

# Software Quality & Quality Assurance Engineering



Angelique Mitchel

Lane Galvin

First Edition, 2012

ISBN 978-81-323-0988-8

© All rights reserved.

*Published by:*  
**Academic Studio**  
4735/22 Prakashdeep Bldg,  
Ansari Road, Darya Ganj,  
Delhi - 110002  
Email: [info@wtbooks.com](mailto:info@wtbooks.com)

# Table of Contents

Chapter 1 - Software Quality

Chapter 2 - Anti–Pattern

Chapter 3 - Software Bug

Chapter 4 - Fault–Tolerant System and Feature Interaction Problem

Chapter 5 - Independent Software Verification and Validation and Software Assurance

Chapter 6 - Software Rot

Chapter 7 - Reverse Semantic Traceability

Chapter 8 - Sneak Circuit Analysis and Verification and Validation (Software)

Chapter 9 - Software Security Assurance

Chapter 10 - Algorithmic Efficiency

Chapter 11 - Quality Assurance

Chapter 12 - Load Testing

Chapter 13 - Trent Accreditation Scheme

Chapter 14 - QCReporting and Software Assurance

Chapter 15 - Software Testing

Chapter 16 - Software Performance Testing

Chapter 17 - Graphical user Interface Testing

Chapter 18 - Evidence–based Medicine

## Chapter-1

# Software Quality

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions, including conformance to customer expectations. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation, *quality of design* measures how valid the design and requirements are in creating a worthwhile product.

### **Definition**

One of the challenges of software quality is that "everyone feels they understand it".

In addition to more software specific definitions given below, there are several applicable definitions of quality which are used in business. [Quality\\_\(business\)#Definitions](#)

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the management in software engineering.

If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better." This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

Another definition, coined by Gerald Weinberg in *Quality Software Management: Systems Thinking*, is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to *them*?"

## ***History***

### **Software product quality**

- Product quality
  - conformance to requirements or program specification; related to Reliability
- Scalability
- Correctness
- Completeness
- Absence of bugs
- Fault-tolerance
  - Extensibility
  - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

It is essential to supplement traditional testing – functional, non-functional, and run-time – with measures of application structural quality. Structural quality is the quality of the

application's architecture and the degree to which its implementation accords with software engineering best practices. Industry data demonstrate that poor application structural quality results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems. As in any other field of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs.

## **Source code quality**

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Methods to improve the quality:

- Refactoring
- Code Inspection or software review
- Documenting the code

## **Software reliability**

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

## **History**

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have even caused human fatalities. The causes have

ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that led to multiple deaths is discussed in Dr. Leveson's paper (PDF). This has resulted in requirements for development of some types of software. In the United States, both the Food and Drug Administration (FDA) and Federal Aviation Administration (FAA) have requirements for software development.

## **Goal of reliability**

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both laypersons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

## **Challenge of reliability**

The circular logic of the preceding sentence is not accidental—it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual

outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

## **Reliability in program development**

### **Requirements**

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

### **Design**

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby—it is hoped—removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

## **Programming**

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed.

## **Software Build and Deployment**

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

## Testing

**Software testing**, when done correctly, can increase overall software *quality of conformance* by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

## Runtime

runtime reliability determinations are similar to tests, but go beyond simple confirmation of behaviour to the evaluation of qualities such as performance and interoperability with other code or particular hardware configurations.

## Software quality factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are

binary; that is, they are not “either you have it or you don’t” traits. Rather, they are characteristics that one seeks to maximize in one’s software to optimize its quality. So rather than asking whether a software product “has” factor  $x$ , ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

#### Understandability

Clarity of purpose. This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

#### Completeness

Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

#### Conciseness

Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

#### Portability

Ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware—such as running on a PC as well as a smartphone—and between different operating systems—such as running on both Mac OS X and GNU/Linux.

#### Consistency

Uniformity in notation, symbology, appearance, and terminology within itself.

#### Maintainability

Propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

#### Testability

Disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

#### Usability

Convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

#### Reliability

Ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself (sometimes termed robustness).

#### Efficiency

Fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

#### Security

Ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

### **Measurement of software quality factors**

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals—or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well.

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

1. What constitutes “many faults?” Does this differ depending upon the purpose of the software (e.g., blogging software vs. navigational software)? Does this take into account the size and complexity of the software?
2. Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this metric by the severity of the fault, or the incidence of users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000?
3. If the count of faults being discovered is shrinking, how do I know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that fewer tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that fewer faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software. If a team discovers that they will benefit from a drop in the number of reported bugs, there

is a strong tendency for the team to start reporting fewer defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously “game” the measurements.

Software quality factors cannot be measured because of their vague definitions. It is necessary to find measurements, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However, there are related attributes to reliability, which can indeed be measured. Some such attributes are mean time to failure, rate of failure occurrence, and availability of the system. Similarly, an attribute of portability is the number of target-dependent statements in a program.

A scheme that could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measurement of the characteristic can be obtained.

### **Understandability**

Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?...

### **Completeness**

Are all necessary components available? Does any process fail for lack of resources or programming? Are all potential pathways through the code accounted for, including proper error handling?

### **Conciseness**

Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

### **Portability**

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? How much effort would be required to transfer the program from one hardware/software system or environment to another?

## **Consistency**

Is one variable name used to represent different logical or physical entities in the program? Does the program contain only one representation for any given physical or mathematical constant? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements?

## **Maintainability**

Has some memory capacity been reserved for future expansion? Is the design cohesive—i.e., does each module have distinct, recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If the code is procedure-based (rather than object-oriented), is a change likely to require restructuring the main program, or just a module?

## **Testability**

Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

## **Usability**

Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

## **Reliability**

Are loop indexes range-tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? It is the probability that the software performs its intended functions correctly in a specified period of time under stated operation conditions, but there could also be a problem with the requirement document...

## **Efficiency**

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into subroutines? Has the program been checked for memory leaks or overflow errors?

## **Security**

Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are security mechanisms appropriate,

adequate and correctly implemented? Can the software withstand attacks that can be anticipated in its intended environment?

### ***User's perspective***

In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive (self-explanatory/self-documenting)?
- Is it easy to perform simple operations?
- Is it feasible to perform complex operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface responsive or too slow?

Also, the availability of (free or paid) support may factor into the usability of the software.

## Chapter-2

# Anti-Pattern

In software engineering, an **anti-pattern** (or **antipattern**) is a pattern that may be commonly used but is ineffective and/or counterproductive in practice.

The term was coined in 1995 by Andrew Koenig, inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns*, which extended the use of the term beyond the field of software design and into general social interaction. According to the authors of the latter, there must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea:

- Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and
- A refactored solution exists that is clearly documented, proven in actual practice and repeatable.

Often pejoratively named with clever oxymoronic neologisms, many anti-pattern ideas amount to little more than mistakes, rants, unsolvable problems, or bad practices to be avoided if possible. Sometimes called *pitfalls* or *dark patterns*, this informal use of the term has come to refer to classes of commonly reinvented bad solutions to problems. Thus, many candidate anti-patterns under debate would not be formally considered anti-patterns.

By formally describing repeated mistakes, one can recognize the forces that lead to their repetition and learn how others have refactored themselves out of these broken patterns.

### ***Known anti-patterns***

#### **Organizational anti-patterns**

- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project

- Cash cow: A profitable legacy product that often leads to complacency about new products
- Design by committee: The result of having many contributors to a design, but no unifying vision
- Escalation of commitment: Failing to revoke a decision when it proves wrong
- Management by perkele: Authoritarian style of management with no tolerance of dissent
- Matrix Management: Unfocused organizational structure that results in divided loyalties and lack of direction
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision
- Mushroom management: Keeping employees uninformed and misinformed (kept in the dark and fed manure), let to stew, and finally canned.
- Stovepipe or Silos: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component

### **Project management anti-patterns**

- Death march: Everyone knows that the project is going to be a disaster – except the CEO -- so the truth is hidden to prevent immediate cancellation of the project - (although the CEO often knows and does it anyway to maximise profit). However, the truth remains hidden and the project is artificially kept alive until the Day Zero finally comes ("Big Bang"). Alternative definition: Employees are pressured to work late nights and weekends on a project with an unreasonable deadline.
- Groupthink: During groupthink, members of the group avoid promoting viewpoints outside the comfort zone of consensus thinking
- Smoke and mirrors: Demonstrating how unimplemented functions will appear
- Software bloat: Allowing successive versions of a system to demand ever more resources
- Waterfall model: An older method of software development that inadequately deals with unanticipated change

### **Analysis anti-patterns**

- Bystander apathy: When a requirement or design decision is wrong, but the people who notice this do nothing because it affects a larger number of people

### **Software design anti-patterns**

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- Ambiguous viewpoint: Presenting a model (usually Object-oriented analysis and design (OOAD)) without specifying its viewpoint

- Big ball of mud: A system with no recognizable structure
- Database-as-IPC: Using a database as the message queue for routine interprocess communication where a much more lightweight mechanism would be suitable
- Gold plating: Continuing to work on a task or project well past the point at which extra effort is adding value
- Inner-platform effect: A system so customizable as to become a poor replica of the software development platform
- Input kludge: Failing to specify and implement the handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: Coding implementation logic directly within interface code, without using abstraction
- Race hazard: Failing to see the consequence of different orders of events
- Stovepipe system: A barely maintainable assemblage of ill-related components

## Object-oriented design anti-patterns

- Anemic Domain Model: The use of domain model without any business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation and mutation logic is placed somewhere outside (most likely in multiple places).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

## Programming anti-patterns

- Accidental complexity: Introducing unnecessary complexity into a solution
- Action at a distance: Unexpected interaction between widely separated parts of a system

- Blind faith: Lack of checking of (a) the correctness of a bug fix or (b) the result of a subroutine
- Boat anchor: Retaining a part of a system that no longer has any use
- Busy spin: Consuming CPU while waiting for something to happen, usually by repeated checking instead of messaging
- Caching failure: Forgetting to reset an error flag when an error has been corrected
- Cargo cult programming: Using patterns and methods without understanding why
- Coding by exception: Adding new code to handle each special case as it is recognized
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- Hard code: Embedding assumptions about the environment of a system in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- Loop-switch sequence: Encoding a set of sequential steps using a switch within a loop statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Including literal strings in code, for comparisons, as event types etc.
- Soft code: Storing business logic in configuration files rather than source code
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

## Methodological anti-patterns

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- Golden hammer: Assuming that a favorite solution is universally applicable (See: Silver Bullet)
- Improbability factor: Assuming that it is improbable that a known error will occur
- Not Invented Here (NIH) syndrome: The tendency towards *reinventing the wheel* (Failing to adopt an existing, adequate solution)
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation (or "programming by accident"): Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the wheel: Failing to adopt an existing, adequate solution
- Reinventing the square wheel: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester Driven Development: Software projects in which new requirements are specified in bug reports

## **Configuration management anti-patterns**

- Dependency hell: Problems with versions of required products
- DLL hell: Inadequate management of dynamic-link libraries (DLLs), specifically on Microsoft Windows
- Extension conflict: Problems with different extensions to pre-Mac OS X versions of the Mac OS attempting to patch the same parts of the operating system
- JAR hell: Overutilization of the multiple JAR files, usually causing versioning and location problems because of misunderstanding of the Java class loading model

## Chapter-3

# Software Bug

A **software bug** is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be *buggy*. Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, change requests, and so forth.

