Advanced Software Testing

Korbin Bowden

First Edition, 2012

ISBN 978-81-323-1693-0

Published by:
Learning Press
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Software Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

1. meets the business and technical requirements that guided its design and development;
2. works as expected; and
3. can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

Overview

Testing can never completely identify all the defects within software. Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected

purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. **Software testing** is the process of attempting to make this assessment.

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug") it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented
- 1957–1978 - Demonstration oriented
- 1979–1982 - Destruction oriented
- 1983–1987 - Evaluation oriented
- 1988–2000 - Prevention oriented

Software testing topics

Scope

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Functional vs non-functional testing

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once".

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements, that result in errors of omission by the program designer. A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software. A single defect may result in a wide range of failure symptoms.

Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

Cost to fix a defect		Time detected				
		Requirements	**Architecture**	**Construction**	**System test**	**Post-release**
Time introduced	**Requirements**	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Compatibility

A common cause of software failure (real or perceived) is a lack of compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

Input combinations and preconditions

A very fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product. This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*)—usability, scalability, performance, compatibility, reliability—can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

Software verification and validation

Software testing is used in association with verification and validation:

- Verification: Have we built the software right? (i.e., does it match the specification).
- Validation: Have we built the right software? (i.e., is this what the customer wants).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

> Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
> Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The software testing team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing, different roles have been established: *manager*, *test lead*, *test designer*, *tester*, *automation developer*, and *test administrator*.

Software quality assurance (SQA)

Though controversial, software testing may be viewed as an important part of the software quality assurance (SQA) process. In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called *defect rate.*

What constitutes an "acceptable defect rate" depends on the nature of the software; A flight simulator video game would have much higher defect tolerance than software for an actual airplane.

Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

Testing methods

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White box testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - testing of the application using public and private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths
- Mutation testing methods
- Static testing - White box testing includes all static testing

Test coverage

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

Black box testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.

Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. On the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion", on the one hand, and the disadvantage of "blind exploring", on the other.

Grey box testing

Grey box testing (American spelling: **gray box testing**) involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

These type of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

Unit testing is also called *component testing*.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System testing

System testing tests a completely integrated system to verify that it meets its requirements.

System integration testing

System integration testing verifies that a system is integrated to any external or third-party systems defined in the system requirements.

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Non-functional testing

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

Software performance testing and load testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*.

Volume testing is a way to test functionality. *Stress testing* is a way to test reliability. *Load testing* is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

Stability testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of non-functional software testing is often referred to as load (or endurance) testing.

Usability testing

Usability testing is needed to check if the user interface is easy to use and understand.

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).

Actual translation to human languages must be tested, too. Possible localization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- If several people translate strings, technical terminology may become inconsistent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically in run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut which has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes which are appropriate in the source language, may be inappropriate in the target language; for example, CJK characters may become unreadable if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to fail.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that wasn't localized.

- Localized operating systems may have differently-named system configuration files and environment variables and different formats for date and currency.

To avoid these and other localization problems, a tester who knows the target language must run the program with all the possible use cases for translation to see if the messages are readable, translated correctly in context and don't cause failures.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

The testing process

Traditional CMMI or waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer. This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Agile or Extreme development model

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process). The ultimate goal of this test process is to achieve continuous deployment where software updates can be published to the public frequently.

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing. The sample below is common among organizations employing the Waterfall development model.

- **Requirements analysis**: Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning**: Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- **Test development**: Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.
- **Test execution**: Testers execute the software based on the plans and test documents then report any errors found to the development team.
- **Test reporting**: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
- **Test result analysis**: Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with later.
- **Defect Retesting**: Once a defect has been dealt with by the development team, it is retested by the testing team. AKA Resolution testing.
- **Regression testing**: It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything, and that the software product as a whole is still working correctly.
- **Test Closure**: Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:

    - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
    - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
    - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently-used software measures, often called *metrics*, which are used to assist in determining the state of the software or the adequacy of the testing.

Testing artifacts

Software testing process can produce several artifacts.

Test plan
: A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Traceability matrix
: A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when the source documents are changed, or to verify that the test results are correct.

Test case
: A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in

more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script
: The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools. Today, test scripts can be manual, automated, or a combination of both.

Test suite
: The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test data
: In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

Test harness
: The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. No certification currently offered actually requires the applicant to demonstrate the ability to test software. No certification is based on a widely accepted body of knowledge. This has led some to declare that the testing field is not ready for certification. Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.

Software testing certification types

- *Exam-based*: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI]
- *Education-based*: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)].

Testing certifications

- Certified Associate in Software Testing (CAST) offered by the Quality Assurance Institute (QAI)
- CATe offered by the *International Institute for Software Testing*
- Certified Manager in Software Testing (CMST) offered by the Quality Assurance Institute (QAI)
- Certified Software Tester (CSTE) offered by the Quality Assurance Institute (QAI)
- Certified Software Test Professional (CSTP) offered by the *International Institute for Software Testing*
- CSTP (TM) (Australian Version) offered by *K. J. Ross & Associates*
- ISEB offered by the Information Systems Examinations Board
- ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board
- ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board
- TMPF TMap Next Foundation offered by the *Examination Institute for Information Science*
- TMPA TMap Next Advanced offered by the *Examination Institute for Information Science*

Quality assurance certifications

- CMSQ offered by the *Quality Assurance Institute* (QAI).
- CSQA offered by the *Quality Assurance Institute* (QAI)
- CSQE offered by the American Society for Quality (ASQ)
- CQIA offered by the American Society for Quality (ASQ)

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?
: Members of the "context-driven" school of testing believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.

Agile vs. traditional
: Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles, whereas

government and military software providers are slow to embrace this methodology in favour of traditional test-last models (e.g. in the Waterfall model).

Exploratory test vs. scripted
: Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated
: Some writers believe that test automation is so expensive relative to its value that it should be used sparingly. More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then can be considered as a way to capture and implement the requirements.

Software design vs. software implementation
: Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?
: The idea is that any form of observation is also an interaction—the act of testing can also affect that which is being tested.

Chapter 2

Application Programming Interface and Code Coverage

Application programming interface

An **Application Programming Interface** (**API**) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. It serves as an interface between different software programs and facilitates their interaction, similar to the way the user interface facilitates interaction between humans and computers.

An API can be created for applications, libraries, operating systems, etc, as a way to define their "vocabularies" and resources request conventions (e.g. functions calling conventions). It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer program and the implementer program of the API.

Concept

An API is an abstraction that describes an interface for the interaction with a set of functions used by components of a software system. The software providing the functions described by an API is said to be an *implementation* of the API.

An API can be:

- general, the full set of an API that is bundled in the libraries of a programming language, e.g. Standard Template Library in C++ or Java API.
- specific, meant to address a specific problem, e.g. Google Maps API or Java API for XML Web Services.
- language-dependent, meaning it is only available by using the syntax and elements of a particular language, which makes the API more convenient to use.

- language-independent, written so that it can be called from several programming languages. This is a desirable feature for a service-oriented API that is not bound to a specific process or system and may be provided as remote procedure calls or web services. For example, a website that allows users to review local restaurants is able to layer their reviews over maps taken from Google Maps, because Google Maps has an API that facilitates this functionality. Google Maps' API controls what information a third-party site can use and how they can use it.

API may be used to refer to a complete interface, a single function, or even a set of APIs provided by an organization. Thus, the scope of meaning is usually determined by the context of usage.

Advanced explanation

An API may describe the ways in which a particular task is performed. For example, in Unix systems, the `math.h` include file for the C language contains the definition of the mathematical functions available in the C language library for mathematical processing (usually called `libm`). This file would describe how to use these functions and the expected result. For example, on a Unix system the command `man 3 sqrt` will present the signature of the function `sqrt` in the form:

```
SYNOPSIS
            #include <math.h>
            double sqrt(double X);
            float  sqrtf(float X);
DESCRIPTION
       DESCRIPTION
       sqrt computes the positive square root of the argument. ...
RETURNS
       On success, the square root is returned. If X is real and
positive...
```

that means that the function returns the square root of a positive floating point number (`single` or `double` precision) as another floating point number. Hence the API in this case can be interpreted as the collection of the included files used by the C language and its human readable description provided by the main pages.

API in modern languages

Most of the modern programming languages provide the documentation associated to an API in some digital format that makes it easy to consult on a computer. E.g. perl comes with the tool perldoc:

```
$ perldoc -f sqrt
       sqrt EXPR
       sqrt    #Return the square root of EXPR.  If EXPR is omitted,
returns
               #square root of $_.  Only works on non-negative
operands, unless
```

```
#you've loaded the standard Math::Complex module.
```

python comes with the tool pydoc:

```
$ pydoc math.sqrt
Help on built-in function sqrt in math:
math.sqrt = sqrt(...)
    sqrt(x)
    Return the square root of x.
```

ruby comes with the tool `ri`:

```
$ ri Math::sqrt
------------------------------------------------------------
Math::sqrt
     Math.sqrt(numeric)    => float
------------------------------------------------------------------------
-
     Returns the non-negative square root of _numeric_.
```

Java comes with the documentation organized in html pages (JavaDoc format), while Microsoft distributes the API documentation for its languages (Visual C++, C#, Visual Basic, F#, etc...) embedded in Visual Studio help system.

API in object-oriented languages

In object oriented languages, an API usually includes a description of a set of class definitions, with a set of behaviours associated with those classes. A *behaviour* is the set of rules for how an object, derived from that class, will act in a given circumstance. This abstract concept is associated with the real functionalities exposed, or made available, by the classes that are implemented in terms of class methods.

The API in this case can be conceived as the totality of all the methods publicly exposed by the classes (usually called the class *interface*). This means that the API prescribes the methods by which one interacts with/handles the objects derived from the class definitions.

More generally, one can see the API as the collection of all the *kind* of objects one can derive from the class definitions, and their associated possible behaviours. Again: the use is mediated by the public methods, but in this interpretation, the methods are seen as a *technical detail* of how the behaviour is implemented.

For instance: a class representing a `Stack` can simply expose publicly two methods `push()` (to add a new item to the stack), and `pop()` (to extract the last item, ideally placed on top of the stack).

In this case the API can be interpreted as the two methods `pop()` and `push()`, or, more generally, as the *idea* that one can use an item of type `Stack` that implements the behaviour of a stack: a pile *exposing* its top to add/remove elements.

This concept can be carried to the point where a class interface in an API has no methods at all, but only behaviours associated with it. For instance, the Java language API includes the interface `Serializable`, which requires that each class that implements it should behave in a serialized fashion. This does not require to have any public method, but rather requires that any class implements it to have a representation that can be *saved* (serialized) at any time (this is typically true for any class containing simple data and no link to external resources, like an open connection to a file, a remote system, or an external device).

In this sense, in object oriented languages, the API defines a set of behaviors, possibly mediated by a set of class methods.

In such languages, the API is still distributed as a library. For example, the Java language libraries include a set of APIs that are provided in the form of the JDK used by the developers to build new Java programs. The JDK includes the documentation of the API in JavaDoc notation.

The quality of the documentation associated to an API is often a factor determining its success in terms of ease of use.

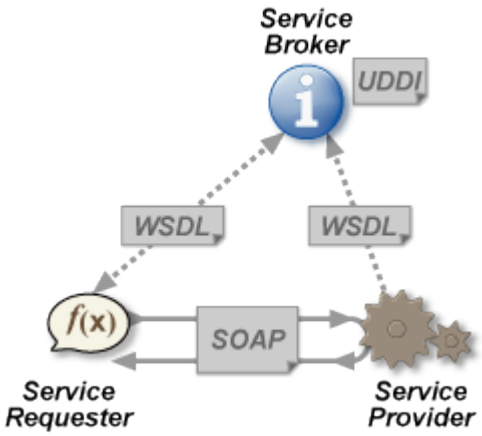
API and protocols

An API can also be an implementation of a protocol.

In general the difference between an API and a protocol is that the protocol defines a standard way to exchange requests and responses based on a common transport, while an API provides a library to be used directly: hence there can be no *transport*, but rather only simple *function calls*.

When an API implements a protocol it can be based on proxy methods for remote invocations that underneath rely on the communication protocol. The role of the API can be exactly to hide the protocol.

Web service



Web services architecture.

A **web service** is typically an application programming interface (API) or **Web API** that is accessed via Hypertext Transfer Protocol (HTTP) and executed on a remote system, hosting the requested service. Web services tend to fall into one of two camps: big web services and RESTful web services.

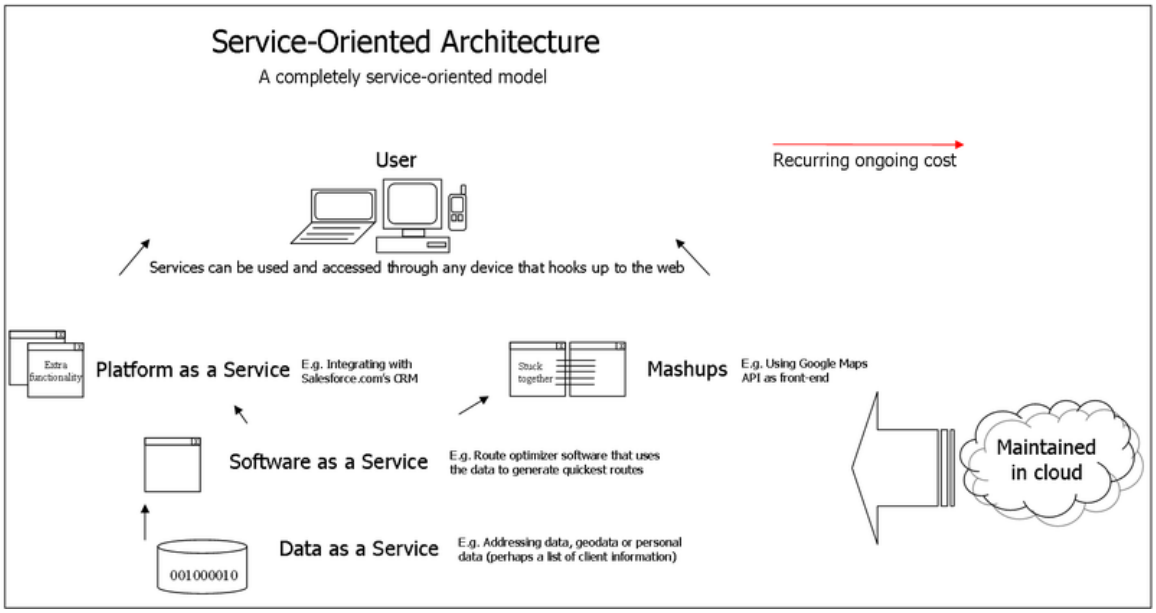
The W3C defines a "web service" as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

The W3C also states, "We can identify two major classes of Web services, REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and arbitrary Web services, in which the service may expose an arbitrary set of operations."

Big web services

"Big web services" use Extensible Markup Language (XML) messages that follow the SOAP standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP *endpoint*, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

Web API



Web services in a service-oriented architecture.

Web API is a development in web services (in a movement called Web 2.0) where emphasis has been moving away from SOAP based services towards Representational State Transfer (REST) based communications. REST services do not require XML, SOAP, or WSDL service-API definitions.

Web APIs allow the combination of multiple web services into new applications known as mashups.

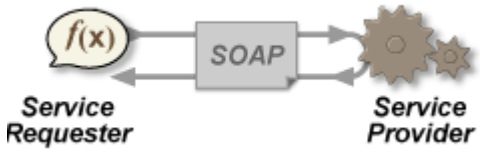
When used in the context of web development, Web API is typically a defined set of Hypertext Transfer Protocol (HTTP) request messages along with a definition of the structure of response messages, usually expressed in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

When running composite web services, each sub service can be considered autonomous. The user has no control over these services. Also the web services themselves are not reliable; the service provider may remove, change or update their services without giving notice to users. The reliability and fault tolerance is not well supported; faults may happen during the execution. Exception handling in the context of web services is still an open research issue. Still it can be handled by responding with an error object to the client.

