are given low priority, while test cases that have not been recently executed are given high priority. In the extreme, heavily weighting execution history approximates round robin selection.

execution history priority schema

Other history-based priority schemes predict fault detection effectiveness. Test cases that have revealed faults in recent versions are given high priority. Faults are not evenly distributed, but tend to accumulate in particular parts of the code or around particular functionality. Test cases that exercised faulty parts of the program in the past often exercise faulty portions of subsequent revisions.

fault revealing priority schema

Structural coverage leads to a set of priority schemes based on the elements covered by a test case. We can give high priority to test cases that exercise elements that have not recently been exercised. Both the number of elements covered and the "age" of each element (time since that element was covered by a test case) can contribute to the prioritization.

structural priority schema

Structural priority schemes produce several criteria depending on which elements we consider: statements, conditions, decisions, functions, files, and so on. The choice of the element of interest is usually driven by the testing level. Fine-grain elements such as statements and conditions are typically used in unit testing, while in integration or system testing one can consider coarser grain elements such as methods, features, and files.

Open Research Issues

System requirements include many nonfunctional behavioral properties. While there is an active research community in reliability testing, in general, assessment of nonfunctional properties is not as well-studied as testing for correctness. Moreover, as trends in software develop, new problems for test and analysis are following the emphasis on particular nonfunctional properties. A prominent example of this over the last several years, and with much left to do, is test and analysis to assess and improve security.

Selective regression test selection based on analysis of source code is now well-studied. There remains need and opportunity for improvement in techniques that give up the safety guarantee (selecting all test cases that *might* be affected by a software change) to obtain more significant test suite reductions. Specification-based regression test selection is a promising avenue of research, particularly as more systems incorporate components without full source code.

Increasingly ubiquitous network access is blurring the once-clear lines between alpha and beta testing and opening possibilities for gathering much more information from execution of deployed software. We expect to see advances in approaches to gathering information (both from failures and from normal execution) as well as exploiting potentially large amounts of gathered information. Privacy and confidentiality are an important research challenge in postdeployment monitoring.

Further Reading

Musa [Mus04] is a guide to reliability engineering from a pioneer in the field; ongoing research appears in the International Symposium on Software Reliability Engineering (ISSRE) conference series. Graves et al. [GHK+98] and Rothermel and Harrold [RH97] provide useful overviews of selective regression testing. Kim and Porter [KP02] describe history-based test prioritization. Barnum [Bar01] is a well-regarded text on usability testing; Nielsen [Nie00] is a broader popular introduction to usability engineering, with a chapter on usability testing.

Exercises

22.1. Consider the Chipmunk Computer Web presence. Define at least one test case that may serve both during final integration and early system testing, at least one that serves only as an integration test case, and at least one that is more suitable as a system test case than as a final integration test case. Explain your choices.

22.2. When and why should testing responsibilities shift from the development team to an independent quality team? In what circumstances might using an independent quality team be impractical?

22.3. Identify some kinds of properties that cannot be efficiently verified with system testing, and indicate how you would verify them.

22.4. Provide two or more examples of resource limitations that may impact system test more than module and integration test. Explain the difference in impact.

22.5. Consider the following required property of the Chipmunk Computer Web presence:

> Customers should perceive that purchasing a computer using the Chipmunk Web presence is at least as convenient, fast, and intuitive as purchasing a computer in an off-line retail store.

Would you check it as part of system or acceptance testing? Reformulate the property to allow test designers to check it in a different testing phase (system testing, if you consider the property checkable as part of acceptance testing, or vice versa).