### **Effects**

Bugs trigger Type I and type II errors that can in turn have a wide variety of ripple effects, with varying levels of inconvenience to the user of the program. Some bugs have only a subtle effect on the program's functionality, and may thus lie undetected for a long time. More serious bugs may cause the program to crash or freeze leading to a denial of service. Others qualify as security bugs and might for example enable a malicious user to bypass access controls in order to obtain unauthorized privileges.

The results of bugs may be extremely serious. Bugs in the code controlling the Therac-25 radiation therapy machine were directly responsible for some patient deaths in the 1980s. In 1996, the European Space Agency's US\$1 billion prototype Ariane 5 rocket was destroyed less than a minute after launch, due to a bug in the on-board guidance computer program. In June 1994, a Royal Air Force Chinook crashed into the Mull of Kintyre, killing 29. This was initially dismissed as pilot error, but an investigation by *Computer Weekly* uncovered sufficient evidence to convince a House of Lords inquiry that it may have been caused by a software bug in the aircraft's engine control computer.

In 2002, a study commissioned by the US Department of Commerce' National Institute of Standards and Technology concluded that *software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated \$59 billion annually, or about 0.6 percent of the gross domestic product.*

## ***Etymology***

The concept that software might contain errors dates back to 1843 in Ada Byron's notes on the analytical engine in which she speaks of the difficulty of preparing program 'cards' for Charles Babbage's Analytical engine:

“ ...an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders. ”

Use of the term "bug" to describe inexplicable defects has been a part of engineering jargon for many decades and predates computers and computer software; it may have originally been used in hardware engineering to describe mechanical malfunctions. For instance, Thomas Edison wrote the following words in a letter to an associate in 1878:

“ It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and [*it is*] then that 'Bugs' — as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached. ”

Problems with radar electronics during World War II were referred to as *bugs* (or glitches) and there is additional evidence that the usage dates back much earlier. Baffle Ball, the first mechanical pinball game, was advertised as being "free of bugs" in 1931.

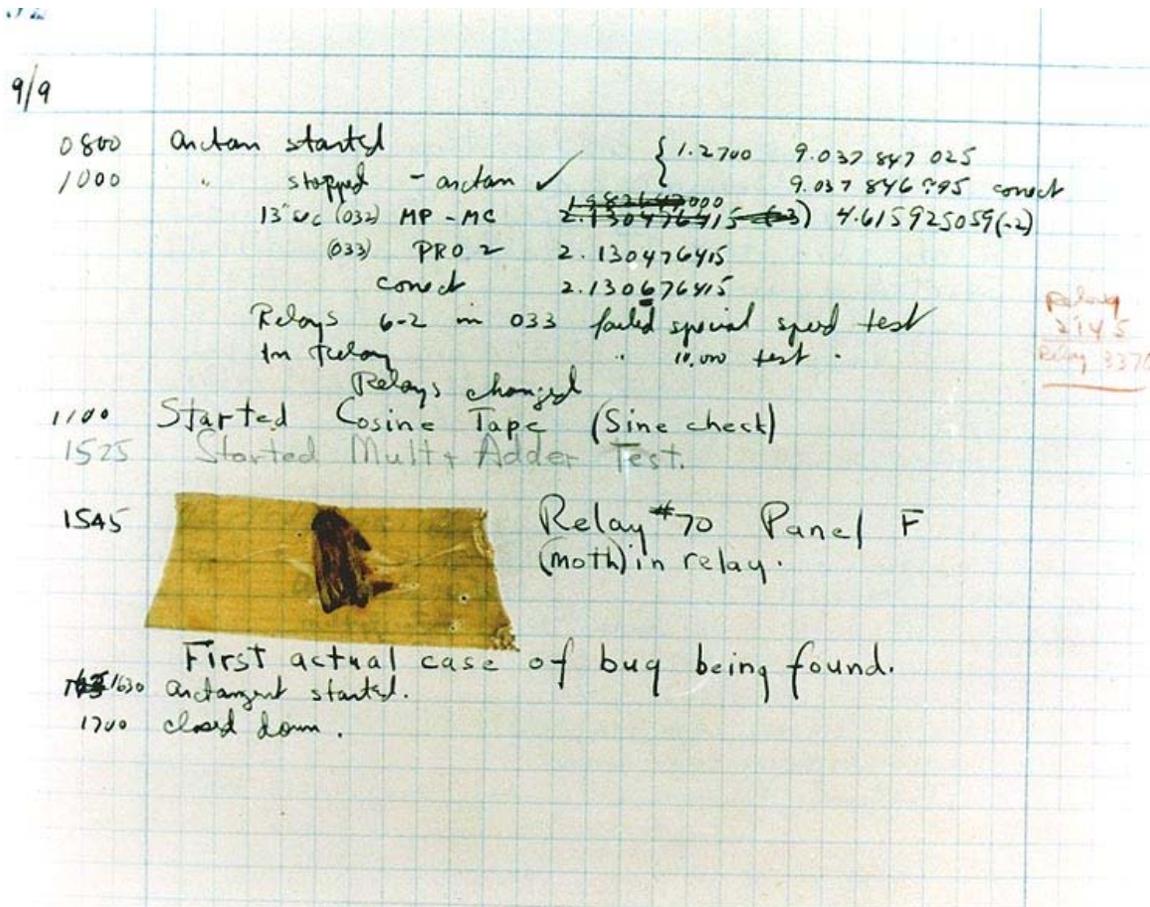


Photo of what is possibly the first real bug found in a computer.

The invention of the term "bug" is often erroneously attributed to Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. A typical version of the story is given by this quote:

“ In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's [*sic*] in a program a *bug*. ”

Hopper was not actually the one who found the insect, as she readily acknowledged. The date in the log book was 9 September 1947, although sometimes erroneously reported as 1945. The operators who did find it, including William "Bill" Burke, later of the Naval Weapons Laboratory, Dahlgren, Virginia, were familiar with the engineering term and, amused, kept the insect with the notation "First actual case of bug being found." Hopper loved to recount the story. This log book is on display in the Smithsonian National Museum of American History, complete with moth attached.

While it is certain that the Harvard Mark II operators did not coin the term "bug", it has been suggested that they did coin the related term, "debug". Even this is unlikely, since the Oxford English Dictionary entry for "debug" contains a use of "debugging" in the context of air-plane engines in 1945.

## **Prevention**

Bugs are a consequence of the nature of human factors in the programming task. They arise from oversights or mutual misunderstandings made by a software team during specification, design, coding, data entry and documentation. For example: In creating a relatively simple program to sort a list of words into alphabetical order, one's design might fail to consider what should happen when a word contains a hyphen. Perhaps, when converting the abstract design into the chosen programming language, one might inadvertently create an off-by-one error and fail to sort the last word in the list. Finally, when typing the resulting program into the computer, one might accidentally type a '<' where a '>' was intended, perhaps resulting in the words being sorted into reverse alphabetical order. More complex bugs can arise from unintended interactions between different parts of a computer program. This frequently occurs because computer programs can be complex—millions of lines long in some cases—often having been programmed by many people over a great length of time, so that programmers are unable to mentally track every possible way in which parts can interact. Another category of bug called a *race condition* comes about either when a process is running in more than one thread or two or more processes run simultaneously, and the exact order of execution of the critical sequences of code have not been properly synchronized.

The software industry has put much effort into finding methods for preventing programmers from inadvertently introducing bugs while writing software. These include:

### Programming style

While typos in the program code are often caught by the compiler, a bug usually appears when the programmer makes a logic error. Various innovations in programming style and defensive programming are designed to make these bugs less likely, or easier to spot. In some programming languages, so-called typos, especially of symbols or logical/mathematical operators, actually represent logic errors, since the mistyped constructs are accepted by the compiler with a meaning other than that which the programmer intended.

### Programming techniques

Bugs often create inconsistencies in the internal data of a running program. Programs can be written to check the consistency of their own internal data while running. If an inconsistency is encountered, the program can immediately halt, so that the bug can be located and fixed. Alternatively, the program can simply inform the user, attempt to correct the inconsistency, and continue running.

### Development methodologies

There are several schemes for managing programmer activity, so that fewer bugs are produced. Many of these fall under the discipline of software engineering (which addresses software design issues as well). For example, formal program

specifications are used to state the exact behavior of programs, so that design bugs can be eliminated. Unfortunately, formal specifications are impractical or impossible for anything but the shortest programs, because of problems of combinatorial explosion and indeterminacy.

#### Programming language support

Programming languages often include features which help programmers prevent bugs, such as static type systems, restricted name spaces and modular programming, among others. For example, when a programmer writes (pseudocode) `LET REAL_VALUE PI = "THREE AND A BIT"`, although this may be syntactically correct, the code fails a type check. Depending on the language and implementation, this may be caught by the compiler or at runtime. In addition, many recently-invented languages have deliberately excluded features which can easily lead to bugs, at the expense of making code slower than it need be: the general principle being that, because of Moore's law, computers get faster and software engineers get slower; it is *almost always* better to write simpler, slower code than "clever", inscrutable code, especially considering that maintenance cost is considerable. For example, the Java programming language does not support pointer arithmetic; implementations of some languages such as Pascal and scripting languages often have runtime bounds checking of arrays, at least in a debugging build.