Styles of use

Web services are a set of tools that can be used in a number of ways. The three most common styles of use are RPC, SOA and REST.

Remote procedure calls



Architectural elements involved in the XML-RPC.

RPC web services present a distributed function (or method) call interface that is familiar with many developers. Typically, the basic unit of RPC web services is the WSDL operation.

The first web services tools were focused on RPC, and as a result this style is widely deployed and supported. However, it is sometimes criticized for not being loosely coupled, because it was often implemented by mapping services directly to language-specific functions or method calls. Many vendors felt this approach to be a dead end, and pushed for RPC to be disallowed in the WS-I Basic Profile.

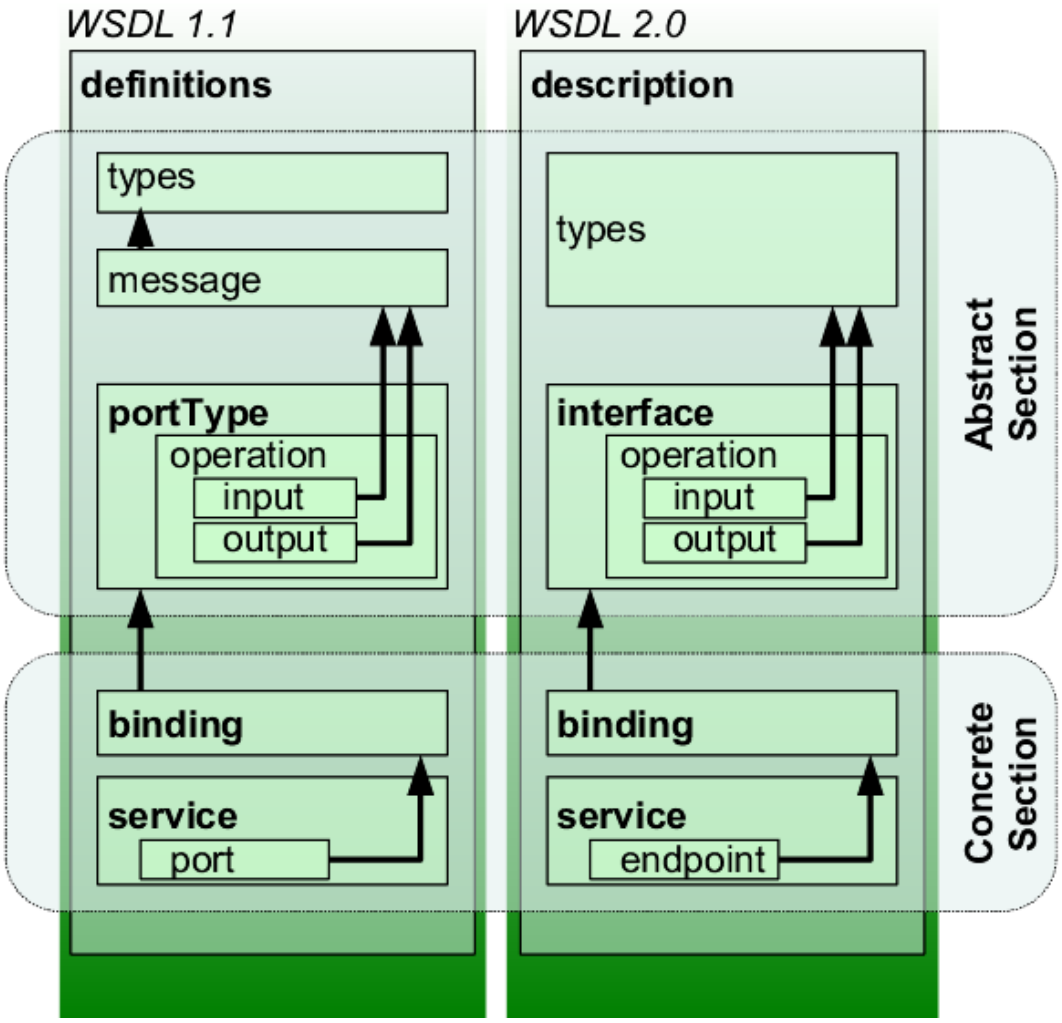
Other approaches with nearly the same functionality as RPC are Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM) or Sun Microsystems's Java/Remote Method Invocation (RMI).

Service-oriented architecture

Web services can also be used to implement an architecture according to service-oriented architecture (SOA) concepts, where the basic unit of communication is a message, rather than an operation. This is often referred to as "message-oriented" services.

SOA web services are supported by most major software vendors and industry analysts. Unlike RPC web services, loose coupling is more likely, because the focus is on the "contract" that WSDL provides, rather than the underlying implementation details.

Middleware analysts use enterprise service buses which combine message-oriented processing and web services to create an event-driven SOA. One example of an open-source ESB is Mule, another one is Open ESB.



Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents.

Representational state transfer (REST)

REST attempts to describe architectures which use HTTP or similar protocols by constraining the interface to a set of well-known, standard operations (like GET, POST, PUT, DELETE for HTTP). Here, the focus is on interacting with stateful resources, rather than messages or operations.

An architecture based on REST (one that is 'RESTful') can use WSDL to describe SOAP messaging over HTTP, can be implemented as an abstraction purely on top of SOAP (e.g., WS-Transfer), or can be created without using SOAP at all.

WSDL version 2.0 offers support for binding to all the HTTP request methods (not only GET and POST as in version 1.1) so it enables a better implementation of RESTful Web services. However, support for this specification is still poor in software development kits, which often offer tools only for WSDL 1.1.

Design methodologies

A web service can be written in two ways:

- A developer using the "bottom up method" first writes the implementing class in a programming language, and then uses a WSDL generating tool to expose its methods as a web service. This is often the simpler approach.
- A developer using the "top down method" first writes the WSDL document and then uses a code generating tool to produce the class skeleton, which he or she later completes. This way is generally considered more difficult but can produce cleaner designs

Criticisms

Critics of non-RESTful web services often complain that they are too complex and based upon large software vendors or integrators, rather than typical open source implementations. There are open source implementations like Apache Axis and Apache CXF.

One key concern of the REST web service developers is that the SOAP WS toolkits make it easy to define new interfaces for remote interaction, often relying on introspection to extract the WSDL, since a minor change on the server (even an upgrade of the SOAP stack) can result in different WSDL and a different service interface. The client-side classes that can be generated from WSDL and XSD descriptions of the service are often similarly tied to a particular version of the SOAP endpoint and can break if the endpoint changes or the client-side SOAP stack is upgraded. Well-designed SOAP endpoints (with handwritten XSD and WSDL) do not suffer from this but there is still the problem that a custom interface for every service requires a custom client for every service.

There are also concerns about performance due to web services' use of XML as a message format and SOAP/HTTP in enveloping and transport.

Use of APIs to share content

The practice of publishing APIs has allowed web communities to create an open architecture for sharing content and data between communities and applications. In this way, content that is created in one place can be dynamically posted and updated in multiple locations on the web.

1. Photos can be shared from sites like Flickr and Photobucket to social network sites like Facebook and MySpace.
2. Content can be embedded, e.g. embedding a presentation from SlideShare on a LinkedIn profile.

3. Content can be dynamically posted. Sharing live comments made on Twitter with a Facebook account, for example, is enabled by their APIs.
4. Video content can be embedded on sites which are served by another host.
5. User information can be shared from web communities to outside applications, delivering new functionality to the web community that shares its user data via an open API. One of the best examples of this is the Facebook Application platform. Another is the Open Social platform.

Implementations

The POSIX standard defines an API that allows a wide range of common computing functions to be written in a way such that they may operate on many different systems (Mac OS X, and various Berkeley Software Distributions (BSDs) implement this interface); however, making use of this requires re-compiling for each platform. A compatible API, on the other hand, allows compiled object code to function without any changes to the system implementing that API. This is beneficial to both software providers (where they may distribute existing software on new systems without producing and distributing upgrades) and users (where they may install older software on their new systems without purchasing upgrades), although this generally requires that various software libraries implement the necessary APIs as well.

Microsoft has shown a strong commitment to a backward compatible API, particularly within their Windows API (Win32) library, such that older applications may run on newer versions of Windows using an executable-specific setting called "Compatibility Mode".

Apple Inc. has shown less concern, breaking compatibility or implementing an API in a slower "emulation mode"; this allows greater freedom in development, at the cost of making older software obsolete.

Among Unix-like operating systems, there are many related but incompatible operating systems running on a common hardware platform (particularly Intel 80386-compatible systems). There have been several attempts to standardize the API such that software vendors may distribute one binary application for all these systems; however, to date, none of these have met with much success. The Linux Standard Base is attempting to do this for the Linux platform, while many of the BSD Unixes, such as FreeBSD, NetBSD, and OpenBSD, implement various levels of API compatibility for both backward compatibility (allowing programs written for older versions to run on newer distributions of the system) and cross-platform compatibility (allowing execution of foreign code without recompiling).

Release policies

The two options for releasing API are:

1. Protecting information on APIs from the general public. For example, Sony used to make its official PlayStation 2 API available only to licensed PlayStation developers. This enabled Sony to control who wrote PlayStation 2 games. This gives companies quality control privileges and can provide them with potential licensing revenue streams.
2. Making APIs freely available. For example, Microsoft makes the Microsoft Windows API public, and Apple releases its APIs Carbon and Cocoa, so that software can be written for their platforms.

A mix of the two behaviors can be used as well.

ABIs

The related term application binary interface (ABI) is a lower level definition concerning details at the assembly language level. For example, the Linux Standard Base is an ABI, while POSIX is an API.

API examples

- ASPI for SCSI device interfacing
- Carbon and Cocoa for the Macintosh
- DirectX for Microsoft Windows
- Java APIs
- OpenGL cross-platform graphics API
- OpenAL cross-platform sound API
- OpenCL cross-platform API for general-purpose computing for CPUs & GPUs
- OpenMP API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms.
- Simple DirectMedia Layer (SDL)
- Windows API
- Talend integrates its data management with BPM from Bonita Open Solution

Language bindings and interface generators

APIs that are intended to be used by more than one high-level programming language often provide, or are augmented with, facilities to automatically map the API to features (syntactic or semantic) that are more natural in those languages. This is known as language binding, and is itself an API. The aim is to encapsulate most of the required functionality of the API, leaving a "thin" layer appropriate to each language.

Below are listed some interface generator tools which bind languages to APIs at compile time.

- SWIG opensource interfaces bindings generator from many languages to many languages (Typically Compiled->Scripted)

- F2PY: Fortran to Python interface generator.

Code coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing. In time, the use of code coverage has been extended to the field of digital hardware, the contemporary design methodology of which relies on hardware description languages (HDLs).

Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM* in 1963.

Code coverage is one consideration in the safety certification of avionics equipment. The standard by which avionics gear is certified by the Federal Aviation Administration (FAA) is documented in DO-178B.

Coverage criteria

To measure how well the program is exercised by a test suite, one or more *coverage criteria* are used.

Basic coverage criteria

There are a number of coverage criteria, the main ones being:

- **Function coverage** - Has each function (or subroutine) in the program been called?
- **Statement coverage** - Has each node in the program been executed?
- **Decision coverage** or **branch coverage** - Has every edge in the program been executed? For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?
- **Condition coverage** (or predicate coverage) - Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Both decision and condition coverage should be satisfied.

For example, consider the following C++ function:

```
int foo(int x, int y)
{
```

```
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Assume this function is a part of some bigger program and this program was run with some test suite.

- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as `foo(1,1)`, as in this case, every line in the function is executed including `z = x;`.
- Tests calling `foo(1,1)` and `foo(1,0)` will satisfy *decision coverage*, as in the first case the `if` condition is satisfied and `z = x;` is executed, and in the second it is not.
- *Condition coverage* can be satisfied with tests that call `foo(1,1)`, `foo(1,0)` and `foo(0,0)`. These are necessary as in the first two cases `(x>0)` evaluates to `true` while in the third it evaluates `false`. At the same time, the first case makes `(y>0)` `true` while the second and third make it `false`.

In languages, like Pascal, where standard boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage. For example, consider the following fragment of code:

```
if a and b then
```

Condition coverage can be satisfied by two tests:

- `a=true`, `b=false`
- `a=false`, `b=true`

However, this set of tests does not satisfy decision coverage as in neither case will the `if` condition be met.

Fault injection may be necessary to ensure that all conditions and branches of exception handling code have adequate coverage during testing.

Modified condition/decision coverage

For safety-critical applications (e.g. for avionics software) it is often required that **modified condition/decision coverage (MC/DC)** is satisifed. This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

```
if (a or b) and c then
```

The condition/decision criteria will be satisfied by the following set of tests:

- a=true, b=true, c=true
- a=false, b=false, c=false

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

- a=false, b=true, c=true
- a=true, b=false, c=true
- a=true, b=true, c=false

Multiple condition coverage

This criteria requires that all combinations of conditions inside each decision are tested. For example, the code fragment from the previous section will require eight tests:

- a=false, b=false, c=false
- a=false, b=false, c=true
- a=false, b=true, c=false
- a=false, b=true, c=true
- a=true, b=false, c=false
- a=true, b=false, c=true
- a=true, b=true, c=false
- a=true, b=true, c=true

Other coverage criteria

There are further coverage criteria, which are used less often:

- **Linear Code Sequence and Jump (LCSAJ) coverage** - has every LCSAJ been executed?
- **JJ-Path coverage** - have all jump to jump paths (aka LCSAJs) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?
- **Loop coverage** - Has every possible loop been executed zero times, once, and more than once?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage.

Some of the coverage criteria above are connected. For instance, path coverage implies decision, statement and entry/exit coverage. Decision coverage implies statement coverage, because every statement is part of a branch.

Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of *n* decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem). Methods for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

In practice

The target software is built with special options or libraries and/or run under a special environment such that every function that is exercised (executed) in the program(s) is mapped back to the function points in the source code. This process allows developers and quality assurance personnel to look for parts of a system that are rarely or never accessed under normal conditions (error handling and the like) and helps reassure test engineers that the most important conditions (function points) have been tested. The resulting output is then analyzed to see what areas of code have not been exercised and the tests are updated to include these areas as necessary. Combined with other code coverage methods, the aim is to develop a rigorous, yet manageable, set of regression tests.

In implementing code coverage policies within a software development environment one must consider the following:

- What are coverage requirements for the end product certification and if so what level of code coverage is required? The typical level of rigor progression is as follows: Statement, Branch/Decision, Modified Condition/Decision Coverage(MC/DC), LCSAJ (Linear Code Sequence and Jump)
- Will code coverage be measured against tests that verify requirements levied on the system under test (DO-178B)?
- Is the object code generated directly traceable to source code statements? Certain certifications, (ie. DO-178B Level A) require coverage at the assembly level if this is not the case: "Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences" (DO-178B) para-6.4.4.2.

Test engineers can look at code coverage test results to help them devise test cases and input or configuration sets that will increase the code coverage over vital functions. Two common forms of code coverage used by testers are statement (or line) coverage and path (or edge) coverage. Line coverage reports on the execution footprint of testing in terms of

which lines of code were executed to complete the test. Edge coverage reports which branches or code decision points were executed to complete the test. They both report a coverage metric, measured as a percentage. The meaning of this depends on what form(s) of code coverage have been used, as 67% path coverage is more comprehensive than 67% statement coverage.

Generally, code coverage tools and libraries exact a performance and/or memory or other resource cost which is unacceptable to normal operations of the software. Thus, they are only used in the lab. As one might expect, there are classes of software that cannot be feasibly subjected to these coverage tests, though a degree of coverage mapping can be approximated through analysis rather than direct testing.

There are also some sorts of defects which are affected by such tools. In particular, some race conditions or similar real time sensitive operations can be masked when run under code coverage environments; and conversely, some of these defects may become easier to find as a result of the additional overhead of the testing code.

Chapter 3

Fault Injection and Mutation Testing

Fault injection

In software testing, **fault injection** is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed. It is often used with stress testing and is widely considered to be an important part of developing robust software. Robustness testing (also known as Syntax Testing, Fuzzing or Fuzz testing) is a type of fault injection commonly used to test for vulnerabilities in communication interfaces such as protocols, command line parameters, or APIs.

The propagation of a fault through to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary, therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures. This mechanism is termed the fault-error-failure cycle and is a key mechanism in dependability.

History

The technique of fault injection dates back to the 1970s when it was first used to induce faults at a hardware level. This type of fault injection is called Hardware Implemented Fault Injection (HWIFI) and attempts to simulate hardware failures within a system. The first experiments in hardware fault injection involved nothing more than shorting connections on circuit boards and observing the effect on the system (bridging faults). It was used primarily as a test of the dependability of the hardware system. Later specialised hardware was developed to extend this technique, such as devices to bombard specific areas of a circuit board with heavy radiation. It was soon found that faults could be induced by software techniques and that aspects of this technique could be useful for assessing software systems. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI).

Software Implemented fault injection

SWIFI techniques for software fault injection can be categorized into two types: compile-time injection and runtime injection.

Compile-time injection is an injection technique where source code is modified to inject simulated faults into a system. One method is called mutation testing which changes existing lines of code so that they contain faults. A simple example of this technique could be changing