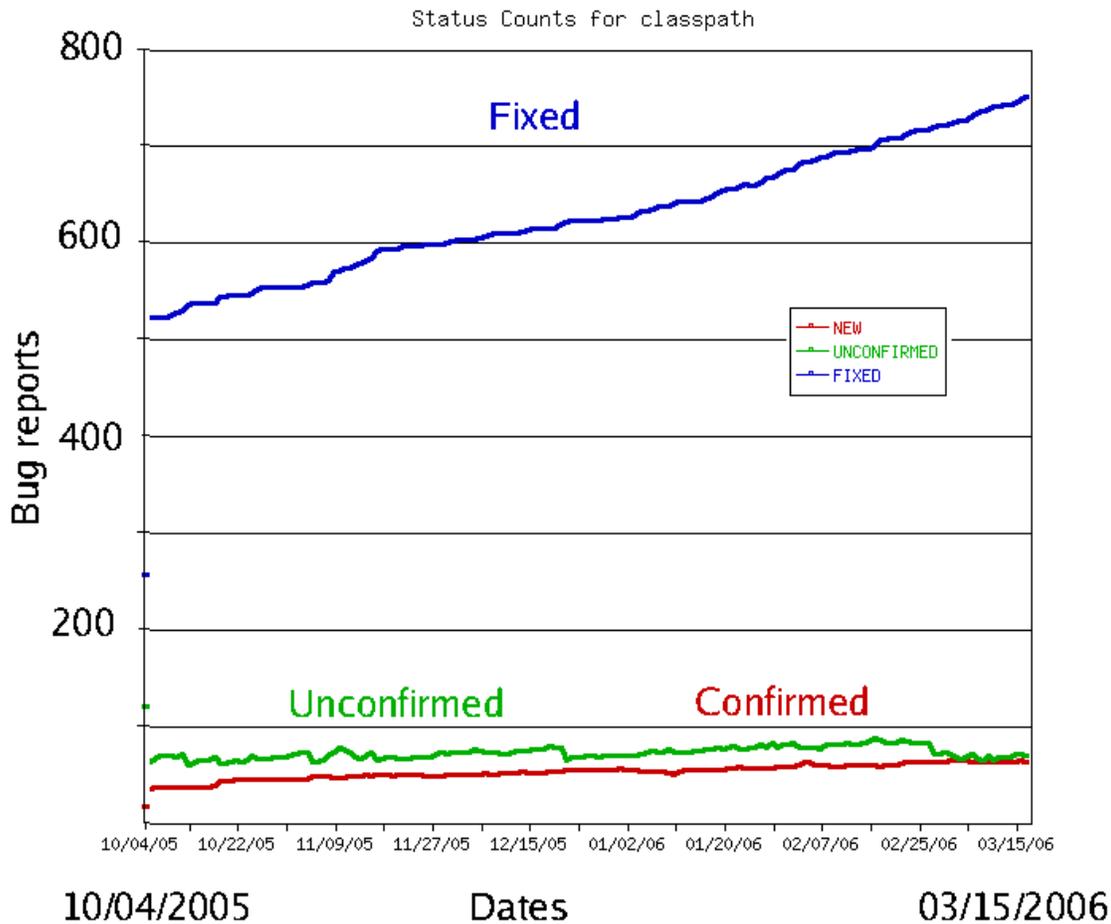
#### Code analysis

Tools for code analysis help developers by inspecting the program text beyond the compiler's capabilities to spot potential problems. Although in general the problem of finding all programming errors given a specification is not solvable, these tools exploit the fact that human programmers tend to make the same kinds of mistakes when writing software.

#### Instrumentation

Tools to monitor the performance of the software as it is running, either specifically to find problems such as bottlenecks or to give assurance as to correct working, may be embedded in the code explicitly (perhaps as simple as a statement saying `PRINT "I AM HERE"`), or provided as tools. It is often a surprise to find where most of the time is taken by a piece of code, and this removal of assumptions might cause the code to be rewritten.

## Debugging



The typical bug history (GNU Classpath project data). A new bug submitted by the user is *unconfirmed*. Once it has been reproduced by a developer, it is a *confirmed* bug. The confirmed bugs are later *fixed*. Bugs belonging to other categories (unreproducible, will not be fixed, etc.) are usually in the minority

Finding and fixing bugs, or "debugging", has always been a major part of computer programming. Maurice Wilkes, an early computing pioneer, described his realization in the late 1940s that much of the rest of his life would be spent finding mistakes in his own programs. As computer programs grow more complex, bugs become more common and difficult to fix. Often programmers spend more time and effort finding and fixing bugs than writing new code. Software testers are professionals whose primary task is to find bugs, or write code to support testing. On some projects, more resources can be spent on testing than in developing the program.

Usually, the most difficult part of debugging is finding the bug in the source code. Once it is found, correcting it is usually relatively easy. Programs known as debuggers exist to help programmers locate bugs by executing code line by line, watching variable values, and other features to observe program behavior. Without a debugger, code can be added

so that messages or values can be written to a console (for example with *printf* in the C programming language) or to a window or log file to trace program execution or show values.

However, even with the aid of a debugger, locating bugs is something of an art. It is not uncommon for a bug in one section of a program to cause failures in a completely different section, thus making it especially difficult to track (for example, an error in a graphics rendering routine causing a file I/O routine to fail), in an apparently unrelated part of the system.

Sometimes, a bug is not an isolated flaw, but represents an error of thinking or planning on the part of the programmer. Such *logic errors* require a section of the program to be overhauled or rewritten. As a part of Code review, stepping through the code modelling the execution process in one's head or on paper can often find these errors without ever needing to reproduce the bug as such, if it can be shown there is some faulty logic in its implementation.

But more typically, the first step in locating a bug is to reproduce it reliably. Once the bug is reproduced, the programmer can use a debugger or some other tool to monitor the execution of the program in the faulty region, and find the point at which the program went astray.

It is not always easy to reproduce bugs. Some are triggered by inputs to the program which may be difficult for the programmer to re-create. One cause of the Therac-25 radiation machine deaths was a bug (specifically, a race condition) that occurred only when the machine operator very rapidly entered a treatment plan; it took days of practice to become able to do this, so the bug did not manifest in testing or when the manufacturer attempted to duplicate it. Other bugs may disappear when the program is run with a debugger; these are heisenbugs (humorously named after the Heisenberg uncertainty principle.)

Debugging is still a tedious task requiring considerable effort. Since the 1990s, particularly following the Ariane 5 Flight 501 disaster, there has been a renewed interest in the development of effective automated aids to debugging. For instance, methods of static code analysis by abstract interpretation have already made significant achievements, while still remaining much of a work in progress.

As with any creative act, sometimes a flash of inspiration will show a solution, but this is rare and, by definition, cannot be relied on.

There are also classes of bugs that have nothing to do with the code itself. If, for example, one relies on faulty documentation or hardware, the code may be written perfectly properly to what the documentation says, but the bug truly lies in the documentation or hardware, not the code. However, it is common to change the code instead of the other parts of the system, as the cost and time to change it is generally less. Embedded systems frequently have workarounds for hardware bugs, since to make a new version of a ROM

is much cheaper than remanufacturing the hardware, especially if they are commodity items.

## ***Bug management***

It is common practice for software to be released with known bugs that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last week, we should have 100 this week"). Most big software projects maintain two lists of "known bugs"— those known to the software team, and those to be told to users. This is not dissimulation, but users are not concerned with the internal workings of the product. The second list informs users about bugs that are not fixed in the current release, or not fixed at all, and a workaround may be offered.

There are various reasons for not fixing bugs:

- The developers often don't have time or it is not economical to fix all non-severe bugs.
- The bug could be fixed in a new version or patch that is not yet released.
- The changes to the code required to fix the bug could be large, expensive, or delay finishing the project.
- Even seemingly simple fixes bring the chance of introducing new unknown bugs into the system. At the end of a test/fix cycle some managers may only allow the most critical bugs to be fixed.
- Users may be relying on the undocumented, buggy behavior, especially if scripts or macros rely on a behavior; it may introduce a breaking change.
- It's "not a bug". A misunderstanding has arisen between expected and provided behavior

Given the above, it is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users. NASA's SATC managed to reduce the number of errors to fewer than 0.1 per 1000 lines of code (SLOC) but this was not felt to be feasible for any real world projects.

The severity of a bug is not the same as its importance for fixing, and the two should be measured and managed separately. On a Microsoft Windows system a blue screen of death is rather severe, but if it only occurs in extreme circumstances, especially if they are well diagnosed and avoidable, it may be less important to fix than an icon not representing its function well, which though purely aesthetic may confuse thousands of users every single day. This balance, of course, depends on many factors; expert users have different expectations from novices, a niche market is different from a general consumer market, and so on. To better achieve this balance, some software developers use a formalized *bug triage* process (borrowing the medical term), in which each new

bug is assigned a priority based on its severity, frequency, risk, and other predetermined factors.

A school of thought popularized by Eric S. Raymond as Linus's Law says that popular open-source software has more chance of having few or no bugs than other software, because "given enough eyeballs, all bugs are shallow". This assertion has been disputed, however: computer security specialist Elias Levy wrote that "it is easy to hide vulnerabilities in complex, little understood and undocumented source code," because, "even if people are reviewing the code, that doesn't mean they're qualified to do so."

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done" and managing purely by bug counts can have unintended consequences. If, for example, developers are rewarded by the number of bugs they fix, they will naturally fix the easiest bugs first—leaving the hardest, and probably most risky or critical, to the last possible moment ("I only have one bug on my list but it says "Make sun rise in West"). If the management ethos is to reward the number of bugs fixed, then some developers may quickly write sloppy code knowing they can fix the bugs later and be rewarded for it, whereas careful, perhaps "slower" developers do not get rewarded for the bugs that were never there.

## ***Security vulnerabilities***

Malicious software may attempt to exploit known vulnerabilities in a system — which may or may not be bugs. Viruses are not bugs in themselves — they are typically programs that are doing precisely what they were designed to do. However, viruses are occasionally referred to as such in the popular press.

## ***Common types of computer bugs***

- Conceptual error (code is syntactically correct, but the programmer or designer intended it to do something else)

### **Arithmetic bugs**

- Division by zero
- Arithmetic overflow or underflow
- Loss of arithmetic precision due to rounding or numerically unstable algorithms

### **Logic bugs**

- Infinite loops and infinite recursion
- Off by one error, counting one too many or too few when looping

## **Syntax bugs**

- Use of the wrong operator, such as performing assignment instead of equality test. In simple cases often warned by the compiler; in many languages, deliberately guarded against by language syntax

## **Resource bugs**

- Null pointer dereference
- Using an uninitialized variable
- Using an otherwise valid instruction on the wrong data type
- Access violations
- Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
- Buffer overflow, in which a program tries to store data past the end of allocated storage. This may or may not lead to an access violation or storage violation. These bugs can form a security vulnerability.
- Excessive recursion which though logically valid causes stack overflow

## **Multi-threading programming bugs**

- Deadlock
- Race condition
- Concurrency errors in critical sections, mutual exclusions and other features of concurrent processing. Time-of-check-to-time-of-use (TOCTOU) is a form of unprotected critical section.

## **Interfacing bugs**

- Incorrect API usage
- Incorrect protocol implementation
- Incorrect hardware handling

## **Teamworking bugs**

- Unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm. These errors are mitigated by the Don't Repeat Yourself philosophy.
- Comments out of date or incorrect: many programmers assume the comments accurately describe the code
- Differences between documentation and the actual product

## Chapter-4

# Fault-Tolerant System and Feature Interaction Problem

## Fault-tolerant system

**Fault-tolerance** or **graceful degradation** is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naïvely-designed system in which even a small failure can cause total breakdown. Fault-tolerance is particularly sought-after in high-availability or life-critical systems.

Fault-tolerance is not just a property of individual machines; it may also characterise the rules by which they interact. For example, the Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to *expect* packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount.

<i>Anti-aliasing made easy</i>	<i>Anti-aliasing made easy</i>
↑	↑
<i>Anti-aliasing made easy</i>	
	<i>Anti-aliasing made easy</i>
↓	↓
<i>Anti-aliasing made easy</i>	

An example of graceful degradation by design in an image with transparency. The top two images are each the result of viewing the composite image in a viewer that recognises transparency. The bottom two images are the result in a viewer with no support for transparency. Because the transparency mask (centre bottom) is discarded, only the overlay (centre top) remains; the image on the left has been designed to degrade gracefully, hence is still meaningful without its transparency information.

Data formats may also be designed to degrade gracefully. HTML for example, is designed to be forward compatible, allowing new HTML entities to be ignored by Web browsers which do not understand them without causing the document to be unusable.

Recovery from errors in fault-tolerant systems can be characterised as either **roll-forward** or **roll-back**. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery reverts the system state back to some earlier, correct version, for example using checkpointing, and moves forward from there. Roll-back recovery requires that the operations between the checkpoint and the detected erroneous state can be made idempotent. Some systems make use of both roll-forward and roll-back recovery for different errors or different parts of one error.

Within the scope of an *individual* system, fault-tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of making it sufficiently reliable is very high, a better solution may be to use some form of duplication. In any case, if the consequence of a system failure is catastrophic, the system must be able to use reversion to fall back to a safe mode. This is similar to roll-back recovery but can be a human action if humans are present in the loop.

## ***Fault tolerance requirements***

The basic characteristics of fault tolerance require:

1. No single point of failure
2. Fault isolation to the failing component
3. Fault containment to prevent propagation of the failure
4. Availability of reversion modes

In addition, fault tolerant systems are characterized in terms of both planned service outages and unplanned service outages. These are usually measured at the application level and not just at a hardware level. The figure of merit is called availability and is expressed as a percentage. For example, a five nines system would statistically provide 99.999% availability.

Fault-tolerant systems are typically based on the concept of redundancy.

## ***Fault-tolerance by replication***

Spare components addresses the first fundamental characteristic of fault-tolerance in three ways:

- Replication: Providing multiple identical instances of the same system or subsystem, directing tasks or requests to all of them in parallel, and choosing the correct result on the basis of a quorum;
- Redundancy: Providing multiple identical instances of the same system and switching to one of the remaining instances in case of a failure (failover);
- Diversity: Providing multiple *different* implementations of the same specification, and using them like replicated systems to cope with errors in a specific implementation.

All implementations of RAID, redundant array of independent disks, except RAID 0 are examples of a fault-tolerant storage device that uses data redundancy.

A lockstep fault-tolerant machine uses replicated elements operating in parallel. At any time, all the replications of each element should be in the same state. The same inputs are provided to each replication, and the same outputs are expected. The outputs of the replications are compared using a voting circuit. A machine with two replications of each element is termed Dual Modular Redundant (DMR). The voting circuit can then only detect a mismatch and recovery relies on other methods. A machine with three replications of each element is termed Triple Modular Redundancy (TMR). The voting circuit can determine which replication is in error when a two-to-one vote is observed. In this case, the voting circuit can output the correct result, and discard the erroneous version. After this, the internal state of the erroneous replication is assumed to be different from that of the other two, and the voting circuit can switch to a DMR mode. This model can be applied to any larger number of replications.

Lockstep fault tolerant machines are most easily made fully synchronous, with each gate of each replication making the same state transition on the same edge of the clock, and the clocks to the replications being exactly in phase. However, it is possible to build lockstep systems without this requirement.

Bringing the replications into synchrony requires making their internal stored states the same. They can be started from a fixed initial state, such as the reset state. Alternatively, the internal state of one replica can be copied to another replica.

One variant of DMR is **pair-and-spare**. Two replicated elements operate in lockstep as a pair, with a voting circuit that detects any mismatch between their operations and outputs a signal indicating that there is an error. Another pair operates exactly the same way. A final circuit selects the output of the pair that does not proclaim that it is in error. Pair-and-spare requires four replicas rather than the three of TMR, but has been used commercially.

## ***No single point of repair***

If a system experiences a failure, it must continue to operate without interruption during the repair process.

## ***Fault isolation to the failing component***

When a failure occurs, the system must be able to isolate the failure to the offending component. This requires the addition of dedicated failure detection mechanisms that exist only for the purpose of fault isolation.

Recovery from a fault condition requires classifying the fault or failing component. The National Institute of Standards and Technology (NIST) categorizes faults based on Locality, Cause, Duration and Effect.

## ***Fault containment***

Some failure mechanisms can cause a system to fail by propagating the failure to the rest of the system. An example of this kind of failure is the "Rogue transmitter" which can swamp legitimate communication in a system and cause overall system failure. Mechanisms that isolate a rogue transmitter or failing component to protect the system are required.

# **Feature interaction problem**

**Feature interaction** is a software engineering concept. It occurs when the integration of two **features** would modify the **behavior** of one or both features.

The term *feature* is used to denote a unit of functionality of a software application. Similar to many concepts in computer science, the term can be used at different levels of abstraction. For example, the plain old telephone service (POTS) is a telephony application feature at one level, but itself is composed of originating features and terminating features. The originating features may in turn include the provide dial tone feature, digit collection feature and so on.

This definition of *feature interaction* allows one to focus on certain behavior of the interacting features such as how their response time may be changed given the integration. Many researchers in the field consider problems that arise due to change in the execution *behavior* of the interacting features. Under that context, the *behavior* of a feature is defined by its execution flow and output for a given input. In other words, the interaction changes the execution flow and output of the interacting features for a given input.

## **Example**

In the context of telephony, a telephone line (the system) typically offers a set of features that include call forwarding and call waiting. Call waiting allows one call to be suspended while a second call is answered, while call forwarding enables a customer to specify a secondary phone number to which additional calls will be forwarded in the event that the customer is already using the phone.

To illustrate the example, we consider a telephone line provided to a customer, and we assume that both call forwarding and call waiting are enabled on the line. When a first call arrives on the line, the phone rings and is answered. Since neither feature is activated by the first call, there is no noticeable problem. When a second call arrives before the first has terminated, the telephone system has a decision to make: whether the call should be forwarded to the secondary number (call forwarding) or the person who answered the first call should be notified that another call has arrived (call waiting). Since this decision has no obvious correct answer, the optimal answer depends on the needs of the customer. This *feature interaction* is a specific example of a general and common problem that has become prevalent due to increasing system complexity.

In this situation, it is possible that the system's decision will be made in a non-deterministic fashion due to race conditions and other design factors. The consequences of feature interactions can range from minor irritations to life-threatening software failures, and therefore there is ongoing research that aims to find ways of *detecting* as well as *resolving* feature interactions.

## Chapter-5

# Independent Software Verification and Validation & Software Assurance

## Independent software verification and validation

ISVV stands for **Independent Software Verification and Validation**. ISVV is targeted at safety-critical software systems and aims to increase the quality of software products, thereby reducing risks and costs through the operational life of the software. ISVV provides assurance that software performs to the specified level of confidence and within its designed parameters and defined requirements.

ISVV activities are performed by independent engineering teams, not involved in the software development process, to assess the processes and the resulting products. The ISVV team independency is performed at three different levels: financial, managerial and technical.

ISVV goes far beyond “traditional” verification and validation techniques, applied by development teams. While the latter aim to ensure that the software performs well against the nominal requirements, ISVV is focused on non-functional requirements such as robustness and reliability, and on conditions that can lead the software to fail. ISVV results and findings are fed back to the development teams for correction and improvement.

### ***ISVV History***

ISVV derives from the application of IV&V (Independent Verification and Validation) to the software. Early ISVV application (as known today) dates back to the early 1970s when the U.S. Army sponsored the first significant program related to IV&V for the Safeguard Anti-Ballistic Missile System.

By the end of the 1970s IV&V was rapidly becoming popular. The constant increase in complexity, size and importance of the software lead to an increasing demand on IV&V applied to software (ISVV).

Meanwhile IV&V (and ISVV for software systems) gets consolidated and is now widely used by organisations such as the DoD, FAA, NASA and ESA. IV&V is mentioned in [DO-178B], [ISO/IEC 12207] and formalised in [IEEE 1012].

Initially in 2004-2005, a European consortium led by the European Space Agency, and composed by DNV(N), Critical Software SA(P), Terma(DK) and CODA Scisys(UK) created the first version of a guide devoted to ISVV, called "ESA Guide for Independent Verification and Validation" with support from other organizations, eg SoftWcare SL (E), etc.