```
a = a + 1
  to
a = a - 1
```

Code mutation produces faults which are very similar to those unintentionally added by programmers.

A refinement of code mutation is *Code Insertion Fault Injection* which adds code, rather than modifies existing code. This is usually done through the use of perturbation functions which are simple functions which take an existing value and perturb it via some logic into another value, for example

```
int pFunc(int value) {
  return value + 20;
}
int main(int argc, char * argv[]) {
  int a = pFunc(aFunction(atoi(argv)));
  if (a > 20) {
    /* do something */
  } else {
    /* do something else */
  }
}
```

In this case pFunc is the perturbation function and it is applied to the return value of the function that has been called introducing a fault into the system.

Runtime Injection techniques use a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways, such as: Time Based triggers (When the timer reaches a specified time an interrupt is generated and the interrupt handler associated with the timer can inject the fault.); Interrupt Based Triggers (Hardware exceptions and software trap mechanisms are used to generate an interrupt at a specific place in the system code or on a particular event within the system, for instance access to a specific memory location).

Runtime injection techniques can use a number of different techniques to insert faults into a system via a trigger.

- Corruption of memory space: This technique consists of corrupting RAM, processor registers, and I/O map.
- Syscall interposition techniques: This is concerned with the fault propagation from operating system kernel interfaces to executing systems software. This is done by intercepting operating system calls made by user-level software and injecting faults into them.
- Network Level fault injection: This technique is concerned with the corruption, loss or reordering of network packets at the network interface.

These techniques are often based around the debugging facilities provided by computer processor architectures.

Protocol software fault injection

Complex software systems, especially multi-vendor distributed systems based on open standards, perform input/output operations to exchange data via stateful, structured exchanges known as "protocols." One kind of fault injection that is particularly useful to test protocol implementations (a type of software code that has the unusual characteristic in that it cannot predict or control its input) is fuzzing. Fuzzing is an especially useful form of Black-box testing since the various invalid inputs that are submitted to the software system do not depend on, and are not created based on knowledge of, the details of the code running inside the system.

Fault injection tools

Although these types of faults can be injected by hand the possibility of introducing an unintended fault is high, so tools exist to parse a program automatically and insert faults.

Research tools

A number of SWIFI Tools have been developed and a selection of these tools is given here. Six commonly used fault injection tools are Ferrari, FTAPE , Doctor, Orchestra, Xception and Grid-FIT.

- Ferrari (Fault and Error Automatic Real-Time Injection) is based around software traps that inject errors into a system. The traps are activated by either a call to a specific memory location or a timeout. When a trap is called the handler injects a fault into the system. The faults can either be transient or permanent. Research conducted with Ferrari shows that error detection is dependent on the fault type and where the fault is inserted .
- FTAPE (Fault Tolerance and Performance Evaluator) can inject faults, not only into memory and registers, but into disk accesses as well. This is achieved by inserting a special disk driver into the system that can inject faults into data sent

and received from the disk unit. FTAPE also has a synthetic load unit that can simulate specific amounts of load for robustness testing purposes .

- DOCTOR (IntegrateD SOftware Fault InjeCTiOn EnviRonment) allows injection of memory and register faults, as well as network communication faults. It uses a combination of time-out, trap and code modification. Time-out triggers inject transient memory faults and traps inject transient emulated hardware failures, such as register corruption. Code modification is used to inject permanent faults .
- Orchestra is a script driven fault injector which is based around Network Level Fault Injection. Its primary use is the evaluation and validation of the fault-tolerance and timing characteristics of distributed protocols. Orchestra was initially developed for the Mach Operating System and uses certain features of this platform to compensate for latencies introduced by the fault injector. It has also been successfully ported to other operating systems.
- Xception is designed to take advantage of the advanced debugging features available on many modern processors. It is written to require no modification of system source and no insertion of software traps, since the processor's exception handling capabilities trigger fault injection. These triggers are based around accesses to specific memory locations. Such accesses could be either for data or fetching instructions. It is therefore possible to accurately reproduce test runs because triggers can be tied to specific events, instead of timeouts .
- Grid-FIT (Grid – Fault Injection Technology) is a dependability assessment method and tool for assessing Grid services by fault injection. Grid-FIT is derived from an earlier fault injector WS-FIT which was targeted towards Java Web Services implemented using Apache Axis transport. Grid-FIT utilises a novel fault injection mechanism that allows network level fault injection to be used to give a level of control similar to Code Insertion fault injection whilst being less invasive .
- LFI (Library-level Fault Injector) is an automatic testing tool suite, used to simulate in a controlled testing environment, exceptional situations that programs need to handle at runtime but that are not easy to check via input testing alone. LFI automatically identifies the errors exposed by shared libraries, finds potentially buggy error recovery code in program binaries and injects the desired faults at the boundary between shared libraries and applications.

Commercial tools

- ExhaustiF is a commercial software tool used for grey box testing based on software fault injection (SWIFI) to improve reliability of software intensive systems. The tool can be used during system integration and system testing phases of any software development lifecycle, complementing other testing tools as well. ExhaustiF is able to inject faults into both software and hardware. When injecting simulated faults in software, ExhaustiF offers the following fault types: Variable Corruption and Procedure Corruption. The catalogue for hardware fault injections includes faults in Memory (I/O, RAM) and CPU (Integer Unit, Floating Unit). There are different versions available for RTEMS/ERC32, RTEMS/Pentium, Linux/Pentium and MS-Windows/Pentium.

- Holodeck is a test tool developed by Security Innovation that uses fault injection to simulate real-world application and system errors for Windows applications and services. Holodeck customers include many major commercial software development companies, including Microsoft, Symantec, EMC and Adobe. It provides a controlled, repeatable environment in which to analyze and debug error-handling code and application attack surfaces for fragility and security testing. It simulates file and network fuzzing faults as well as a wide range of other resource, system and custom-defined faults. It analyzes code and recommends test plans and also performs function call logging, API interception, stress testing, code coverage analysis and many other application security assurance functions.

- Codenomicon Defensics is a blackbox test automation framework that does fault injection to more than 150 different interfaces including network protocols, API interfaces, files, and XML structures. The commercial product was launched in 2001, after five years of research at University of Oulu in the area of software fault injection. A thesis work explaining the used fuzzing principles was published by VTT, one of the PROTOS consortium members.

- The Mu Service Analyzer is a commercial service testing tool developed by Mu Dynamics. The Mu Service Analyzer performs black box and white box testing of services based on their exposed software interfaces, using denial-of-service simulations, service-level traffic variations (to generate invalid inputs) and the replay of known vulnerability triggers. All these techniques exercise input validation and error handling and are used in conjunction with valid protocol monitors and SNMP to characterize the effects of the test traffic on the software system. The Mu Service Analyzer allows users to establish and track system-level reliability, availability and security metrics for any exposed protocol implementation. The tool has been available in the market since 2005 by customers in the North America, Asia and Europe, especially in the critical markets of network operators (and their vendors) and Industrial control systems (including Critical infrastructure).

- Xception is a commercial software tool developed by Critical Software SA used for black box and white box testing based on software fault injection (SWIFI) and Scan Chain fault injection (SCIFI). Xception allows users to test the robustness of their systems or just part of them, allowing both Software fault injection and Hardware fault injection for a specific set of architectures. The tool has been used in the market since 1999 and has customers in the American, Asian and European markets, especially in the critical market of aerospace and the telecom market. The full Xception product family includes: a) The main Xception tool, a state-of-the-art leader in Software Implemented Fault Injection (SWIFI) technology; b) The Easy Fault Definition (EFD) and Xtract (Xception Analysis Tool) add-on tools; c) The extended Xception tool (eXception), with the fault injection extensions for Scan Chain and pin-level forcing.

Libraries

- TestApi is a shared-source API library, which provides facilities for fault injection testing as well as other testing types, data-structures and algorithms for .NET applications.

Application of fault injection

Fault injection can take many forms. In the testing of operating systems for example, fault injection is often performed by a *driver* (kernel-mode software) that intercepts *system calls* (calls into the kernel) and randomly returning a failure for some of the calls. This type of fault injection is useful for testing low level user mode software. For higher level software, various methods inject faults. In managed code, it is common to use instrumentation. Although fault injection can be undertaken by hand a number of fault injection tools exist to automate the process of fault injection .

Depending on the complexity of the API for the level where faults are injected, fault injection tests often must be carefully designed to minimise the number of false positives. Even a well designed fault injection test can sometimes produce situations that are impossible in the normal operation of the software. For example, imagine there are two API functions, `Commit` and `PrepareForCommit`, such that alone, each of these functions can possibly fail, but if `PrepareForCommit` is called and succeeds, a subsequent call to `Commit` is guaranteed to succeed. Now consider the following code:

```
error = PrepareForCommit();
if (error == SUCCESS) {
  error = Commit();
  assert(error == SUCCESS);
}
```

Often, it will be infeasible for the fault injection implementation to keep track of enough state to make the guarantee that the API functions make. In this example, a fault injection test of the above code might hit the assert, whereas this would never happen in normal operation.

Mutation testing

Mutation testing (or *Mutation analysis* or *Program mutation*) is a method of software testing, which involves modifying programs' source code or byte code in small ways. In short, any tests which pass after code has been mutated are considered defective. These so-called *mutations*, are based on well-defined *mutation operators* that either mimic typical programming errors (such as using the wrong operator or variable name) or force

the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Tests can be created to verify the correctness of the implementation of a given software system. But the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. (This technological problem is itself an instance of a deeper philosophical problem named "Quis custodiet ipsos custodes?" ["Who will guard the guards?"].) In this context, mutation testing was pioneered in the 1970s to locate and expose weaknesses in test suites. The theory was that if a mutation was introduced without the behavior (generally output) of the program being affected, this indicated either that the code that had been mutated was never executed (redundant code) or that the testing suite was unable to locate the injected fault. In order for this to function at any scale, a large number of mutations had to be introduced into a large program, leading to the compilation and execution of an extremely large number of copies of the program. This problem of the expense of mutation testing had reduced its practical use as a method of software testing, but the increased use of object oriented programming languages and unit testing frameworks has led to the creation of mutation testing tools for many programming languages as a means to test individual portions of an application.

Mutation testing was originally proposed by Richard Lipton as a student in 1971, and first developed and published by DeMillo, Lipton and Sayward. The first implementation of a mutation testing tool was by Timothy Budd as part of his PhD work (titled *Mutation Analysis*) in 1980 from Yale University.

Recently, with the availability of massive computing power, there has been a resurgence of mutation analysis within the computer science community, and work has been done to define methods of applying mutation testing to object oriented programming languages and non-procedural languages such as XML, SMV, and finite state machines.

In 2004 a company called Certess Inc. extended many of the principles into the hardware verification domain. Whereas mutation analysis only expects to detect a difference in the output produced, Certess extends this by verifying that a checker in the testbench will actually detect the difference. This extension means that all three stages of verification, namely: activation, propagation and detection are evaluated. They have called this functional qualification.

Fuzzing is a special area of mutation testing. In fuzzing, the messages or data exchanged inside communication interfaces (both inside and between software instances) are mutated, in order to catch failures or differences in processing the data. Codenomicon (2001) and Mu Dynamics (2005) evolved fuzzing concepts to a fully stateful mutation testing platform, complete with monitors for thoroughly exercising protocol implementations.

Mutation testing overview

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be *killed*.

For example, consider the following C++ code fragment:

```
if (a && b)
    c = 1;
else
    c = 0;
```

The condition mutation operator would replace '&&' with '||' and produce the following mutant:

```
if (a || b)
    c = 1;
else
    c = 0;
```

Now, for the test to kill this mutant, the following condition should be met:

- Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this.
- The value of 'c' should be propagated to the program's output and checked by the test.

Weak mutation testing (or *weak mutation coverage*) requires that only the first condition is satisfied. *Strong mutation testing* requires that both conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

Equivalent mutants

Many mutation operators can produce equivalent mutants. For example, consider the following code fragment:

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

Boolean relation mutation operator will replace "==" with ">=" and produce the following mutant:

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index>=10)
        break;
}
```

However, it is not possible to find a test case which could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called *equivalent mutants*.

Equivalent mutants detection is one of biggest obstacles for practical usage of mutation testing. The effort, needed to check if mutants are equivalent or not, can be very high even for small programs.

Mutation operators

A variety of mutation operators were explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion.
- Replace each boolean subexpression with *true* and *false*.
- Replace each arithmetic operation with another one, e.g. + with *, - and /.
- Replace each boolean relation with another one, e.g. > with >=, == and <=.
- Replace each variable with another variable declared in the same scope (variable types should be the same).

These mutation operators are also called traditional mutation operators. Beside this, there are mutation operators for object-oriented languages , for concurrent constructions, complex objects like containers etc. They are called class-level mutation operators. For example the MuJava tool offers various class-level mutation operators such as: Access Modifier Change, Type Cast Operator Insertion, Type Cast Operator Deletion. Moreover, mutation operators have been developed to perform security vulnerability testing of programs

Chapter 4

Exploratory Testing, Fuzz Testing and Equivalence Partitioning

Exploratory testing

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design and test execution. Cem Kaner, who coined the term in 1983, now defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."

While the software is being tested, the tester learns things that together with experience and creativity generates new good tests to run. Exploratory testing is often thought of as a black box testing technique. Instead, those who have studied it consider it a test *approach* that can be applied to any test technique, at any stage in the development process. The key is not the test technique nor the item being tested or reviewed; the key is the cognitive engagement of the tester, and the tester's responsibility for managing his or her time.

History

Exploratory testing has always been performed by skilled testers. In the early 1990s, ad hoc was too often synonymous with sloppy and careless work. As a result, a group of test methodologists (now calling themselves the Context-Driven School) began using the term "exploratory" seeking to emphasize the dominant thought process involved in unscripted testing, and to begin to develop the practice into a teachable discipline. This new terminology was first published by Cem Kaner in his book *Testing Computer Software* and expanded upon in *Lessons Learned in Software Testing*. Exploratory testing can be as disciplined as any other intellectual activity.