In 2008 the European Space Agency released a second version, being SoftWcare SL was the supporting editor having received inputs from many different European Space ISVV stakeholders. This guide covers the methodologies applicable to all the software engineering phases in what concerns ISVV.

## ***ISVV Methodology***

ISVV is usually composed by five principal phases, these phases can be executed sequentially or as results of a tailoring process.

### *ISVV Planning*

- Planning of ISVV Activities
- System Criticality Analysis: Identification of Critical Components through a set of RAMS activities (Value for Money)
- Selection of the appropriate Methods and Tools

### *Requirements Verification*

- Verification for: Completeness, Correctness, Testability

### *Design Verification*

- Design adequacy and conformance to Software Requirements and Interfaces
- Internal and External Consistency
- Verification of Feasibility and Maintenance

### *Code Verification*

- Verification for: Completeness, Correctness, Consistency
- Code Metrics Analysis
- Coding Standards Compliance Verification

## *Validation*

- Identification of unstable components/functionalities
- Validation focused on Error-Handling: complementary (not concurrent!) validation regarding the one performed by the Development team (More for the Money, More for the Time)
- Compliance with Software and System Requirements
- Black box testing and White box testing techniques
- Experience based techniques

## **Software assurance**

**Software Assurance** (SwA) is defined as “the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle, and that the software functions in the intended manner.”

### ***Alternate definitions***

#### **Department of Homeland Security (DHS)**

According to the DHS, software assurance addresses:

- Trustworthiness - No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended;
- Conformance - Planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements, standards/ procedures.

Contributing SwA disciplines, articulated in Bodies of Knowledge and Core Competencies: Software Engineering, Systems Engineering, Information Systems Security Engineering, Information Assurance, Test and Evaluation, Safety, Security, Project Management, and Software Acquisition.

Software Assurance is a strategic initiative of the U.S. Department of Homeland Security (DHS) to promote integrity, security, and reliability in software. The SwA Program is based upon the National Strategy to Secure Cyberspace - Action/Recommendation 2-14:

“DHS will facilitate a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code

development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development.”

### **United States Department of Defense (DoD)**

According to the DoD, software assurance relates to "the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software."

### **Software Assurance Metrics and Tool Evaluation (SAMATE) project**

According to the NIST SAMATE project, software assurance is "the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures to help achieve:

- Trustworthiness - No exploitable vulnerabilities exist, either of malicious or unintentional origin, and
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended."

### **National Aeronautics and Space Administration (NASA)**

According to NASA, Software Assurance is a "planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. It includes the disciplines of Quality Assurance, Quality Engineering, Verification and Validation, Nonconformance Reporting and Corrective Action, Safety Assurance, and Security Assurance and their application during a software life cycle." The NASA Software Assurance Standard also states: "The application of these disciplines during a software development life cycle is called Software Assurance."

### **Object Management Group (OMG)**

According to the OMG, Software Assurance is “justifiable trustworthiness in meeting established business and security objectives.”

OMG's SwA Special Interest Group (SIG), works with Platform and Domain Task Forces and other software industry entities and groups external to the OMG, to coordinate the establishment of a common framework for analysis and exchange of information related to software trustworthiness by facilitating the development of a specification for a Software Assurance Framework that will:

- Establish a common framework of software properties that can be used to represent any/all classes of software so software suppliers and acquirers can represent their claims and arguments(respectively), along with the corresponding evidence, employing automated tools (to address scale)

- Verify that products have sufficiently satisfied these characteristics in advance of product acquisition, so that system engineers/integrators can use these products to build (compose) larger assured systems with them
- Enable industry to improve visibility into the current status of software assurance during development of its software
- Enable industry to develop automated tools that support the common framework.

### **Software Assurance Forum for Excellence in Code (SAFECode)**

According to SAFECode, Software Assurance is “confidence that software, hardware and services are free from intentional and unintentional vulnerabilities and that the software functions as intended.”

### **Webopedia**

According to Webopedia, Software Quality Assurance, abbreviated as SQA, and also called "software assurance", is a level of confidence that software is free from vulnerabilities, either intentionally designed into the software or inserted at anytime during its lifecycle, and that the software functions in the intended manner."

As indicated in the Webopedia definition, the term "software assurance" has been used as a shorthand for Software Quality Assurance (SQA) when not necessarily considering security or trustworthiness. SQA is defined in the *Handbook of Software Quality Assurance* as: "the set of systematic activities providing evidence of the ability of the software process to produce a software product that is fit to use."

## Chapter-6

# Software Rot

**Software rot**, also known as **code rot** or **software erosion** or **software decay** or software entropy, is a type of bit rot. It describes the perceived slow deterioration of software over time that will eventually lead to it becoming faulty, unusable, or otherwise in need of maintenance. This is not a physical phenomenon: the software does not actually decay, but rather suffers from a lack of being updated with respect to the changing environment in which it resides.

Some software can deteriorate in performance over time as it runs and accumulates errors; this is not generally considered software rot, though it may have some of the same consequences. Usually, such a degraded state of software can be remedied by completely reinitializing its state (as by a complete reinstallation of all relevant software components, possibly including operating system software); this may also remedy some kinds of software rot, whereas other software rot is irreversible, as the operating environment of the software, or components of the software itself, have irrevocably changed.

### **Causes**

There are several factors responsible for software rot. These include changes to the environment in which the software operates, degradation of compatibility between parts of the software itself, and the appearance of bugs in unused or rarely used code.

### **Environment change**

When changes occur in the program's environment, particularly changes which the designer of the program did not anticipate, the software may no longer operate as originally intended. For example, many early computer game designers made assumptions about processing speed of the CPU which the games were designed for. A consequence of this was that when the CPU's clock speed was used as a timer in the game, the gameplay speed would increase with that of the CPU, making the software less usable over time.

## **Onceability**

There are changes in the environment not related to the program's designer, but its users. A user could set up the system working once and have it working flawlessly for a time. But when the system stops working correctly, or the users want to access the configuration controls, they are unable to repeat that initial step because of the different context and the unavailable information (password lost, missing instructions, or simply a hard-to-manage user interface that was first configured by trial and error). Information Architect Jonas Söderström has named this concept *Onceability*, and defines it as "the quality in a technical system that prevents a user from restoring the system, once it has failed".

## **Unused code**

Portions of code which are not normally executed, such as document filters or interfaces designed to be used by other programs, may contain bugs that go unnoticed. With changes in user requirements and other external factors, this code may be executed later, thereby exposing the bugs and making the software appear less functional.

## **Rarely updated code**

Normal maintenance of software and systems may also cause software rot. In particular, when a program contains multiple parts which function at arm's length from one another, failing to consider how changes to one part affect the others may introduce bugs.

In some cases, this may take the form of libraries that the software uses being changed in a way which adversely affects the software. If the old version of a library that was previously used with the software cannot be used any longer due to conflicts with other software or security flaws that were found in the old version, there may no longer be a viable version of a needed library for the program to use.

## ***Classification***

Software rot is usually classified as being either **dormant rot** or **active rot**.

### **Dormant rot**

Software that is not currently being used gradually becomes unusable as the remainder of the application changes. Changes in user requirements and the software environment also contribute to the deterioration.

### **Active rot**

Software that is being continuously modified may lose its integrity over time if proper mitigating processes are not consistently applied. However, much software requires continuous changes to meet new requirements and correct bugs, and re-engineering

software each time a change is made is rarely practical. This creates what is essentially an evolution process for the program, causing it to depart from the original engineered design. As a consequence of this and a changing environment, assumptions made by the original designers may be invalidated, introducing bugs.

Developers are often encouraged to fully understand a problem before programming a solution to it, and to keep accurate and complete software documentation. In practice, however, adding new features may be prioritized over updating documentation. Without documentation, however, it is possible for specific knowledge pertaining to parts of the program to be lost. To some extent, this can be mitigated by following best current practices with regards to internal documentation and variable names.

Active software rot slows once an application is near the end of its commercial life and further development ceases. Users often learn to work around any remaining software bugs, and the behaviour of the software becomes consistent as nothing is changing.

### **Example**

Many seminal programs from the early days of AI research have suffered from irreparable software rot. For example, the original SHRDLU program (an early natural language understanding program) cannot be run on any modern day computer or computer simulator, as it was developed during the days when LISP and PLANNER were still in development stage, and thus uses non-standard macros and software libraries which do not exist anymore.

Suppose an administrator creates a forum using phpBB or other online forum software. He then heavily modifies and "hacks" it, adding new features and options. This process requires extensive modifications to existing code and deviation from the original functionality of that software.

From here, there are several ways software rot can affect the system:

- The administrator can accidentally make changes which conflict with each other or the original software, causing the forum to behave unexpectedly or break down altogether. This leaves him in a very bad position: as he has deviated so greatly from the original code, technical support and assistance in reviving the forum will be difficult to obtain.
- A security hole may be discovered in the original forum source code, requiring a security patch. However, because the administrator has modified the code so extensively, the patch may not be directly applicable to his code, requiring the administrator to effectively rewrite the update.
- The administrator who made the modifications could vacate his position, leaving the new administrator with a convoluted and heavily-modified forum that lacks full documentation. Without fully understanding the modifications, it is difficult for the new administrator to make changes without introducing conflicts and bugs. (Furthermore, documentation of the original system might no longer be available;

it might have been abandoned, become proprietary closed software, or, with the passage of enough time, been lost.)

## ***Refactoring***

Refactoring is a means of addressing the problem of software rot. It is described as the process of rewriting existing code to improve its structure without affecting its external behaviour. This includes removing dead code and rewriting sections that have been modified extensively and no longer work efficiently. Care must be taken not to change the software's external behaviour, as this could introduce incompatibilities and thereby itself contribute to software rot.

## Chapter-7

# Reverse Semantic Traceability

**Reverse Semantic Traceability (RST)** is a quality control method for verification improvement that helps to insure high quality of artifacts by backward translation at each stage of the software development process.

### ***Brief introduction***

Each stage of development process can be treated as a series of “translations” from one language to another. At the very beginning a project team deals with customer’s requirements and expectations expressed in natural language. These customer requirements sometimes might be incomplete, vague or even contradictory to each other. The first step is specification and formalization of customer expectations, transition (“translation”) of them into a formal requirement document for the future system. Then requirements are translated into system architecture and step by step the project team generates megabytes of code written in a very formal programming language. There is always a threat of inserting mistakes, misinterpreting or losing something during the translation. Even a small defect in requirement or design specifications can cause huge amounts of defects at the late stages of the project. Sometimes such misunderstandings can lead to project failure or complete customer dissatisfaction.