Description

Exploratory testing seeks to find out how the software actually works, and to ask questions about how it will handle difficult and easy cases. The quality of the testing is dependent on the tester's skill of inventing test cases and finding defects. The more the tester knows about the product and different test methods, the better the testing will be.

To further explain, comparison can be made of freestyle exploratory testing to its antithesis scripted testing. In this activity test cases are designed in advance. This includes both the individual steps and the expected results. These tests are later performed by a tester who compares the actual result with the expected. When performing exploratory testing, expectations are open. Some results may be predicted and expected; others may not. The tester configures, operates, observes, and evaluates the product and its behaviour, critically investigating the result, and reporting information that seems like to be a bug (which threatens the value of the product to some person) or an issue (which threatens the quality of the testing effort).

In reality, testing almost always is a combination of exploratory and scripted testing, but with a tendency towards either one, depending on context.

According to Cem Kaner & James Bach, exploratory testing is more a mindset or "...a way of thinking about testing" than a methodology. They also say that it crosses a continuum from slightly exploratory (slightly ambiguous or vaguely scripted testing) to highly exploratory (freestyle exploratory testing).

The documentation of exploratory testing ranges from documenting all tests performed to just documenting the bugs. During pair testing, two persons create test cases together; one performs them, and the other documents. Session-based testing is a method specifically designed to make exploratory testing auditable and measurable on a wider scale.

Exploratory testers often use tools, including screen capture or video tools as a record of the exploratory session, or tools to quickly help generate situations of interest, e.g. James Bach's Perlclip.

Benefits and drawbacks

The main advantage of exploratory testing is that less preparation is needed, important bugs are found quickly, and at execution time, the approach tends to be more intellectually stimulating than execution of scripted tests.

Another major benefit is that testers can use deductive reasoning based on the results of previous results to guide their future testing on the fly. They do not have to complete a current series of scripted tests before focusing in on or moving on to exploring a more target rich environment. This also accelerates bug detection when used intelligently.

Another benefit is that, after initial testing, most bugs are discovered by some sort of exploratory testing. This can be demonstrated logically by stating, "Programs that pass certain tests tend to continue to pass the same tests and are more likely to fail other tests or scenarios that are yet to be explored."

Disadvantages are that tests invented and performed on the fly can't be reviewed in advance (and by that prevent errors in code and test cases), and that it can be difficult to show exactly which tests have been run.

Freestyle exploratory test ideas, when revisited, are unlikely to be performed in exactly the same manner, which can be an advantage if it is important to find new errors; or a disadvantage if it is more important to repeat specific details of the earlier tests. This can be controlled with specific instruction to the tester, or by preparing automated tests where feasible, appropriate, and necessary, and ideally as close to the unit level as possible.

Usage

Exploratory testing is particularly suitable if requirements and specifications are incomplete, or if there is lack of time. The approach can also be used to verify that previous testing has found the most important defects.

Fuzz testing

Fuzz testing or **fuzzing** is a software testing technique that provides invalid, unexpected, or random data to the inputs of a program. If the program fails (for example, by crashing or failing built-in code assertions), the defects can be noted.

The term Fuzz originated from a class project topic in Prof. Barton Miller's graduate Advanced Operating System class at the University of Wisconsin in the Fall of 1988. The assignment was titled "Operating System Utility Program Reliability - The Fuzz Generator". In quality assurance and testing, the same approach (using unexpected data or syntax) has been called robustness testing, syntax testing or negative testing. Even white-noise testing can be thought of as fuzzing.

File formats and network protocols are the most common targets of fuzz testing, but any type of program input can be fuzzed. Interesting inputs include environment variables, keyboard and mouse events, and sequences of API calls. Even items not normally considered "input" can be fuzzed, such as the contents of databases, shared memory, or the precise interleaving of threads.

For the purpose of security, input that crosses a trust boundary is often the most interesting. For example, it is more important to fuzz code that handles the upload of a

file by any user than it is to fuzz the code that parses a configuration file that is accessible only to a privileged user.

Uses

Fuzz testing is often used in large software development projects that employ black-box testing. These projects usually have a budget to develop test tools, and fuzz testing is one of the techniques which offers a high benefit to cost ratio.

However, fuzz testing is not a substitute for exhaustive testing or formal methods: it can only provide a random sample of the system's behavior, and in many cases passing a fuzz test may only demonstrate that a piece of software can handle exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can only be regarded as an assurance of overall quality rather than a bug-finding tool.

As a gross measurement of reliability, fuzzing can suggest which parts of a program should get special attention, in the form of a code audit, application of static analysis, or partial rewrites.

Techniques

The simplest form of fuzzing technique is sending a stream of random bits to software, either as command line options, randomly mutated protocol packets, or as events. The oldest folklore on testing similar to fuzzing tells about The Monkey, from before 1983, which used journaling hooks to feed random events to Macintosh applications. The first command line fuzzer originated from Prof. Barton Miller's group at the University of Wisconsin in 1988. This technique of random inputs still continues to be a powerful tool to find bugs in command-line applications, network protocols, and GUI-based applications and services. Another common technique that is easy to implement is mutating existing input (e.g. files from a test suite) by flipping bits at random or moving blocks of the file around. However, the most successful fuzzers have detailed understanding of the format or protocol being tested.

The understanding can be based on a specification. A specification-based fuzzer involves writing the entire array of specifications into the tool, and then using model-based test generation techniques in walking through the specifications and adding anomalies in the data contents, structures, messages, and sequences. This "smart fuzzing" technique is also known as robustness testing, syntax testing, grammar testing, and (input) fault injection. The protocol awareness can also be created heuristically from examples using a tool such as Sequitur. These fuzzers can *generate* test cases from scratch, or they can *mutate* examples from test suites or real life. They can concentrate on *valid* or *invalid* input, with *mostly-valid* input tending to trigger the "deepest" error cases.

There are two limitations of protocol-based fuzzing based on protocol implementations of published specifications: 1) Testing cannot proceed until the specification is relatively mature, since a specification is a prerequisite for writing such a fuzzer; and 2) Many

useful protocols are proprietary, or involve proprietary extensions to published protocols. If fuzzing is based only on published specifications, test coverage for new or proprietary protocols will be limited or nonexistent.

Fuzz testing can be combined with other testing techniques. **White-box fuzzing** uses symbolic execution and constraint solving. **Evolutionary fuzzing** leverages feedback from code coverage, effectively automating the approach of *exploratory testing*.

Types of bugs found

Straight-up failures such as crashes, assertion failures, and memory leaks are easy to detect. The use of a memory debugger can help find bugs too subtle to always crash.

Fuzz testing is especially useful against large applications, where any bug affecting memory safety is likely to be a severe vulnerability. It is these security concerns that motivate the development of most fuzzers.

Since fuzzing often generates invalid input, it is especially good at testing error-handling routines, which are important for software that does not control its input. As such, simple fuzzing can be thought of as a way to automate negative testing. More sophisticated fuzzing tests more "main-line" code, along with error paths deep within it.

Fuzzing can also find some types of "correctness" bugs. For example, it can be used to find incorrect-serialization bugs by complaining whenever a program's serializer emits something that the same program's parser rejects. It can also find unintentional differences between two versions of a program or between two implementations of the same specification.

Reproduction and isolation

As a practical matter, developers need to reproduce errors in order to fix them. For this reason, almost all fuzz testing makes a record of the data it manufactures, usually before applying it to the software, so that if the computer fails dramatically, the test data is preserved. If the fuzz stream is pseudo-random number generated it may be easier to store the seed value to reproduce the fuzz attempt.

Once a bug found through fuzzing is reproduced, it is often desirable to produce a simple test case to make the issue easier to understand and debug. A simple testcase may also be faster and therefore more suitable for inclusion in a test suite that is run frequently. It can even help to hide the way in which the bug was found. Some fuzzers are designed to work well with testcase reduction programs such as Delta or Lithium.

Advantages and disadvantages

The main problem with fuzzing to find program faults is that it generally only finds very simple faults. The computational complexity of the software testing problem is of

exponential order ($O(c^n),\ c > 1$) and every fuzzer takes shortcuts to find something interesting in a timeframe that a human cares about. A primitive fuzzer may have poor code coverage; for example, if the input includes a checksum which is not properly updated to match other random changes, only the checksum validation code will be verified. Code coverage tools are often used to estimate how "well" a fuzzer works, but these are only guidelines to fuzzer quality. Every fuzzer can be expected to find a different set of bugs.

On the other hand, bugs found using fuzz testing are sometimes severe, exploitable bugs that could be used by a real attacker. This has become more common as fuzz testing has become more widely known, as the same techniques and tools are now used by attackers to exploit deployed software. This is a major advantage over binary or source auditing, or even fuzzing's close cousin, fault injection, which often relies on artificial fault conditions that are difficult or impossible to exploit.

The randomness of inputs used in fuzzing is often seen as a disadvantage, as catching a boundary value condition with random inputs is highly unlikely.

Fuzz testing enhances software security and software safety because it often finds odd oversights and defects which human testers would fail to find, and even careful human test designers would fail to create tests for.

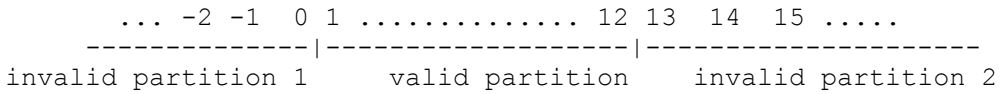
Frameworks

- Powerfuzzer - GPLv3, Python. Fuzz test HTTP targets.
- OWASP JBroFuzz - web server/web application fuzz tester.
- Sulley - GPLv2, Python. Fuzz targets unclear.
- Peach Fuzzing Platform - C# .NET. File/network fuzzer.
- SPIKE - GPL, C. Linux only, focuses on network protocol fuzzing. Lots of examples, but unmaintained since 2004.
- Evolutionary Fuzzing System- GPLv2, Python. Aims to discover protocols automatically based on feedback from fuzz target.
- Tarantula - MIT, Ruby. Fuzz tests Ruby on Rails applications.

Equivalence partitioning

Equivalence partitioning is a software testing technique that divides the input data of a software unit into partitions of data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once. This technique tries to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed.

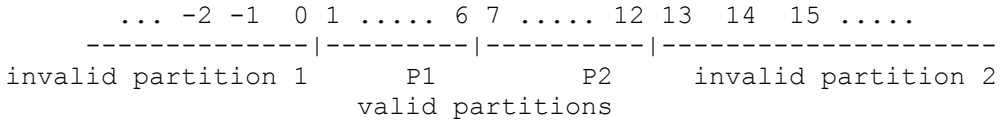
In rare cases equivalence partitioning is also applied to outputs of a software component, typically it is applied to the inputs of a tested component. The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object. An input has certain ranges which are valid and other ranges which are invalid. Invalid data here does not mean that the data is incorrect, it means that this data lies outside of specific partition. This may be best explained by the example of a function which takes a parameter "month". The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition. In this example there are two further partitions of invalid ranges. The first invalid partition would be <= 0 and the second invalid partition would be >= 13.



The testing theory related to equivalence partitioning says that only one test case of each partition is needed to evaluate the behaviour of the program for the related partition. In other words it is sufficient to select one test case out of each partition to check the behaviour of the program. To use more or even all test cases of a partition will not find new faults in the program. The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.

An additional effect of applying this technique is that you also find the so called "dirty" test cases. An inexperienced tester may be tempted to use as test cases the input data 1 to 12 for the month and forget to select some out of the invalid partitions. This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

The tendency is to relate equivalence partitioning to so called black box testing which is strictly checking a software component at its interface, without consideration of internal structures of the software. But having a closer look at the subject there are cases where it applies to grey box testing as well. Imagine an interface to a component which has a valid range between 1 and 12 like the example above. However internally the function may have a differentiation of values between 1 and 6 and the values between 7 and 12. Depending upon the input value the software internally will run through different paths to perform slightly different actions. Regarding the input and output interfaces to the component this difference will not be noticed, however in your grey-box testing you would like to make sure that both paths are examined. To achieve this it is necessary to introduce additional equivalence partitions which would not be needed for black-box testing. For this example this would be:



To check for the expected results you would need to evaluate some internal intermediate values rather than the output interface. It is not necessary that we should use multiple values from each partition. In the above scenario we can take -2 from invalid partition 1, 6 from valid partition, and 15 from invalid partition 2.

Equivalence partitioning is not a stand alone method to determine test cases. It has to be supplemented by boundary value analysis. Having determined the partitions of possible inputs the method of boundary value analysis has to be applied to select the most effective test cases out of these partitions.

Chapter 5

Unit Testing and Integration Testing

Unit testing

In computer programming, **unit testing** is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers.

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects, fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

Benefits

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle.

Facilitates change

Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.

Readily-available unit tests make it easy for the programmer to check whether a piece of code is still working properly.

In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests. Integration testing cannot be fully automated and thus still relies heavily on human testers.

Documentation

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

By contrast, ordinary narrative documentation is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g., design changes, feature creep, relaxed practices in keeping documents up-to-date).

Design

When software is developed using a test-driven approach, the unit test may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour. The following Java example will help illustrate this point.

Here is a test class that specifies a number of elements of the implementation. First, that there must be an interface called Adder, and an implementing class with a zero-argument constructor called AdderImpl. It goes on to assert that the Adder interface should have a method called add, with two integer parameters, which returns another integer. It also specifies the behaviour of this method for a small range of values.

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        assert(adder.add(0, 0) == 0);
        assert(adder.add(-1, -2) == -3);
        assert(adder.add(-1, 1) == 0);
```

```
        assert(adder.add(1234, 988) == 2222);
    }
}
```

In this case the unit test, having been written first, acts as a design document specifying the form and behaviour of a desired solution, but not the implementation details, which are left for the programmer. Following the "do the simplest thing that could possibly work" practice, the easiest solution that will make the test pass is shown below.

```
interface Adder {
    int add(int a, int b);
}
class AdderImpl implements Adder {
    int add(int a, int b) {
        return a + b;
    }
}
```

Unlike other diagram-based design methods, using a unit-test as a design has one significant advantage. The design document (the unit-test itself) can be used to verify that the implementation adheres to the design. With the unit-test design method, the tests will never pass if the developer does not implement the solution according to the design.

It is true that unit testing lacks some of the accessibility of a diagram, but UML diagrams are now easily generated for most modern languages by free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

Separation of interface from implementation

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when test cases fail, makes it less clear which component is causing the failure.

Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher quality unit that is also more maintainable.

Unit testing limitations

Testing cannot be expected to catch every error in the program: it is impossible to evaluate every execution path in all but the most trivial programs. The same is true for unit testing. Additionally, unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors (such as functions performed across multiple units, or non-functional test areas such as performance). Unit testing should be done in conjunction with other software testing activities. Like all forms of software testing, unit tests can only show the presence of errors; they cannot show the absence of errors.

Software testing is a combinatorial problem. For example, every boolean decision statement requires at least two tests: one with an outcome of "true" and one with an outcome of "false". As a result, for every line of code written, programmers often need 3 to 5 lines of test code. This obviously takes time and its investment may not be worth the effort. There are also many problems that cannot easily be tested at all – for example those that are nondeterministic or involve multiple threads. In addition, writing code for a unit test is as likely to be at least as buggy as the code it is testing. Fred Brooks in The Mythical Man-Month quotes: *never take two chronometers to sea. Always take one or three.* Meaning, if two chronometers contradict, how do you know which one is correct?

To obtain the intended benefits from unit testing, rigorous discipline is needed throughout the software development process. It is essential to keep careful records not only of the tests that have been performed, but also of all changes that have been made to the source code of this or any other unit in the software. Use of a version control system is essential. If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.

It is also essential to implement a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately. If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the test suite.

Applications

Extreme Programming

Unit testing is the cornerstone of Extreme Programming, which relies on an automated unit testing framework. This automated unit testing framework can be either third party, e.g., xUnit, or created within the development group.

Extreme Programming uses the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement isn't implemented yet, or because it

intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, pass.

Most code in a system is unit tested, but not necessarily all paths through the code. Extreme Programming mandates a "test everything that can possibly break" strategy, over the traditional "test every execution path" method. This leads developers to develop fewer tests than classical methods, but this isn't really a problem, more a restatement of fact, as classical methods have rarely ever been followed methodically enough for all execution paths to have been thoroughly tested. Extreme Programming simply recognizes that testing is rarely exhaustive (because it is often too expensive and time-consuming to be economically viable) and provides guidance on how to effectively focus limited resources.

Crucially, the test code is considered a first class project artifact in that it is maintained at the same quality as the implementation code, with all duplication removed. Developers release unit testing code to the code repository in conjunction with the code it tests. Extreme Programming's thorough unit testing allows the benefits mentioned above, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. These unit tests are also constantly run as a form of regression test.

Techniques

Unit testing is commonly automated, but may still be performed manually. The IEEE does not favor one over the other. A manual approach to unit testing may employ a step-by-step instructional document. Nevertheless, the objective in unit testing is to isolate a unit and validate its correctness. Automation is efficient for achieving this, and enables the many benefits listed here. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing.

To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it was originally created. Testing in such an isolated manner reveals unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

Using an automation framework, the developer codes criteria into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. Depending upon the severity of a failure, the framework may halt subsequent testing.

As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the best solution may emerge.

Unit testing frameworks

Unit testing frameworks are most often third-party products that are not distributed as part of the compiler suite. They help simplify the process of unit testing, having been developed for a wide variety of languages. Some examples of frameworks are XUnit, and PHPUnit.

It is generally possible to perform unit testing without the support of a specific framework by writing client code that exercises the units under test and uses assertions, exception handling, or other control flow mechanisms to signal failure. Unit testing without a framework is valuable in that there is a barrier to entry for the adoption of unit testing; having scant unit tests is hardly better than having none at all, whereas once a framework is in place, adding unit tests becomes relatively easy. In some frameworks many advanced unit test features are missing or must be hand-coded.

Language-level unit testing support

Some programming languages support unit testing directly (Eg. Java). Their grammar allows the direct declaration of unit tests without importing a library (whether third party or standard). Additionally, the boolean conditions of the unit tests can be expressed in the same syntax as boolean expressions used in non-unit test code, such as what is used for `if` and `while` statements.

Languages that directly support unit testing include:

- Cobra
- D

Integration testing

Integration testing (sometimes called Integration and Testing, abbreviated "I&T") is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Purpose

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using Black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Some different types of integration testing are big bang, top-down, and bottom-up.

Big Bang

In this approach, all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. The Big Bang method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.

A type of Big Bang Integration testing is called **Usage Model testing**. Usage Model testing can be used in both software and hardware integration testing. The basis behind this type of integration testing is to run user-like workloads in integrated user-like environments. In doing the testing in this manner, the environment is proofed, while the individual components are proofed indirectly through their use. Usage Model testing takes an optimistic approach to testing, because it expects to have few problems with the individual components. The strategy relies heavily on the component developers to do the isolated unit testing for their product. The goal of the strategy is to avoid redoing the testing done by the developers, and instead flesh out problems caused by the interaction of the components in the environment. For integration testing, Usage Model testing can be more efficient and provides better test coverage than traditional focused functional integration testing. To be more efficient and accurate, care must be used in defining the user-like workloads for creating realistic scenarios in exercising the environment. This gives that the integrated environment will work as expected for the target customers.

Top-down and Bottom-up

Bottom Up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

Sandwich Testing is an approach to combine top down testing with bottom up testing.

The main advantage of the Bottom-Up approach is that bugs are more easily found. With Top-Down, it is easier to find a missing branch link.

Limitations

Any conditions not stated in specified integration tests, outside of the confirmation of the execution of design items, will generally not be tested.

Chapter 6

GUI Software Testing and Software Performance Testing

GUI software testing

In computer science, **GUI software testing** is the process of testing a product that uses a graphical user interface, to ensure it meets its written specifications. This is normally done through the use of a variety of test cases.

Test Case Generation

To generate a 'good' set of test cases, the test designers must be certain that their suite covers all the functionality of the system and also has to be sure that the suite fully exercises the GUI itself. The difficulty in accomplishing this task is twofold: one has to deal with domain size and then one has to deal with sequences. In addition, the tester faces more difficulty when they have to do regression testing.

The size problem can be easily illustrated. Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A very small program such as Microsoft WordPad has 325 possible GUI operations. In a large program, the number of operations can easily be an order of magnitude larger.

The second problem is the sequencing problem. Some functionality of the system may only be accomplishable by following some complex sequence of GUI events. For example, to open a file a user may have to click on the File Menu and then select the Open operation, and then use a dialog box to specify the file name, and then focus the application on the newly opened window. Obviously, increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

Regression testing becomes a problem with GUIs as well. This is because the GUI may change significantly across versions of the application, even though the underlying

application may not. A test designed to follow a certain path through the GUI may not be able to follow that path since a button, menu item, or dialog may have changed location or appearance.

These issues have driven the GUI testing problem domain towards automation. Many different techniques have been proposed to automatically generate test suites that are complete and that simulate user behavior.

Most of the techniques used to test GUIs attempt to build on techniques previously used to test CLI programs. However, most of these have scaling problems when they are applied to GUI's. For example, Finite State Machine-based modeling — where a system is modeled as a finite state machine and a program is used to generate test cases that exercise all states — can work well on a system that has a limited number of states but may become overly complex and unwieldy for a GUI.

Planning and artificial intelligence

A novel approach to test suite generation, adapted from a **CLI** technique involves using a planning system. Planning is a well-studied technique from the artificial intelligence (AI) domain that attempts to solve problems that involve four parameters:

- an initial state,
- a goal state,
- a set of operators, and
- a set of objects to operate on.

Planning systems determine a path from the initial state to the goal state by using the operators. An extremely simple planning problem would be one where you had two words and one operation called 'change a letter' that allowed you to change one letter in a word to another letter – the goal of the problem would be to change one word into another.

For GUI testing, the problem is a bit more complex. In the authors used a planner called IPP to demonstrate this technique. The method used is very simple to understand. First, the systems UI is analyzed to determine what operations are possible. These operations become the operators used in the planning problem. Next an initial system state is determined. Next a goal state is determined that the tester feels would allow exercising of the system. Lastly the planning system is used to determine a path from the initial state to the goal state. This path becomes the test plan.

Using a planner to generate the test cases has some specific advantages over manual generation. A planning system, by its very nature, generates solutions to planning problems in a way that is very beneficial to the tester:

1. The plans are always valid. What this means is that the output of the system can be one of two things, a valid and correct plan that uses the operators to attain the

goal state or no plan at all. This is beneficial because much time can be wasted when manually creating a test suite due to invalid test cases that the tester thought would work but didn't.

2. A planning system pays attention to order. Often to test a certain function, the test case must be complex and follow a path through the GUI where the operations are performed in a specific order. When done manually, this can lead to errors and also can be quite difficult and time consuming to do.
3. Finally, and most importantly, a planning system is goal oriented. What this means and what makes this fact so important is that the tester is focusing test suite generation on what is most important, testing the functionality of the system.

When manually creating a test suite, the tester is more focused on how to test a function (i. e. the specific path through the GUI). By using a planning system, the path is taken care of and the tester can focus on what function to test. An additional benefit of this is that a planning system is not restricted in any way when generating the path and may often find a path that was never anticipated by the tester. This problem is a very important one to combat.

Another interesting method of generating GUI test cases uses the theory that good GUI test coverage can be attained by simulating a novice user. One can speculate that an expert user of a system will follow a very direct and predictable path through a GUI and a novice user would follow a more random path. The theory therefore is that if we used an expert to test the GUI, many possible system states would never be achieved. A novice user, however, would follow a much more varied, meandering and unexpected path to achieve the same goal so it's therefore more desirable to create test suites that simulate novice usage because they will test more.

The difficulty lies in generating test suites that simulate 'novice' system usage. Using Genetic algorithms is one proposed way to solve this problem. Novice paths through the system are not random paths. First, a novice user will learn over time and generally won't make the same mistakes repeatedly, and, secondly, a novice user is not analogous to a group of monkeys trying to type Hamlet, but someone who is following a plan and probably has some domain or system knowledge.

Genetic algorithms work as follows: a set of 'genes' are created randomly and then are subjected to some task. The genes that complete the task best are kept and the ones that don't are discarded. The process is again repeated with the surviving genes being replicated and the rest of the set filled in with more random genes. Eventually one gene (or a small set of genes if there is some threshold set) will be the only gene in the set and is naturally the best fit for the given problem.

For the purposes of the GUI testing, the method works as follows. Each gene is essentially a list of random integer values of some fixed length. Each of these genes represents a path through the GUI. For example, for a given tree of widgets, the first value in the gene (each value is called an allele) would select the widget to operate on, the following alleles would then fill in input to the widget depending on the number of

possible inputs to the widget (for example a pull down list box would have one input…the selected list value). The success of the genes are scored by a criterion that rewards the best 'novice' behavior.

The system to do this testing described in can be extended to any windowing system but is described on the X window system. The X Window system provides functionality (via XServer and the editors' protocol) to dynamically send GUI input to and get GUI output from the program without directly using the GUI. For example, one can call XSendEvent() to simulate a click on a pull-down menu, and so forth. This system allows researchers to automate the gene creation and testing so for any given application under test, a set of novice user test cases can be created.

Event Flow Graphs

A 2007 development in the field of automated GUI testing is a new graph model called the event-flow graph that represents events and event interactions. In much the same way as a control-flow graph represents all possible execution paths in a program, and a data-flow graph represents all possible definitions and uses of a memory location, the event-flow model represents all possible sequences of events that can be executed on the GUI. A GUI is decomposed into a hierarchy of modal dialogues; this hierarchy is represented as an integration tree; each modal dialogue is represented as an event-flow graph that shows all possible event execution paths in the dialogue.

Running the test cases

At first the strategies were migrated and adapted from the CLI testing strategies. A popular method used in the CLI environment is capture/playback. Capture playback is a system where the system screen is "captured" as a bitmapped graphic at various times during system testing. This capturing allowed the tester to "play back" the testing process and compare the screens at the output phase of the test with expected screens. This validation could be automated since the screens would be identical if the case passed and different if the case failed.

Using capture/playback worked quite well in the CLI world but there are significant problems when one tries to implement it on a GUI-based system . The most obvious problem one finds is that the screen in a GUI system may look different while the state of the underlying system is the same, making automated validation extremely difficult. This is because a GUI allows graphical objects to vary in appearance and placement on the screen. Fonts may be different, window colors or sizes may vary but the system output is basically the same. This would be obvious to a user, but not obvious to an automated validation system.

To combat this and other problems, testers have gone 'under the hood' and collected GUI interaction data from the underlying windowing system . By capturing the window 'events' into logs the interactions with the system are now in a format that is decoupled from the appearance of the GUI. Now, only the event streams are captured. There is some

filtering of the event streams necessary since the streams of events are usually very detailed and most events aren't directly relevant to the problem. This approach can be made easier by using an MVC architecture for example and making the view (i. e. the GUI here) as simple as possible while the model and the controller hold all the logic. Another approach is to use the software's built-in assistive technology, to use an HTML interface or a three-tier architecture that makes it also possible to better separate the user interface from the rest of the application.

Another way to run tests on a GUI is to build a driver into the GUI so that commands or events can be sent to the software from another program. This method of directly sending events to and receiving events from a system is highly desirable when testing, since the input and output testing can be fully automated and user error is eliminated.

Software performance testing

In software engineering, **performance testing** is testing that is performed, to determine how fast some aspect of a system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage.

Performance testing is a subset of Performance engineering, an emerging computer science practice which strives to build performance into the design and architecture of a system, prior to the onset of actual coding effort.

Performance Testing Sub-Genres

Load Testing

Load testing is the simplest form of performance testing. A load test is usually conducted to understand the behavior of the application under a specific expected load. This load can be the expected concurrent number of users on the application performing a specific number of transactions within the set duration. This test will give out the response times of all the important business critical transactions. If the database, application server, etc. are also monitored, then this simple test can itself point towards any bottlenecks in the application software.

Stress Testing

Stress testing is normally used to understand the upper limits of capacity within the application landscape. This kind of test is done to determine the application's robustness

in terms of extreme load and helps application administrators to determine if the application will perform sufficiently if the current load goes well above the expected maximum.

Endurance Testing (Soak Testing)

Endurance testing is usually done to determine if the application can sustain the continuous expected load. During endurance tests, memory utilization is monitored to detect potential leaks. Also important, but often overlooked is performance degradation. That is, to ensure that the throughput and/or response times after some long period of sustained activity are as good or better than at the beginning of the test.

Spike Testing

Spike testing, as the name suggests is done by spiking the number of users and understanding the behavior of the application; whether performance will suffer, the application will fail, or it will be able to handle dramatic changes in load.

Configuration Testing

Configuration testing is another variation on traditional performance testing. Rather than testing for performance from the perspective of load you are testing the effects of configuration changes in the application landscape on application performance and behaviour. A common example would be experimenting with different methods of load-balancing.

Isolation Testing

Isolation testing is unique to performance testing but a term used to describe repeating a test execution that resulted in an application problem. Often used to isolate and confirm the fault domain.

Setting performance goals

Performance testing can serve different purposes.

- It can demonstrate that the system meets performance criteria.
- It can compare two systems to find which performs better.
- Or it can measure what parts of the system or workload causes the system to perform badly.

Many performance tests are undertaken without due consideration to the setting of realistic performance goals. The first question from a business perspective should always be "why are we performance testing?". These considerations are part of the business case of the testing. Performance goals will differ depending on the application technology and purpose however they should always include some of the following:-

Concurrency / Throughput

If an application identifies end-users by some form of login procedure then a concurrency goal is highly desirable. By definition this is the largest number of concurrent application users that the application is expected to support at any given moment. The work-flow of your scripted transaction may impact true application concurrency especially if the iterative part contains the Login & Logout activity

If your application has no concept of end-users then your performance goal is likely to be based on a maximum throughput or transaction rate. A common example would be casual browsing of a web site.

Server response time

This refers to the time taken for one application node to respond to the request of another. A simple example would be a HTTP 'GET' request from browser client to web server. In terms of response time this is what all load testing tools actually measure. It may be relevant to set server response time goals between all nodes of the application landscape.

Render response time

A difficult thing for load testing tools to deal with as they generally have no concept of what happens within a node apart from recognising a period of time where there is no activity 'on the wire'. To measure render response time it is generally necessary to include functional test scripts as part of the performance test scenario which is a feature not offered by many load testing tools.

Performance specifications

It is critical to detail performance specifications (requirements) and document them in any performance test plan. Ideally, this is done during the requirements development phase of any system development project, prior to any design effort.

However, **performance testing** is frequently not performed against a specification i.e. no one will have expressed what the maximum acceptable response time for a given population of users should be. Performance testing is frequently used as part of the process of performance profile tuning. The idea is to identify the “weakest link” – there is inevitably a part of the system which, if it is made to respond faster, will result in the overall system running faster. It is sometimes a difficult task to identify which part of the system represents this critical path, and some test tools include (or can have add-ons that provide) instrumentation that runs on the server (agents) and report transaction times, database access times, network overhead, and other server monitors, which can be analyzed together with the raw performance statistics. Without such instrumentation one might have to have someone crouched over Windows Task Manager at the server to see how much CPU load the performance tests are generating (assuming a Windows system is under test).

There is an apocryphal story of a company that spent a large amount optimizing their software without having performed a proper analysis of the problem. They ended up rewriting the system's 'idle loop', where they had found the system spent most of its time, but even having the most efficient idle loop in the world obviously didn't improve overall performance one iota!

Performance testing can be performed across the web, and even done in different parts of the country, since it is known that the response times of the internet itself vary regionally. It can also be done in-house, although routers would then need to be configured to introduce the lag what would typically occur on public networks. Loads should be introduced to the system from realistic points. For example, if 50% of a system's user base will be accessing the system via a 56K modem connection and the other half over a T1, then the load injectors (computers that simulate real users) should either inject load over the same connections (ideal) or simulate the network latency of such connections, following the same user profile.

It is always helpful to have a statement of the likely peak numbers of users that might be expected to use the system at peak times. If there can also be a statement of what constitutes the maximum allowable 95 percentile response time, then an injector configuration could be used to test whether the proposed system met that specification.

Questions to ask

Performance specifications should ask the following questions, at a minimum:

- In detail, what is the performance test scope? What subsystems, interfaces, components, etc. are in and out of scope for this test?
- For the user interfaces (UI's) involved, how many concurrent users are expected for each (specify peak vs. nominal)?
- What does the target system (hardware) look like (specify all server and network appliance configurations)?
- What is the Application Workload Mix of each application component? (for example: 20% login, 40% search, 30% item select, 10% checkout).
- What is the System Workload Mix? [Multiple workloads may be simulated in a single performance test] (for example: 30% Workload A, 20% Workload B, 50% Workload C)
- What are the time requirements for any/all backend batch processes (specify peak vs. nominal)?

Pre-requisites for Performance Testing

A stable build of the application which must resemble the Production environment as close to possible.

The performance testing environment should not be clubbed with User acceptance testing (UAT) or development environment. This is dangerous as if an UAT or Integration test or other tests are going on in the same environment, then the results obtained from the performance testing may not be reliable. As a best practice it is always advisable to have a separate performance testing environment resembling the production environment as much as possible.

Test conditions

In performance testing, it is often crucial (and often difficult to arrange) for the test conditions to be similar to the expected actual use. This is, however, not entirely possible in actual practice. The reason is that the workloads of production systems have a random nature, and while the test workloads do their best to mimic what may happen in the production environment, it is impossible to exactly replicate this workload variability - except in the most simple system.

Loosely-coupled architectural implementations (e.g.: SOA) have created additional complexities with performance testing. Enterprise services or assets (that share a common infrastructure or platform) require coordinated performance testing (with all consumers creating production-like transaction volumes and load on shared infrastructures or platforms) to truly replicate production-like states. Due to the complexity and financial and time requirements around this activity, some organizations now employ tools that can monitor and create production-like conditions (also referred as "noise") in their performance testing environments (PTE) to understand capacity and resource requirements and verify / validate quality attributes.

Timing

It is critical to the cost performance of a new system, that performance test efforts begin at the inception of the development project and extend through to deployment. The later a performance defect is detected, the higher the cost of remediation. This is true in the case of functional testing, but even more so with performance testing, due to the end-to-end nature of its scope.

Tools

In the diagnostic case, software engineers use tools such as profilers to measure what parts of a device or software contributes most to the poor performance or to establish throughput levels (and thresholds) for maintained acceptable response time.

Myths of Performance Testing

Some of the very common myths are given below.
1. Performance Testing is done to break the system.

Stress Testing is done to understand the break point of the system. Otherwise normal load testing is generally done to understand the behavior of the application under the expected user load. Depending on other requirements, such as expectation of spike load, continued load for an extended period of time would demand spike, endurance soak or stress testing.

2. Performance Testing should only be done after the System Integration Testing

Although this is mostly the norm in the industry, performance testing can also be done while the initial development of the application is taking place. This kind of approach is known as the **Early Performance Testing**. This approach would ensure a holistic development of the application keeping the performance parameters in mind. Thus the finding of a performance bug just before the release of the application and the cost involved in rectifying the bug is reduced to a great extent.

3. Performance Testing only involves creation of scripts and any application changes would cause a simple refactoring of the scripts.

Performance Testing in itself is an evolving science in the Software Industry. Scripting itself although important, is only one of the components of the performance testing. The major challenge for any performance tester is to determine the type of tests needed to execute and analyzing the various performance counters to determine the performance bottleneck.

The other segment of the myth concerning the change in application would result only in little refactoring in the scripts is also untrue as any form of change on the UI especially in the Web protocol would entail complete re-development of the scripts from scratch. This problem becomes bigger if the protocols involved include Web Services, Siebel, Web Click n Script, Citrix, SAP.

Technology

Performance testing technology employs one or more PCs or Unix servers to act as injectors – each emulating the presence of numbers of users and each running an automated sequence of interactions (recorded as a script, or as a series of scripts to emulate different types of user interaction) with the host whose performance is being tested. Usually, a separate PC acts as a test conductor, coordinating and gathering metrics from each of the injectors and collating performance data for reporting purposes. The usual sequence is to ramp up the load – starting with a small number of virtual users and increasing the number over a period to some maximum. The test result shows how the performance varies with the load, given as number of users vs response time. Various tools, are available to perform such tests. Tools in this category usually execute a suite of tests which will emulate real users against the system. Sometimes the results can reveal oddities, e.g., that while the average response time might be acceptable, there are outliers of a few key transactions that take considerably longer to complete – something that might be caused by inefficient database queries, pictures etc.

Performance testing can be combined with stress testing, in order to see what happens when an acceptable load is exceeded –does the system crash? How long does it take to recover if a large load is reduced? Does it fail in a way that causes collateral damage?

Analytical Performance Modeling is a method to model the behaviour of an application in a spreadsheet. The model is fed with measurements of transaction resource demands (CPU, disk I/O, LAN, WAN), weighted by the transaction-mix (business transactions per hour). The weighted transaction resource demands are added-up to obtain the hourly resource demands and divided by the hourly resource capacity to obtain the resource loads. Using the responsetime formula (R=S/(1-U), R=responsetime, S=servicetime, U=load), responsetimes can be calculated and calibrated with the results of the performance tests. Analytical performance modelling allows evaluation of design options and system sizing based on actual or anticipated business usage. It is therefore much faster and cheaper than performance testing, though it requires thorough understanding of the hardware platforms.

Tasks to undertake

Tasks to perform such a test would include:

- Decide whether to use internal or external resources to perform the tests, depending on inhouse expertise (or lack thereof)
- Gather or elicit performance requirements (specifications) from users and/or business analysts
- Develop a high-level plan (or project charter), including requirements, resources, timelines and milestones
- Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc.)
- Choose test tool(s)
- Specify test data needed and charter effort (often overlooked, but often the death of a valid performance test)
- Develop proof-of-concept scripts for each application/component under test, using chosen test tools and strategies
- Develop detailed performance test project plan, including all dependencies and associated timelines
- Install and configure injectors/controller
- Configure the test environment (ideally identical hardware to the production platform), router configuration, quiet network (we don't want results upset by other users), deployment of server instrumentation, database test sets developed, etc.
- Execute tests – probably repeatedly (iteratively) in order to see whether any unaccounted for factor might affect the results
- Analyze the results - either pass/fail, or investigation of critical path and recommendation of corrective action

Chapter 7

Regression Testing and Acceptance Testing

Regression testing

Regression testing is any type of software testing that seeks to uncover software errors after changes to the program (e.g. bugfixes or new functionality) have been made, by retesting the program. The intent of regression testing is to assure that a change, such as a bugfix, did not introduce new bugs. Regression testing can be used to test the system efficiently by systematically selecting the appropriate minimum suite of tests needed to adequately cover the affected change. *Common methods* of regression testing include rerunning previously run tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged. "One of the main reasons for regression testing is that it's often extremely difficult for a programmer to figure out how a change in one part of the software will echo in other parts of the software." This is done by comparing results of previous tests to results of the current tests being run.

Background

Experience has shown that as software is fixed, emergence of new and/or reemergence of old faults is quite common. Sometimes reemergence occurs because a fix gets lost through poor revision control practices (or simple human error in revision control). Often, a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Frequently, a fix for a problem in one area inadvertently causes a software bug in another area. Finally, it is often the case that when some feature is redesigned, some of the same mistakes that were made in the original implementation of the feature were made in the redesign.

Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a test suite contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set

up automated systems to automatically re-run all regression tests at specified intervals and report any failures (which could imply a regression or an out-of-date test). Common strategies are to run such a system after every successful compile (for small projects), every night, or once a week. Those strategies can be automated by an external tool, such as BuildBot or Hudson.

Regression testing is an integral part of the extreme programming software development method. In this method, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle.

In the corporate world, regression testing has traditionally been performed by a software quality assurance team after the development team has completed work. However, defects found at this stage are the most costly to fix. This problem is being addressed by the rise of unit testing. Although developers have always written test cases as part of the development cycle, these test cases have generally been either functional tests or unit tests that verify only intended outcomes. Developer testing compels a developer to focus on unit testing and to include both positive and negative test cases.

Uses

Regression testing can be used not only for testing the *correctness* of a program, but often also for tracking the quality of its output. For instance, in the design of a compiler, regression testing could track the code size, simulation time and time of the test suite cases.

Regression testing should be part of a test plan. Regression testing can be automated.

"Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such *regression testing* must indeed approximate this theoretical idea, and it is very costly."

– Fred Brooks, The Mythical Man Month, p 122

Regression tests can be broadly categorized as functional tests or unit tests. Functional tests exercise the complete program with various inputs. Unit tests exercise individual functions, subroutines, or object methods. Both functional testing tools and unit testing tools tend to be third party products that are not part of the compiler suite, and both tend to be automated. Functional tests may be a scripted series of program inputs, possibly even an automated mechanism for controlling mouse movements. Unit tests may be separate functions within the code itself, or driver layer that links to the code without altering the code being tested.

Acceptance testing

In engineering and its various subdisciplines, **acceptance testing** is black-box testing performed on a system (for example: a piece of software, lots of manufactured mechanical parts, or batches of chemical products) prior to its delivery. It is also known as functional testing, black-box testing, QA testing, application testing, confidence testing, final testing, validation testing, or factory acceptance testing.

Software developers often distinguish acceptance testing by the system provider from acceptance testing by the customer (the user or client) prior to accepting transfer of ownership. In the case of software, acceptance testing performed by the customer is known as user acceptance testing (UAT), end-user testing, site (acceptance) testing, or field (acceptance) testing.

A smoke test is used as an acceptance test prior to introducing a build to the main testing process.

Overview

Acceptance testing generally involves running a suite of tests on the completed system. Each individual test, known as a case, exercises a particular operating condition of the user's environment or feature of the system, and will result in a pass or fail, or boolean, outcome. There is generally no degree of success or failure. The test environment is usually designed to be identical, or as close as possible, to the anticipated user's environment, including extremes of such. These test cases must each be accompanied by test case input data or a formal description of the operational activities (or both) to be performed—intended to thoroughly exercise the specific case—and a formal description of the expected results.

Acceptance Tests/Criteria (in Agile Software Development) are usually created by business customers and expressed in a business domain language. These are high-level tests to test the completeness of a user story or stories 'played' during any sprint/iteration. These tests are created ideally through collaboration between business customers, business analysts, testers and developers, however the business customers (product owners) are the primary owners of these tests. As the user stories pass their acceptance criteria, the business owners can be sure of the fact that the developers are progressing in the right direction about how the application was envisaged to work and so it's essential that these tests include both business logic tests as well as UI validation elements (if need be).

Acceptance test cards are ideally created during sprint planning or iteration planning meeting, before development begins so that the developers have a clear idea of what to develop. Sometimes (due to bad planning!) acceptance tests may span multiple stories (that are not implemented in the same sprint) and there are different ways to test them out during actual sprints. One popular technique is to mock external interfaces or data to

mimic other stories which might not be played out during an iteration (as those stories may have been relatively lower business priority). A user story is not considered complete until the acceptance tests have passed.

Process

The acceptance test suite is run against the supplied input data or using an acceptance test script to direct the testers. Then the results obtained are compared with the expected results. If there is a correct match for every case, the test suite is said to pass. If not, the system may either be rejected or accepted on conditions previously agreed between the sponsor and the manufacturer.

The objective is to provide confidence that the delivered system meets the business requirements of both sponsors and users. The acceptance phase may also act as the final quality gateway, where any quality defects not previously detected may be uncovered.

A principal purpose of acceptance testing is that, once completed successfully, and provided certain additional (contractually agreed) acceptance criteria are met, the sponsors will then sign off on the system as satisfying the contract (previously agreed between sponsor and manufacturer), and deliver final payment.

User acceptance testing

User Acceptance Testing (UAT) is a process to obtain confirmation that a system meets mutually agreed-upon requirements. A Subject Matter Expert (SME), preferably the owner or client of the object under test, provides such confirmation after trial or review. In software development, UAT is one of the final stages of a project and often occurs before a client or customer accepts the new system.

Users of the system perform these tests, which developers derive from the client's contract or the user requirements specification.

Test-designers draw up formal tests and devise a range of severity levels. Ideally the designer of the user acceptance tests should not be the creator of the formal integration and system test cases for the same system, however in some situations this may not be avoided. The UAT acts as a final verification of the required business function and proper functioning of the system, emulating real-world usage conditions on behalf of the paying client or a specific large customer. If the software works as intended and without issues during normal use, one can reasonably extrapolate the same level of stability in production.

User tests, which are usually performed by clients or end-users, do not normally focus on identifying simple problems such as spelling errors and cosmetic problems, nor showstopper defects, such as software crashes; testers and developers previously identify and fix these issues during earlier unit testing, integration testing, and system testing phases.

The results of these tests give confidence to the clients as to how the system will perform in production. There may also be legal or contractual requirements for acceptance of the system.

Q-UAT - Quantified User Acceptance Testing

Quantified User Acceptance Testing (Q-UAT or, more simply, the "Quantified Approach") is a revised Business Acceptance Testing process which aims to provide a smarter and faster alternative to the traditional UAT phase. Depth-testing is carried out against business requirements only at specific planned points in the application or service under test. A reliance on better quality code-delivery from the development/build phase is assumed and a complete understanding of the appropriate business process is a pre-requisite. This methodology - if carried out correctly - results in a quick turnaround against plan, a decreased number of test scenarios which are more complex and wider in breadth than traditional UAT and ultimately the equivalent confidence-level attained via a shorter delivery-window, allowing products/changes to come to market quicker.

The Q-UAT approach depends on a "gated" three-dimensional model. The key concepts are:

1. Linear Testing (LT, the 1st dimension)
2. Recursive Testing (RT, the 2nd dimension)
3. Adaptive Testing (AT, the 3rd dimension).

The four "gates" which conjoin and support the 3-dimensional model act as quality safeguards and include contemporary testing concepts such as:

- Internal Consistency Checks (ICS)
- Major Systems/Services Checks (MSC)
- Realtime/Reactive Regression (RTR).

The Quantified Approach was shaped by the former "guerilla" method of acceptance testing which was itself a response to testing phases which proved too costly to be sustainable for many small/medium-scale projects.

Acceptance testing in Extreme Programming

Acceptance testing is a term used in agile software development methodologies, particularly Extreme Programming, referring to the functional testing of a user story by the software development team during the implementation phase.

The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, whatever it takes to ensure the functionality works. Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed

tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release. A user story is not considered complete until it has passed its acceptance tests. This means that new acceptance tests must be created for each iteration or the development team will report zero progress.

Types of acceptance testing

Typical types of acceptance testing include the following

User acceptance testing
: This may include factory acceptance testing, i.e. the testing done by factory users before the factory is moved to its own site, after which site acceptance testing may be performed by the users at the site.

Operational Acceptance Testing (OAT)
: Also known as operational readiness testing, this refers to the checking done to a system to ensure that processes and procedures are in place to allow the system to be used and maintained. This may include checks done to back-up facilities, procedures for disaster recovery, training for end users, maintenance procedures, and security procedures.

Contract and regulation acceptance testing
: In contract acceptance testing, a system is tested against acceptance criteria as documented in a contract, before the system is accepted. In regulation acceptance testing, a system is tested to ensure it meets governmental, legal and safety standards.

Alpha and beta testing
: Alpha testing takes place at developers' sites, and involves testing of the operational system by internal staff, before it is released to external customers. Beta testing takes place at customers' sites, and involves testing by a group of customers who use the system at their own locations and provide feedback, before the system is released to other customers. The latter is often called “field testing”.

Chapter 8

Security Testing and Test Automation

Security testing

Security testing is a process to determine that an information system protects data and maintains functionality as intended.

The six basic security concepts that need to be covered by security testing are: confidentiality, integrity, authentication, availability, authorization and non-repudiation. Security testing as a term has a number of different meanings and can be completed in a number of different ways. As such a Security Taxonomy helps us to understand these different approaches and meanings by providing a base level to work from.

Confidentiality

- A security measure which protects against the disclosure of information to parties other than the intended recipient that is by no means the only way of ensuring the security.

Integrity

- A measure intended to allow the receiver to determine that the information which it is providing is correct.
- Integrity schemes often use some of the same underlying technologies as confidentiality schemes, but they usually involve adding additional information to a communication to form the basis of an algorithmic check rather than the encoding all of the communication.

Authentication

- It is a type of security testing in which one will enter different combinations of usernames and passwords and will check whether only the authorized people are able to access it or not.
- The process of establishing the identity of the user.

- Authentication can take many forms including but not limited to: passwords, biometrics, radio frequency identification, etc.

Authorization

- The process of determining that a requester is allowed to receive a service or perform an operation.
- Access control is an example of authorization.

Availability

- Assuring information and communications services will be ready for use when expected.
- Information must be kept available to authorized persons when they need it.

Non-repudiation

- A measure intended to prevent the later denial that an action happened, or a communication that took place etc.
- In communication terms this often involves the interchange of authentication information combined with some form of provable time stamp.

Security Testing Taxonomy

Common terms used for the delivery of security testing;

- **Discovery** - The purpose of this stage is to identify systems within scope and the services in use. It is not intended to discover vulnerabilities, but version detection may highlight deprecated versions of software / firmware and thus indicate potential vulnerabilities.

- **Vulnerability Scan** - Following the discovery stage this looks for known security issues by using automated tools to match conditions with known vulnerabilities. The reported risk level is set automatically by the tool with no manual verification or interpretation by the test vendor. This can be supplemented with credential based scanning that looks to remove some common false positives by using supplied credentials to authenticate with a service (such as local windows accounts).

- **Vulnerability Assessment** - This uses discovery and vulnerability scanning to identify security vulnerabilities and places the findings into the context of the environment under test. An example would be removing common false positives from the report and deciding risk levels that should be applied to each report finding to improve business understanding and context.

- **Security Assessment** - Builds upon Vulnerability Assessment by adding manual verification to confirm exposure, but does not include the exploitation of vulnerabilities to gain further access. Verification could be in the form of authorised access to a system to confirm system settings and involve examining logs, system responses, error messages, codes, etc. A Security Assessment is looking to gain a broad coverage of the systems under test but not the depth of exposure that a specific vulnerability could lead to.

- **Penetration Test** - Penetration test simulates an attack by a malicious party. Building on the previous stages and involves exploitation of found vulnerabilities to gain further access. Using this approach will result in an understanding of the ability of an attacker to gain access to confidential information, affect data integrity or availability of a service and the respective impact. Each test is approached using a consistent and complete methodology in a way that allows the tester to use their problem solving abilities, the output from a range of tools and their own knowledge of networking and systems to find vulnerabilities that would/ could not be identified by automated tools. This approach looks at the depth of attack as compared to the Security Assessment approach that looks at the broader coverage.

- **Security Audit** - Driven by an Audit / Risk function to look at a specific control or compliance issue. Characterised by a narrow scope, this type of engagement could make use of any of the earlier approaches discussed (vulnerability assessment, security assessment, penetration test).

- **Security Review** - Verification that industry or internal security standards have been applied to system components or product. This is typically completed through gap analysis and utilises build / code reviews or by reviewing design documents and architecture diagrams. This activity does not utilise any of the earlier approaches (Vulnerability Assessment, Security Assessment, Penetration Test, Security Audit)

Test automation

Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Commonly, test automation involves automating a manual process already in place that uses a formalized testing process.

Overview

Although manual tests may find many defects in a software application, it is a laborious and time consuming process. In addition, it may not be effective in finding certain classes of defects. Test automation is a process of writing a computer program to do testing that would otherwise need to be done manually. Once tests have been automated, they can be run quickly. This is often the most cost effective method for software products that have a long maintenance life, because even minor patches over the lifetime of the application can cause features to break which were working at an earlier point in time.

There are two general approaches to test automation:

- **Code-driven testing**. The public (usually) interfaces to classes, modules, or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.
- Graphical user interface testing. A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

Test automation tools can be expensive, and it is usually employed in combination with manual testing. It can be made cost-effective in the longer term, especially when used repeatedly in regression testing.

One way to generate test cases automatically is model-based testing through use of a model of the system for test case generation but research continues into a variety of alternative methodologies for doing so.

What to automate, when to automate, or even whether one really needs automation are crucial decisions which the testing (or development) team must make. Selecting the correct features of the product for automation largely determines the success of the automation. Automating unstable features or features that are undergoing changes should be avoided.

Code-driven testing

A growing trend in software development is the use of testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) that allow the execution of unit tests to determine whether various sections of the code are acting as expected under various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected.

Code driven test automation is a key feature of Agile software development, where it is known as Test-driven development (TDD). Unit tests are written to define the functionality *before* the code is written. Only when all tests pass is the code considered complete. Proponents argue that it produces software that is both more reliable and less

costly than code that is tested by manual exploration. It is considered more reliable because the code coverage is better, and because it is run constantly during development rather than once at the end of a waterfall development cycle. The developer discovers defects immediately upon making a change, when it is least expensive to fix. Finally, code refactoring is safer; transforming the code into a simpler form with less code duplication, but equivalent behavior, is much less likely to introduce new defects.

Graphical User Interface (GUI) testing

Many test automation tools provide record and playback features that allow users to interactively record user actions and replay them back any number of times, comparing actual results to those expected. The advantage of this approach is that it requires little or no software development. This approach can be applied to any application that has a graphical user interface. However, reliance on these features poses major reliability and maintainability problems. Relabelling a button or moving it to another part of the window may require the test to be re-recorded. Record and playback also often adds irrelevant activities or incorrectly records some activities.

A variation on this type of tool is for testing of web sites. Here, the "interface" is the web page. This type of tool also requires little or no software development. However, such a framework utilizes entirely different techniques because it is reading html instead of observing window events.

Another variation is scriptless test automation that does not use record and playback, but instead builds a model of the application under test and then enables the tester to create test cases by simply editing in test parameters and conditions. This requires no scripting skills, but has all the power and flexibility of a scripted approach. Test-case maintenance is easy, as there is no code to maintain and as the application under test changes the software objects can simply be re-learned or added. It can be applied to any GUI-based software application.

What to test

Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

One must keep satisfying popular requirements when thinking of test automation:

- Platform and OS independence
- Data driven capability (Input Data, Output Data, Meta Data)
- Customizable Reporting (DB Access, crystal reports)
- Easy debugging and logging
- Version control friendly – minimal binary files
- Extensible & Customizable (Open APIs to be able to integrate with other tools)

- Common Driver (For example, in the Java development ecosystem, that means Ant or Maven and the popular IDEs). This enables tests to integrate with the developers' workflows.
- Support unattended test runs for integration with build processes and batch runs. Continuous Integration servers require this.
- Email Notifications (automated notification on failure or threshold levels). This may be the test runner or tooling that executes it.
- Support distributed execution environment (distributed test bed)
- Distributed application support (distributed SUT)

Framework approach in automation

A framework is an integrated system that sets the rules of Automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled to represent a business process. The framework provides the basis of test automation and simplifies the automation effort.

Defining boundaries between automation framework and a testing tool

Tools are specifically designed to target some particular test environment. Such as: Windows automation tool, web automation tool etc. It serves as driving agent for an automation process. However, automation framework is not a tool to perform some specific task, but is an infrastructure that provides the solution where different tools can plug itself and do their job in an unified manner. Hence providing a common platform to the automation engineer doing their job.

1. [An Ideal framework]

There are various types of frameworks. They are categorized on the basis of the automation component they leverage. These are:

1. Data-driven testing
2. Modularity-driven testing
3. Keyword-driven testing
4. Hybrid testing
5. Model-based testing

Popular Test Automation Tools

Tool Name	Company Name	Latest Version
HP QuickTest Professional	HP	11.0
IBM Rational Functional Tester	IBM Rational	8.1.0.3
Parasoft SOAtest	Parasoft	9.0

Rational robot	IBM Rational	2003
Selenium	OpenSource Tool	1.0.6
SilkTest	Micro Focus	2010
TestComplete	SmartBear Software	8.0
TestPartner	Micro Focus	6.3
WATIR	OpenSource Tool	1.6.5

Chapter 9

Testing Tools & Features

Instruction set simulator

An **instruction set simulator** (ISS) is a simulation model, usually coded in a high-level programming language, which mimics the behavior of a mainframe or microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers.

Instruction simulation is a methodology employed for one of several possible reasons:

- To simulate the machine code of another hardware device or entire computer for upward compatibility -- a full system simulator typically includes an instruction set simulator.

  For example, the IBM 1401 was simulated on the later IBM/360 through use of microcode emulation.

- To monitor and execute the machine code instructions (but treated as an input stream) on the same hardware for test and debugging purposes, e.g. with memory protection (which protects against accidental or deliberate Buffer overflow).
- To improve the speed performance of simulations involving a processor core where the processor itself is not one of the elements being verified; verilog simulation with ISS by means of "PLI" (not to be confused with PL/1, which is a programming language) speeds considerably.

Quotations

"In the authors opinion, entirely too much programmers' time has been spent in writing such [machine language] simulators and entirely too much computer time has been wasted in using them"

- Donald Knuth, renowned computer scientist and Professor Emeritus , at Stanford University.

In the following section however, the author then proceeds to give examples of how such simulators are useful as trace or monitor routines for debugging purposes.

Implementation

An ISS is often provided with (or is itself) a debugger in order for a Software engineer/Programmer to debug the program prior to obtaining target hardware. GDB is one debugger which have compiled-in ISS. It is sometimes integrated with simulated peripheral circuits such as timers, interrupts, serial port, general I/O port, etc to mimic the behavior of microcontroller.

The basic instruction simulation technique is the same regardless of purpose - first execute the monitoring program passing the name of the target program as an additional input parameter.

The target program is then loaded into memory, but control is never passed to the code. Instead, the entry point within the loaded program is calculated and a pseudo program status word (PSW) is set to this location. A set of pseudo registers are set to what they would have contained if the program had been given control directly.

It may be necessary to amend some of these to point to other pseudo "control blocks" depending on the hardware and operating system. It may also be necessary to reset the original parameter list to 'strip out' the previously added program name parameter.

Thereafter, execution proceeds as follows:

1. Determine length of instruction at pseudo PSW location (initially the first instruction in the target program). If this instruction offset within program matches a set of previously given "pause" points, set "Pause" reason, go to 7.
2. "Fetch" the instruction from its original location (if necessary) into the monitor's memory. If "trace" is available and "on", store program name, instruction offset and any other values.
3. Depending upon instruction type, perform pre-execution checks and execute. If the instruction cannot proceed for any reason (invalid instruction, incorrect mode etc) go to 7. If the instruction is about to alter memory, check memory destination exists (for this thread) and is sufficiently large. If OK, load appropriate pseudo registers into temporary real registers, perform equivalent move with the real registers, save address and length of altered storage if trace is "on" and go to 4. If the instruction is a "register-to-register" operation, load pseudo registers into monitors real registers, perform operation, store back to respective pseudo registers, go to 4. If the instruction is a conditional branch, determine if the condition is satisfied: if not go to 4, if condition IS satisfied, calculate branch to address, determine if valid (if not, set error = "Wild branch") and go to 7. If OK, go to 5. If instruction is an operating system call, do real call from monitorng program by "faking" addresses to return control to monitor program and then reset pseudo registers to reflect call; go to 4.

4. Add instruction length to current Pseudo PSW value.
5. Store next address in Pseudo PSW.
6. Go to 1.
7. Halt execution.

For test and debugging purposes, the monitoring program can provide facilities to view and alter registers, memory, and re-start location or obtain a mini core dump or print symbolic program names with current data values. It could permit new conditional "pause" locations, remove unwanted pauses and suchlike.

Instruction simulation provides the opportunity to detect errors BEFORE execution which means that the conditions are still exactly as they were and not destroyed by the error. A very good example from the IBM S/360 world is the following instruction sequence that can cause difficulties debugging without an instruction simulation monitor.