The highest usage scenarios of Reverse Semantic Traceability method can be:

- Validating UML models: quality engineers restore a textual description of a domain, original and restored descriptions are compared.
- Validating model changes for a new requirement: given an original and changed versions of a model, quality engineers restore the textual description of the requirement, original and restored descriptions are compared.
- Validating a bug fix: given an original and modified Source code, quality engineers restore a textual description of the bug that was fixed, original and restored descriptions are compared.
- Integrating new software engineer into a team: a new team member gets an assignment to do Reverse Semantic Traceability for the key artifacts from the current projects.

## ***RST roles***

Main roles involved in RST session are:

- authors of project artifacts (both input and output),
- reverse engineers,
- expert group,
- project manager.

## ***RST process***



### **Define all project artifacts and their relationship**

Reverse Semantic Traceability as a validation method can be applied to any project artifact, to any part of project artifact or even to a small piece of document or code. However, it is obvious that performing RST for all artifacts can create overhead and should be well justified (for example, for medical software where possible information loss is very critical).

It is a responsibility of company and Project manager to decide what amount of project artifacts will be “reverse engineered”. This amount depends on project specific details: trade-off matrix, project and company quality assurance policies. Also it depends on importance of particular artifact for project success and level of quality control applied to this artifact.

Amount of RST sessions for project is defined at the project planning stage.

First of all project manager should create a list of all artifacts project team will have during the project. They could be presented as a tree with dependencies and relationships. Artifacts can be present in one occurrence (like Vision document) or in several occurrences (like risks or bugs). This list can be changed later during the project but the idea behind the decisions about RST activities will be the same.

## **Prioritize**

The second step is to analyze deliverable importance for project and level of quality control for each project artifact.

Importance of document is the degree of artifact impact to project success and quality of final product. It's measured by the scale:

- Crucial (1): the quality of deliverable is very important for overall quality of project and even for project success. Examples: Functional requirements, System architecture, critical bug fixes (show stoppers), risks with high probability and critical impact.
- High (2): the deliverable has an impact to quality of final product. Examples: Test cases, User interface requirements, major severity bug fixes, risks with high expose.
- Medium (3): the artifact has a medium or indirect impact to quality of final product. Examples: Project plan, medium severity bug fixes, risks with medium expose.
- Low (4): the artifact has insignificant impact to the final product quality. Example: employees' tasks, cosmetic bugs, risks with low probability.

Level of quality control is a measure that defines amount of verification and validation activities applied to artifact, and probability of miscommunication during artifact creation.

- Low (1): No review is supposed for the artifact, miscommunication and information loss are high probable, information channel is distributed, language barrier exists etc
- Medium (2): No review is supposed for the artifact, information channel is not distributed (e.g. creator of artifact and information provider are members of one team)
- Sufficient (3): Pair development or peer review is done, information channel is not distributed.
- Excellent (4): Pair development, peer review and/or testing are done, automation or unit testing is done, or there are some tools for artifact development and validation.

## **Define responsible people**

Success of RST session strongly depends on correct assignment of responsible people.

## **Perform Reverse Semantic Traceability of artifact**

Reverse Semantic Traceability starts when decision that RST should be performed is made and resources for it are available.

Project manager defines what documents will be an input for RST session. For example, it can be not only an artifact to restore but some background project information. It is recommended to give to reverse engineers number of words in original text so that they have an idea about amount of text they should get as a result: it can be one sentence or several pages of text. Though, the restored text may not contain the same number of words as original text nevertheless the values should be comparable.

After that reverse engineers take the artifact and restore the original text from it. RST itself takes about 1 hour for one page text (750 words).

## **Value the level of quality and make a decision**

To complete RST session, restored and original texts of artifact should be compared and quality of artifact should be assessed. Decision about artifacts rework and its amount is made based on this assessment.

For assessment a group of experts is formed. Experts should be aware of project domain and be an experienced enough to assess quality level of compared artifacts. For example, business analysts will be experts for comparison of vision statement and restored vision statement from scenario.

RST assessment criteria:

1. Restored and original texts have quite big differences in meaning and crucial information loss
2. Restored and original texts have some differences in meaning, important information loss
3. Restored and original texts have some differences in meaning, some insignificant information loss
4. Restored and original texts are very close, some insignificant information loss
5. Restored and original texts are very close, none information is lost

Each of experts gives his assessment, and then the average value is calculated. Depending on this value Project Manager makes a decision should both artifacts be corrected or one of them or rework is not required.

If the average RST quality level is in range from 1 to 2 the quality of artifact is poor and it is recommended not only rework of validated artifact to eliminate defects but corrections of original artifact to clear misunderstandings. In this case one more RST session after rework of artifacts is required. For artifacts that have more than 2 but less than 3 corrections of validated artifact to fix defects and eliminate information loss is required, however review of original artifact to find out if there any vague piece of information that cause misunderstandings is recommended. No additional RST sessions is needed. If the average mark is more than 3 but less than 4 then corrections of validated artifact to remove defects and insignificant information loss is supposed. If the mark is greater than 4 it means that artifact is of good quality and no special corrections or rework is required.

Obviously the final decision about rework of artifacts is made by project manager and should be based on analysis of reasons of differences in texts.

## Chapter-8

# Sneak Circuit Analysis & Verification and Validation (Software)

## Sneak circuit analysis

**Sneak Circuit Analysis** is a vital part of the safety assurance of safety-critical electronic and electro-mechanical systems.

Sneak conditions are defined as latent hardware, software, or integrated conditions that may cause unwanted actions or may inhibit a desired function, and are not caused by component failure.

Sneak Circuit Analysis (SCA) is used in safety-critical systems to identify sneak (or hidden) paths in electronic circuits and electro-mechanical systems that may cause unwanted action or inhibit desired functions. The analysis is aimed at uncovering design flaws that allow for sneak conditions to develop. The sneak circuit analysis technique differs from other system analysis techniques in that it is based on identification of designed-in inadvertent modes of operation and is not based on failed equipment or software.

SCA is most applicable to circuits that can cause irreversible events. These include:

- a. Systems that control or perform active tasks or functions
- b. Systems that control electrical power and its distribution.
- c. Embedded code which controls and times system functions.

Sneak conditions are classified into four basic types:

1. Sneak paths - unintended electrical (current) paths within a circuit and its external interfaces.
2. Sneak timing—unexpected interruption or enabling of a signal due to switch circuit timing problems which may cause or prevent the activation or inhibition of a function at an unexpected time.
3. Sneak indications—undesired activation or deactivation of an indicator which may cause an ambiguous or false display of system operating conditions.

4. Sneak labels—incorrect or ambiguous labeling of a switch which may cause operator error through inappropriate control activation.

### ***Historical background***

SCA is a detailed examination of switching circuitry that controls irreversible functions such as squibs and latches. The former Military Standard for Reliability Program (MIL-STD-785B) defines SCA (Task 205) as a task “ ... to identify latent paths which cause occurrence of unwanted functions or inhibit desired functions, assuming all components are functioning properly.”

The Mercury-Redstone launch failure 1961 and a number of other mishaps caused by sneak circuits in missiles and torpedoes caused the military services and NASA to require formal procedures for prevention of these incidents. The first computer aided implementation of SCA was for the NASA Apollo program in 1967 by the Boeing Company. Among early publications in the field are a 1970 Boeing report “Sneak Circuit Analysis Handbook” by J. P. Rankin and C. F. White (NTIS N71-12487), and a 1977 AGARD report by J. L. Wilson and R. C. Clardy “Sneak Circuit Analysis Application to Control System Design” (AD A041042). By 1980 the requirements for SCA had become sufficiently common to lead to the Navy publication of a “Contract and Management Guide for Sneak Circuit Analysis” (NAVSEA-TE001-AA-GYD-010/SCA) . Subsequent efforts led to several Air Force reports in 1990: "Sneak Circuit Analysis for the Common Man", Rome Air Development Center Technical Report, RADC-TR-89-223, October, 1989 and "Intergration of Sneak Analysis with Design", Air Development Center Technical Report, RADC-TR-90-109, (June, 1990.).

Current standards and guidelines include NASA’s Sneak Circuit Analysis Guideline for Electromechanical Systems (PD-AP-1314) and AIAA’s Performance-Based Sneak Circuit Analysis (SCA) Requirements (BSR/ANSI/AIAA S-102.2.5-2xxx).

### ***Sneak circuit example***

Most sneak circuits reported from production systems are too complex to describe in an introductory discussion. However, the essential characteristics of a sneak circuit can be explained with a hypothetical example of an aircraft cargo door release latch as shown in Figure 1-1.

Figure 1-1 Sneak Circuit in Cargo Door Latching Function

To prevent unintended opening of the cargo door in flight, the normal cargo door control (CARGO OPEN) is powered in series with the GEAR DOWN switch. This permits routine opening on the ground. But there can be emergencies that require jettisoning cargo, and to be prepared for these there is an EMERGENCY CARGO OPEN switch that may be guarded with a safety wire to prevent its unintended operation. Now assume that an in-flight emergency exists that requires opening the cargo door. The flight personnel flips the normal CARGO OPEN switch and nothing happens (since the GEAR DOWN

switch is open). It is realized that it is necessary to close the EMERGENCY CARGO OPEN switch, and when that action is taken the cargo door latch is indeed released, permitting the door to be opened. But at the same time the landing gear is lowered, not a desired action and one that probably will aggravate the emergency. The condition that permits this undesired lowering of the landing gear to occur when both cargo door switches are closed is a sneak circuit.

Two observations about this sneak circuit apply generally:

1. Switches or other control elements are operated in an unusual or even prohibited manner
2. The unintended function (in this example the lowering of the landing gear) is associated with current flow through a circuit element that is opposite to the intended current flow.

The latter of these conditions permits elimination of the sneak circuit by inserting a diode as shown in Figure 1-2.

Figure 1-2. Corrected Cargo Door Latching Circuit

### ***Conventional SCA Techniques***

The original SCA techniques depended on recognition of circuit patterns or “clues” for the detection of potential sneak circuits. The most common of these circuit patterns are shown in Figure 1-3.

Figure 1-3 Circuit Patterns for Sneak Circuit Analysis

The box symbols represent arbitrary circuit elements; in many cases the individual legs of the patterns include switches. It will be recognized that the leg containing the normal CARGO OPEN switch in Figure 1-1 constitutes the middle horizontal leg of an H-pattern. The inverted Y is also called a ground dome; note that the two bottom legs terminate in different ground levels, such as chassis ground and signal ground. The Y-pattern is also called a power dome. The two upper legs terminate at different power sources, such as V1 and V2.

To facilitate the recognition of these patterns or clues, the schematic diagrams were redrawn as “network trees”, with power sources at the top and grounds at the bottom. In sneak circuit analysis both positive and negative sources will be shown at the top of the figure. Because searching for the patterns is very labor intensive, computer programs were developed to recognize the common clues in the network trees. Main frame computers had to be utilized to perform the topological searches even on small designs. Despite the aid of computers, SCA remained a very expensive and lengthy activity, and it was usually conducted only after the circuit design was frozen to avoid having to repeat it after changes. This had a distinct disadvantage when a sneak circuit was detected: it

became very expensive to fix it because usually the circuit card or cabling was already in production.

An effort of the Rome Air Development Center (now part of the USAF Research Laboratories) directed at finding techniques that would permit sneak circuit analysis to be conducted as part of the design activity led to the “bi-path” methodology, developed by SoHaR. The “bi-path” algorithm was implemented into an automated software tool that allowed large designs to be analyzed very quickly early in the design phase of a safety-critical circuit.

## ***Editing***

Editing is used to eliminate paths that cannot contribute to operation of sensitive elements (elements that can lead to critical actions). Circuits that control squibs or latches usually contain computational, instrumentation, and switching elements. An example of the integration of these functions for a hypothetical and simplified missile detonation system is shown in Figure 1-4. The computational elements at the top of the figure establish the conditions for operation of the pre-arm, arm, and detonate switches. The heavy lines constitute the switching elements. The instrumentation functions are shown in the lower part of the figure. Sneak circuit analysis encompasses only the switching functions; the computational and instrumentation elements are eliminated from the traced paths.

This editing is justified because the connection between the computational elements and the switches (shown as dashed lines in the figure) is non-conducting. In most cases the output of the computational element goes to the gate of a MOSFET while the switching function uses the source-drain path. The computational elements are typically quite complex and their failure probability is much higher than that of the switching path. Thus safeguards are provided to tolerate the worst failure modes of these devices and sneak circuit analysis of the computational elements is not required.

Figure 1-4. Hypothetical Missile Detonation System

The elimination of the instrumentation functions is justified by the isolation resistors at the connection with the switching function. The resistance values are typically of the order of 10k ohms. Since the switching voltage is in the 20V–30V range, the current flow through the isolation resistors cannot exceed a few milliamperes, while squibs fire only above 1 ampere. In addition to this editing of major blocks, individual elements connected to the switching circuit may have to be eliminated or modified by editing as shown in the examples of Figure 1-5. The feedback resistor  $R_f$  constitutes an intentional bi-path (not a sneak circuit). Its high resistance prevents significant current flow. In part b. of the figure a mechanical connection keeps switches S1 and S2 from being closed at the same time, preventing a power-to-power tie.

Figure 1-5. Editing for Intentional and Irrelevant Paths

1. Sneak Circuit Analysis for the Common Man.

2. Integration of Sneak Analysis with Design.
3. Sneak Circuit Analysis.
4. Sneak Circuit Analysis Automated Tool, SCAT.

## Verification and Validation (software)

In software project management, software testing, and software engineering, **Verification and Validation (V&V)** is the process of checking that a software system meets specifications and that it fulfils its intended purpose. It is normally part of the software testing process of a project.

### **Definitions**

Also known as software quality control.

Validation checks that the product design satisfies or fits the intended usage (high-level checking) — i.e., you built the right product. This is done through dynamic testing and other forms of review.

According to the Capability Maturity Model (CMMI-SW v1.1),

- **Verification:** The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-STD-610].
- **Validation:** The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. [IEEE-STD-610]

In other words, validation ensures that the product actually meets the user's needs, and that the specifications were correct in the first place, while verification is ensuring that the product has been built according to the requirements and design specifications. Validation ensures that 'you built the right thing'. Verification ensures that 'you built it right'. Validation confirms that the product, as provided, will fulfill its intended use.

From testing perspective:

- **Fault** - wrong or missing function in the code.
- **Failure** - the manifestation of a fault during execution.
- **Malfunction** - according to its specification the system does not meet its specified functionality.

Within the modeling and simulation community, the definitions of validation, verification and accreditation are similar:

- Validation is the process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s).
- Accreditation is the formal certification that a model or simulation is acceptable to be used for a specific purpose.
- Verification is the process of determining that a computer model, simulation, or federation of models and simulations implementations and their associated data accurately represents the developer's conceptual description and specifications.

### ***Related concepts***

Both verification and validation are related to the concepts of quality and of software quality assurance. By themselves, verification and validation do not guarantee software quality; planning, traceability, configuration management and other aspects of software engineering are required.

### ***Classification of methods***

In mission-critical systems where flawless performance is absolutely necessary, formal methods can be used to ensure the correct operation of a system. However, often for non-mission-critical systems, formal methods prove to be very costly and an alternative method of V&V must be sought out. In this case, syntactic methods are often used.

### **Test cases**

A test case is a tool used in the process.

Test cases are prepared for verification: to determine if the process that was followed to develop the final product is right.

Test case are executed for validation: if the product is built according to the requirements of the user. Other methods, such as reviews, are used when used early in the Software Development Life Cycle provide for validation.

### ***Independent Verification and Validation***

Verification and validation often is carried out by a separate group from the development team; in this case, the process is called "**Independent Verification and Validation**", or **IV&V**.

## ***Regulatory environment***

Verification and validation must meet the compliance requirements of law regulated industries, which is often guided by government agencies or industrial administrative authorities. e.g. The FDA requires software versions and patches to be validated.

## Chapter-9

# Software Security Assurance

**Software Security Assurance** is a process that helps design and implement software that protects the data and resources contained in and controlled by that software. Software is itself a resource and thus must be afforded appropriate security.

### ***What is Software Security Assurance?***

Software Security Assurance (SSA) is the process of ensuring that software is designed to operate at a level of security that is consistent with the potential harm that could result from the loss, inaccuracy, alteration, unavailability, or misuse of the data and resources that it uses, controls, and protects.

The Software Security Assurance process begins by identifying and categorizing the information that is to be contained in, or used by, the software. The information should be categorized according to its sensitivity. For example, in the lowest category, the impact of a security violation is minimal (i.e. the impact on the software owner's mission, functions, or reputation is negligible). For a top category, however, the impact may pose a threat to human life; may have an irreparable impact on software owner's missions, functions, image, or reputation; or may result in the loss of significant assets or resources.

Once the information is categorized, security requirements can be developed. The security requirements should address access control, including network access and physical access; data management and data access; environmental controls (power, air conditioning, etc.) and off-line storage; human resource security; and audit trails and usage records.

### ***What causes software security problems?***

All security vulnerabilities in software are the result of security bugs, or defects, within the software. In most cases, these defects are created by two primary causes: (1) non-conformance, or a failure to satisfy requirements; and (2) an error or omission in the software requirements.

## **Non-conformance, or a failure to satisfy requirements**

A non-conformance may be simple—the most common is a coding error or defect—or more complex (i.e., a subtle timing error or input validation error). The important point about non-conformances is that verification and validation techniques are designed to detect them and security assurance techniques are designed to prevent them. Improvements in these methods, through a software security assurance program, can improve the security of software.

## **Errors or omissions in software requirements**

The most serious security problems with software-based systems are those that develop when the software requirements are incorrect, inappropriate, or incomplete for the system situation. Unfortunately, errors or omissions in requirements are more difficult to identify. For example, the software may perform exactly as required under normal use, but the requirements may not correctly deal with some system state. When the system enters this problem state, unexpected and undesirable behavior may result. This type of problem cannot be handled within the software discipline; it results from a failure of the system and software engineering processes which developed and allocated the system requirements to the software.

## ***Software Security Assurance Activities***

There are two basic types of Software Security Assurance activities.

1. Some focus on ensuring that information processed by an information system is assigned a proper sensitivity category, and that the appropriate protection requirements have been developed and met in the system.
2. Others focus on ensuring the control and protection of the software, as well as that of the software support tools and data.

At a minimum, a Software Security Assurance program should ensure that:

1. A security evaluation has been performed for the software.
2. Security requirements have been established for the software.
3. Security requirements have been established for the software development and/or operations and maintenance (O&M) processes.
4. Each software review, or audit, includes an evaluation of the security requirements.
5. A configuration management and corrective action process is in place to provide security for the existing software and to ensure that any proposed changes do not inadvertently create security violations or vulnerabilities.
6. Physical security for the software is adequate.

## ***Building in security***

Improving the software development process and building better software are ways to improve software security, by producing software with fewer defects and vulnerabilities. A first-order approach is to identify the critical software components that control security-related functions and pay special attention to them throughout the development and testing process. This approach helps to focus scarce security resources on the most critical areas.

## **Tools and techniques**

There are many commercial off-the-shelf (COTS) software packages that are available to support software security assurance activities. However, before they are used, these tools must be carefully evaluated and their effectiveness must be assured.

## **Common weaknesses enumeration**

One way to improve software security is to gain a better understanding of the most common weaknesses that can affect software security. With that in mind, there is a current community-based program called the Common Weaknesses Enumeration project, which is sponsored by The Mitre Corporation to identify and describe such weaknesses. The list, which is currently in a very preliminary form, contains descriptions of common software weaknesses, faults, and flaws.

## **Security architecture/design analysis**

Security Architecture/Design Analysis verifies that the software design correctly implements security requirements. Generally speaking, there are four basic techniques that are used for security architecture/design analysis.

### **Logic analysis**

Logic analysis evaluates the equations, algorithms, and control logic of the software design.

### **Data analysis**

Data analysis evaluates the description and intended usage of each data item used in design of the software component. The use of interrupts and their effect on data should receive special attention to ensure interrupt handling routines do not alter critical data used by other routines.

## **Interface analysis**

Interface analysis verifies the proper design of a software component's interfaces with other components of the system, including hardware, software, and end-users.