```
     LM    R14,R12,12(R13)   where r13 incorrectly points to string of
X"00"s
     BR    R14               causes PSW to contain X"0000002" with
program check "Operation Exception"
*                            all registers on error contain nulls.
```

Consequences

Overhead

The number of instructions to perform the above basic "loop" (Fetch/Execute/calculate new address) depends on hardware but it could be accomplished on IBM S/360/370/390/ES9000 range of machines in around 12 or 13 instructions for many instruction types. Checking for valid memory locations or for conditional "pause"s add considerably to the overhead but optimization techniques can reduce this to acceptable levels. For testing purposes this is normally quite acceptable as powerful debugging capabilities are provided including instruction step, trace and deliberate jump to test error routine (when no actual error). In addition, a full instruction trace can be used to test actual (executed) code coverage.

Added benefits

Occasionally, monitoring the execution of a target program can help to highlight random errors that appear (or sometimes disappear) while monitoring but not in real execution. This can happen when the target program is loaded at a different location than normal because of the physical presence of the monitoring program in the same address space.

If the target program picks up the value from a "random" location in memory (one it doesn't 'own' usually), it may for example be nulls (X"00") in almost every normal situation and the program works OK. If the monitoring program shifts the load point, it may pick up say X"FF" and the logic would cause different results during a comparison

operation. Alternatively, if the monitoring program is now occupying the space where the value is being "picked up" from, similar results might occur.

Re-entrancy bugs: accidental use of static variables instead of "dynamic" thread memory can cause re-entrancy problems in many situations. Use of a monitoring program can detect these even without a storage protect key.

Illegal operations: some operating systems (or hardware) require the application program to be in the correct "mode" for certain calls to the Operating system. Instruction simulation can detect these conditions before execution.

Hot spot analysis & instruction usage by counting the instructions executed during simulation (which will match the number executed on the actual processor or unmonitored execution), the simulator can provide both a measure of relative performance between different versions of algorithm and also be used to detect "hot spots" where optimization can then be targeted by the programmer. In this role it can be considered a form of Performance analysis as it is not easy to obtain these statistics under normal execution and this is especially true for high level language programs which effectively 'disguise' the extent of machine code instructions by their nature.

Example

Typical trace output from simulation by monitoring program used for test & debugging:

```
Program        offset         instruction            Dis-assembled
register/ storage (after execution)
TEST001        000000         X'05C0'                BALR   R12,0
R12=002CE00A
               000002         X'47F0C00E'            BC
15,X'00C'(R12)
               00000E         X'98ECD00C'            STM
R14,R12,X'00C'(R13)   X'002E0008' ==> X'00004CE,002CE008,..etc....'
               000012         X'45E0C122'            BAL
R14,X'122'(R12)     R14=002C0016
SUB1           000124         X'50E0C28A'            ST
R14,X'28A'(R12)       X'002CE294' ==> X'002C0016'
etc...
```

Program animation

Program animation or **Stepping** refers to the very common debugging method of executing code one "line" at a time. The programmer may examine the state of the program, machine, and related data *before and after* execution of a particular line of code. This allows evaluatation the effects of that statement or instruction in isolation and thereby gain insight into the behavior (or misbehavior) of the executing program. Nearly all modern IDEs and debuggers support this mode of execution. Some Testing tools allow programs to be executed step-by-step optionally at either source code level or machine code level depending upon the availability of data collected at compile time.

History

System/360 (Model 65) operator's console, with register value lamps and toggle switches and buttons (middle of picture).

Instruction **stepping** or **single cycle** also referred to the related, more microscopic, but now obsolete method of debugging code by stopping the processor clock and manually advancing it one cycle at a time. For this to be possible, three things are required:

- A control that allows the clock to be stopped (e.g. a "Stop" button).
- A second control that allows the stopped clock to be manually advanced by one cycle (e.g. An "instruction step" switch and a "Start" button).
- Some means of recording the state of the processor after each cycle (e.g. register and memory displays).

On the IBM System 360 processor range, these facilities were provided by front panel switches, buttons and banks of neon lights.

Other systems such as the PDP-11 provided similar facilities, again on some models. The precise configuration was also model-dependent. It would not be easy to provide such facilities on LSI processors such as the Intel x86 and Pentium lines, owing to cooling considerations.

As multiprocessing became more commonplace, such techniques would have limited practicality, since many independent processes would be stopped simultaneously. This led to the development of proprietory software from several independent vendors that provided similar features but deliberately restricted breakpoints and instruction stepping to particular application programs in particular address spaces and threads. The program state (as applicable to the chosen application/thread) was saved for examination at each step and restored before resumption, giving the impression of a single user environment. This is normally sufficient for dignosing problems at the application layer.

Instead of using a physical stop button to suspend execution - to then begin stepping through the application program, a breakpoint or "Pause" request must usually be set beforehand, usually at a particular statement/instruction in the program (chosen beforehand or alternatively, by default, at the first instruction).

Techniques for program animation

There are at least three distinct software techniques for creating 'animation' during programs execution.

- **instrumentation** involves adding additional source code to the program at compile time to call the animator before or after each statement to halt normal execution.
- **Induced interrupt** This technique involves forcing a breakpoint at certain points in a program at execution time, usually by altering the machine code instruction at that point (this might be an inserted system call or deliberate invalid operation) and waiting for an interrupt. When the interrupt occurs, it is handled by the testing tool to report the status back to the programmer. This method allows program execution at full speed (until the interrupt occurs) but suffers from the

disadvantage that most of the instructions leading up to the interrupt are not monitored by the tool.
- **Instruction Set Simulator** This technique treats the compiled programs machine code as its input 'data' and fully simulates the host machine instructions, monitors the code for conditional or unconditional breakpoints or programmer requested "single cycle" animation requests between every step.

Comparison of methods

The advantage of the last method is that no changes are made to the compiled program to provide the diagnostic and there is almost unlimited scope for extensive diagnostics since the tool can augment the host system diagnostics with additional software tracing features. It is also possible to diagnose (and prevent) many program errors automatically using this technique, including storage violations and buffer overflows. Loop detection is also possible using automatic instruction trace together with instruction count thresholds (e.g. pause after 10,000 instructions; display last n instructions) The second method only alters the instruction that will halt before it is executed and may also then restore it before optional resumption by the programmer. Some animators optionally allow the use of more than one method depending on requirements. For example, using method 2 to execute to a particular point at full speed and then using instruction set simulation thereafer.

Additional features

The animator may, or may not, combine other test/debugging features within it such as program trace, dump, conditional breakpoint and memory alteration, program flow alteration, code coverage analysis, "hot spot" detection or similar.

Benchmark (computing)

In computing, a **benchmark** is the act of running a computer program, a set of programs, or other operations, in order to assess the relative **performance** of an object, normally by running a number of standard tests and trials against it. The term 'benchmark' is also mostly utilized for the purposes of elaborately-designed benchmarking programs themselves.

Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems.

Another type of test program, namely test suites or validation suites, are intended to assess the **correctness** of software.

Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures.

Purpose

As computer architecture advanced, it became more difficult to compare the performance of various computer systems simply by looking at their specifications. Therefore, tests were developed that allowed comparison of different architectures. For example, Pentium 4 processors generally operate at a higher clock frequency than Athlon XP processors, which does not necessarily translate to more computational power. A slower processor, with regard to clock frequency, can perform as well as a processor operating at a higher frequency.

Benchmarks are designed to mimic a particular type of workload on a component or system. Synthetic benchmarks do this by specially created programs that impose the workload on the component. Application benchmarks run real-world programs on the system. Whilst application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks are useful for testing individual components, like a hard disk or networking device.

Benchmarks are particularly important in CPU design, giving processor architects the ability to measure and make tradeoffs in microarchitectural decisions. For example, if a benchmark extracts the key algorithms of an application, it will contain the performance-sensitive aspects of that application. Running this much smaller snippet on a cycle-accurate simulator can give clues on how to improve performance.

Prior to 2000, computer and microprocessor architects used SPEC to do this, although SPEC's Unix-based benchmarks were quite lengthy and thus unwieldy to use intact.

Computer manufacturers are known to configure their systems to give unrealistically high performance on benchmark tests that are not replicated in real usage. For instance, during the 1980s some compilers could detect a specific mathematical operation used in a well-known floating-point benchmark and replace the operation with a faster mathematically-equivalent operation. However, such a transformation was rarely useful outside the benchmark until the mid-1990s, when RISC and VLIW architectures emphasized the importance of compiler technology as it related to performance. Benchmarks are now regularly used by compiler companies to improve not only their own benchmark scores, but real application performance.

CPUs that have many execution units — such as a superscalar CPU, a VLIW CPU, or a reconfigurable computing CPU — typically have slower clock rates than a sequential CPU with one or two execution units when built from transistors that are just as fast. Nevertheless, CPUs with many execution units often complete real-world and benchmark tasks in less time than the supposedly faster high-clock-rate CPU.

Given the large number of benchmarks available, a manufacturer can usually find at least one benchmark that shows its system will outperform another system; the other systems can be shown to excel with a different benchmark.

Manufacturers commonly report only those benchmarks (or aspects of benchmarks) that show their products in the best light. They also have been known to mis-represent the significance of benchmarks, again to show their products in the best possible light. Taken together, these practices are called *bench-marketing*.

Ideally benchmarks should only substitute for real applications if the application is unavailable, or too difficult or costly to port to a specific processor or computer system. If performance is critical, the only benchmark that matters is the target environment's application suite.

Challenges

Benchmarking is not easy and often involves several iterative rounds in order to arrive at predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult. Here is a partial list of common challenges:

- Vendors tend to tune their products specifically for industry-standard benchmarks. Norton SysInfo (SI) is particularly easy to tune for, since it mainly biased toward the speed of multiple operations. Use extreme caution in interpreting such results.
- Some vendors have been accused of "cheating" at benchmarks—doing things that give much higher benchmark numbers, but make things worse on the actual likely workload.
- Many benchmarks focus entirely on the speed of computational performance, neglecting other important features of a computer system, such as:
    - Qualities of service, aside from raw performance. Examples of unmeasured qualities of service include security, availability, reliability, execution integrity, serviceability, scalability (especially the ability to quickly and nondisruptively add or reallocate capacity), etc. There are often real trade-offs between and among these qualities of service, and all are important in business computing. Transaction Processing Performance Council Benchmark specifications partially address these concerns by specifying ACID property tests, database scalability rules, and service level requirements.
    - In general, benchmarks do not measure Total cost of ownership. Transaction Processing Performance Council Benchmark specifications partially address this concern by specifying that a price/performance metric must be reported in addition to a raw performance metric, using a simplified TCO formula. However, the costs are necessarily only partial, and vendors have been known to price specifically (and only) for the benchmark, designing a highly specific "benchmark special" configuration with an artificially low price. Even a tiny deviation from the benchmark package results in a much higher price in real world experience.

    - Facilities burden (space, power, and cooling). When more power is used, a portable system will have a shorter battery life and require recharging more often. A server that consumes more power and/or space may not be able to fit within existing data center resource constraints, including cooling limitations. There are real trade-offs as most semiconductors require more power to switch faster..
    - In some embedded systems, where memory is a significant cost, better code density can significantly reduce costs.

- Vendor benchmarks tend to ignore requirements for development, test, and disaster recovery computing capacity. Vendors only like to report what might be narrowly required for production capacity in order to make their initial acquisition price seem as low as possible.
- Benchmarks are having trouble adapting to widely distributed servers, particularly those with extra sensitivity to network topologies. The emergence of grid computing, in particular, complicates benchmarking since some workloads are "grid friendly", while others are not.
- Users can have very different perceptions of performance than benchmarks may suggest. In particular, users appreciate predictability — servers that always meet or exceed service level agreements. Benchmarks tend to emphasize mean scores (IT perspective) rather than low standard deviations (user perspective).
- Many server architectures degrade dramatically at high (near 100%) levels of usage — "fall off a cliff" — and benchmarks should (but often do not) take that factor into account. Vendors, in particular, tend to publish server benchmarks at continuous at about 80% usage — an unrealistic situation — and do not document what happens to the overall system when demand spikes beyond that level.
- Many benchmarks focus on one application, or even one application tier, to the exclusion of other applications. Most data centers are now implementing virtualization extensively for a variety of reasons, and benchmarking is still catching up to that reality where multiple applications and application tiers are concurrently running on consolidated servers.
- There are few (if any) high quality benchmarks that help measure the performance of batch computing, especially high volume concurrent batch and online computing. Batch computing tends to be much more focused on the predictability of completing long-running tasks correctly before deadlines, such as end of month or end of fiscal year. Many important core business processes are batch-oriented and probably always will be, such as billing.
- Benchmarking institutions often disregard or do not follow basic scientific method. This includes, but is not limited to: small sample size, lack of variable control, and the limited repeatability of results.

Types of benchmarks

1. Real program
    - word processing software
    - tool software of CDA

    - user's application software (MIS)
2. Microbenchmark
    - Designed to measure the performance of a very small and specific piece of code.
3. Kernel
    - contains key codes
    - normally abstracted from actual program
    - popular kernel: Livermore loop
    - linpack benchmark (contains basic linear algebra subroutine written in FORTRAN language)
    - results are represented in MFLOPS
4. Component Benchmark/ micro-benchmark
    - programs designed to measure performance of a computer's basic components
    - automatic detection of computer's hardware parameters like number of registers, cache size, memory latency
5. Synthetic Benchmark
    - Procedure for programming synthetic benchmark:
        - take statistics of all types of operations from many application programs
        - get proportion of each operation
        - write program based on the proportion above
    - Types of Synthetic Benchmark are:
        - Whetstone
        - Dhrystone
    - These were the first general purpose industry standard computer benchmarks. They do not necessarily obtain high scores on modern pipelined computers.
6. I/O benchmarks
7. Database benchmarks: to measure the throughput and response times of database management systems (DBMS')
8. Parallel benchmarks: used on machines with multiple processors or systems consisting of multiple machines

Profiling (computer programming)

In software engineering, **program profiling**, **software profiling** or simply **profiling**, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

- A **(code) profiler** is a **performance analysis** tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.
- An instruction set simulator which is also — by necessity — a profiler, can measure the totality of a program's behaviour from invocation to termination.

Gathering program events

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. The usage of profilers is 'called out' in the performance engineering process.

Use of profilers

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing... (ATOM, PLDI, '94)

The output of a profiler may be:-

- A statistical *summary* of the events observed (a **profile**)

  Summary profile information is often shown annotated against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program.

```
/* ------------ source------------------------- count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

- A stream of recorded events (a **trace**)

  For sequential programs, a summary profile is usually sufficient, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening.
  The size of a (full) trace is linear to the program's instruction path length, making it somewhat impractical. A trace may therefore be initiated at one point in a program and terminated at another point to limit the output.

- An ongoing interaction with the hypervisor (continuous or periodic monitoring via on-screen display for instance)

  This provides the opportunity to switch a trace on or off at any desired point during execution in addition to viewing on-going metrics about the (still executing) program. It also provides the opportunity to suspend asynchronous processes at critical points to examine interactions with other parallel processes in more detail.

History

Performance analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the Program status word (PSW) at set timer intervals to detect "hot spots" in executing code. This was an early example of sampling (see below). In early 1974, Instruction Set Simulators permitted full trace and other performance monitoring features.

Profiler-driven program analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "prof" that listed each function and how much of program execution time it used. In 1982, gprof extended the concept to a complete call graph analysis

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM . ATOM is a platform for converting a program into its own profiler. That is, at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "instrumentation".

In 2004, both the Gprof and ATOM papers appeared on the list of the 50 most influential PLDI papers of all time.

Profiler types based on output

Flat profiler

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph profiler

Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Methods of data gathering

Event based profilers

The programming languages listed here have event-based profilers:

1. .NET: Can attach a profiling agent as a COM server to the CLR. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can rewrite the target application's bytecode in arbitrary ways.
2. Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
3. Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile()' module to trap events like c_{call,return,exception}, python_{call,return,exception}.
4. Ruby: Ruby also uses a similar interface like Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical profilers

Some profilers operate by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.

The resulting data are not exact, but a statistical approximation. *The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.*

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of cpu cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

Dedicated hardware can go beyond this: some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the program counter in a truly undetectable manner.

Some of the most commonly used statistical profilers are AMD CodeAnalyst, Apple Inc. Shark, gprof, Intel VTune and Parallel Amplifier (part of Intel Parallel Studio).

Instrumenting profilers

Some profilers **instrument** the target program with additional instructions to collect the required information.

Instrumenting the program can cause changes in the performance of the program, potentially causing inaccurate results and heisenbugs. Instrumenting will always have some impact on the program execution, typically always slowing it. However, instrumentation can be very specific and be carefully controlled to have a minimal impact. The impact on a particular program depends on the placement of instrumentation points and the mechanism used to capture the trace. Hardware support for trace capture means that on some targets, instrumentation can be on just one machine instruction. The impact of instrumentation can often be deducted (i.e. eliminated by subtraction) from the results.

gprof is an example of a profiler that uses both instrumentation and sampling. Instrumentation is used to gather caller information and the actual timing values are obtained by statistical sampling.

Instrumentation

- **Manual**: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- **Automatic source level**: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- **Compiler assisted**: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
- **Binary translation**: The tool adds instrumentation to a compiled binary. Example: ATOM
- **Runtime instrumentation**: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: Pin, Valgrind
- **Runtime injection**: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

Interpreter instrumentation

- **Interpreter debug** options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator

- **Hypervisor**: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
- **Simulator** and **Hypervisor**: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator. Examples: SIMON and OLIVER.