## **Constraint analysis**

Constraint analysis evaluates the design of a software component against restrictions imposed by requirements and real-world limitations. The design must be responsive to all known or anticipated restrictions on the software component. These restrictions may include timing, sizing, and throughput constraints, input and output data limitations, equation and algorithm limitations, and other design limitations.

## **Secure code reviews, inspections, and walkthroughs**

Code analysis verifies that the software source code is written correctly, implements the desired design, and does not violate any security requirements. Generally speaking, the techniques used in the performance of code analysis mirror those used in design analysis.

Secure Code Reviews are conducted during and at the end of the development phase to determine whether established security requirements, security design concepts, and security-related specifications have been satisfied. These reviews typically consist of the presentation of material to a review group. Secure Code Reviews are most effective when conducted by personnel who have not been directly involved in the development of the software being reviewed.

## **Informal reviews**

Informal secure code reviews can be conducted on an as-needed basis. To conduct an informal review, the developer simply selects one or more reviewer(s) and provides and/or presents the material to be reviewed. The material may be as informal as pseudo-code or hand-written documentation.

## **Formal reviews**

Formal secure code reviews are conducted at the end of the development phase for each software component. The client of the software appoints the formal review group, who may make or affect a "go/no-go" decision to proceed to the next step of the software development life cycle.

## **Inspections and walkthroughs**

A secure code inspection or walkthrough is a detailed examination of a product on a step-by-step or line-by-line (of source code) basis. The purpose of conducting secure code inspections or walkthroughs is to find errors. Typically, the group that does an inspection

or walkthrough is composed of peers from development, security engineering and quality assurance.

## **Security testing**

Software security testing, which includes penetration testing, confirms the results of design and code analysis, investigates software behavior, and verifies that the software complies with security requirements. Special security testing, conducted in accordance with a security test plan and procedures, establishes the compliance of the software with the security requirements. Security testing focuses on locating software weaknesses and identifying extreme or unexpected situations that could cause the software to fail in ways that would cause a violation of security requirements. Security testing efforts are often limited to the software requirements that are classified as "critical" security items.

## Chapter-10

# Algorithmic Efficiency

In computer science, **efficiency** is used to describe properties of an algorithm relating to how much of various types of resources it consumes. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process, where the goal is to reduce resource consumption, including time to completion, to some acceptable, optimal level.

### **Software metrics**

The two most frequently encountered and measurable metrics of an algorithm are:-

- speed or running time - the time it takes for an algorithm to complete, and
- 'space' - the memory or 'non-volatile storage' used by the algorithm **during its operation**.

but also might apply to

- transmission size - such as required bandwidth during normal operation or
- size of external memory- such as temporary disk space used to accomplish its task

and perhaps even

- the size of required 'longterm' disk space required **after its operation** to record its output or maintain its required function during its required useful lifetime (examples: a data table, archive or a computer log) and also
- the performance per watt and the total energy, consumed by the chosen hardware implementation (with its System requirements, necessary auxiliary support systems including interfaces, cabling, switching, cooling and security), during its required useful lifetime.

(An extreme example of these metrics might be to consider their values in relation to a repeated simple algorithm for calculating and storing  $(\pi+n)$  to 50 decimal places running

for say, 24 hours, either on a "pocket calculator" sized processor such as an ipod or an early mainframe operating in its own purpose-built heated or air conditioned unit.)

The process of making code more efficient is known as optimization and in the case of automatic optimization (i.e. compiler optimization - performed by compilers on request or by default), usually focus on space at the cost of speed, or vice versa. There are also quite simple programming techniques and 'avoidance strategies' that can actually improve both at the same time, usually irrespective of hardware, software or language. Even the re-ordering of nested conditional statements - to put the least frequently occurring condition first (example: test patients for blood type = 'AB-', before testing age > 18, since this type of blood occurs in only about 1 in 100 of the population - thereby eliminating the second test at runtime in 99% of instances), can reduce actual instruction path length, something an optimizing compiler would almost certainly not be aware of - but which a programmer can research relatively easily even without specialist medical knowledge.

## **History**

The first machines that were capable of computation were severely limited by purely mechanical considerations. As later electronic machines were developed they were, in turn, limited by the speed of their electronic counterparts. As software replaced hard-wired circuits, the efficiency of algorithms also became important. It has long been recognized that the precise 'arrangement of processes' is critical in reducing elapse time.

- "In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation"

Ada Lovelace 1815-1852, generally considered as 'the first programmer' who worked on Charles Babbage's early mechanical general-purpose computer

- "In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"

Extract from "Structured Programming with go to Statements" by Donald Knuth, renowned computer scientist and Professor Emeritus of the Art of Computer Programming at Stanford University.

- "The key to performance is elegance, not battalions of special cases"

attributed to Jon Bentley and (Malcolm) Douglas McIlroy

## **Speed**

The **absolute** speed of an algorithm for a given input can simply be measured as the duration of execution (or clock time) and the results can be averaged over several executions to eliminate possible random effects. Most modern processors operate in a multi-processing & multi-programming environment so consideration must be made for parallel processes occurring on the same physical machine, eliminating these as far as possible. For superscalar processors, the speed of a given algorithm can sometimes be improved through instruction-level parallelism within a single processor (but, for optimal results, the algorithm may require some adaptation to this environment to gain significant advantage ('speedup'), becoming, in effect, an entirely different algorithm). A **relative** measure of an algorithm's performance can sometimes be gained from the total instruction path length which can be determined by a run time Instruction Set Simulator (where available).

An **estimate** of the speed of an algorithm can be determined in various ways. The most common method uses time complexity to determine the Big-O of an algorithm.

## **Memory**

Often, it is possible to make an algorithm faster at the expense of memory. This might be the case whenever the result of an 'expensive' calculation is cached rather than recalculating it afresh each time. The additional memory requirement would, in this case, be considered additional overhead although, in many situations, the stored result occupies very little extra space and can often be held in pre-compiled static storage, reducing not just processing time but also allocation & deallocation of working memory. This is a very common method of improving speed, so much so that some programming languages often add special features to support it, such as C++'s 'mutable' keyword.

The memory requirement of an algorithm is actually two separate but related things:-

- The memory taken up by the compiled executable code (the object code or binary file) itself (on disk or equivalent, depending on the hardware and language). This can often be reduced by preferring run-time decision making mechanisms (such as virtual functions and run-time type information) over certain compile-time decision making mechanisms (such as macro substitution and templates). This, however, comes at the cost of speed.
- Amount of temporary "dynamic memory" allocated during processing. For example, dynamically pre-caching results, as mentioned earlier, improves speed at the cost of this attribute. Even the depth of sub-routine calls can impact heavily on this cost and increase path length too, especially if there are 'heavy' dynamic memory requirements for the particular functions invoked. The use of copied function parameters (rather than simply using pointers to earlier, already defined, and sometimes static values) actually **doubles** the memory requirement for this particular memory metric (as well as carrying its own processing overhead for the

copying itself. This can be particularly relevant for quite 'lengthy' parameters such as html script, JavaScript source programs or extensive freeform text such as letters or emails.

## **Rematerialization**

It has been argued that Rematerialization (re-calculating) may occasionally be more efficient than holding results in cache. This is the somewhat non-intuitive belief that it can be faster to re-calculate from the input - even if the answer is already known - when it can be shown, in some special cases, to decrease "register pressure". Some optimizing compilers have the ability to decide when this is considered worthwhile based on a number of criteria such as complexity and no side effects, and works by keeping track of the expression used to compute each variable, using the concept of available expressions.

This is most likely to be true when a calculation is very fast (such as addition or bitwise operations), while the amount of data which must be cached would be very large, resulting in inefficient storage. Small amounts of data can be stored very efficiently in registers or fast cache, while in most contemporary computers large amounts of data must be stored in slower memory or even slower hard drive storage, and thus the efficiency of storing data which can be computed quickly drops significantly.

## **Precomputation**

Precomputing a complete range of results prior to compiling, or at the beginning of an algorithm's execution, can often increase algorithmic efficiency substantially. This becomes advantageous when one or more inputs is constrained to a small enough range that the results can be stored in a reasonably sized block of memory. Because memory access is essentially constant in time complexity (except for caching delays), any algorithm with a component which has worse than constant efficiency over a small input range can be improved by precomputing values. In some cases efficient approximation algorithms can be obtained by computing a discrete subset of values and interpolating for intermediate input values, since interpolation is also a linear operation.

## ***Transmission size***

Data compression algorithms can be useful because they help reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. This however also comes at a cost - which is additional processing time to compress and subsequently decompress. Depending upon the speed of the data transfer, compression may reduce overall response times which, ultimately, equates to speed - even though processing within the computer itself takes longer. For audio, MP3 is a compression method used extensively in portable sound systems. The efficiency of a data compression algorithm relates to the compression factor and speed of achieving both compression and decompression. For the purpose of archiving an extensive database, it might be considered worthwhile to achieve a very high compression ratio, since decompression is less likely to occur on the majority of the data.

## **Data presentation**

Output data can be presented to the end user in many ways - such as via punched tape or card, digital displays, local display monitors, remote computer monitors or printed. Each of these has its own inherent initial cost and, in some cases, an ongoing cost (e.g. refreshing an image from memory). As an example, the web site "Google" recently showed, as its logo, an image of the Vancouver olympics that is around 8K of gif image. The normally displayed Google image is a PNG image of 28K (or 48K), yet the raw text string for "Google" occupies only 6 octets or 48 bits (4,778 or 8192 *times* less). This graphically illustrates that how data is presented can significantly effect the overall efficiency of transmission (and also the complete algorithm - since both GIF and PNG images require yet more processing).

It is estimated by "Internet World Stats" that there were 1,733,993,741 internet users in 2009 and, to transmit this new image to each one of them, would require around 136,000 *billion* ( $10^9$ ) octets of data to be transmitted - at least once - into their personal web cache. In "Computational Energy Cost of TCP", co-authors Bokyoung Wang and Suresh Singh examine the energy costs for TCP and calculated, for their chosen example, a cost of 0.022 Joules per packet (of approx 1489 octets). On this basis, a total of around 2,000,000,000 joules (2 GJ) of energy might be expended by TCP elements alone to display the new logo for all users for the first time. To maintain or re-display this image requires still more processing and consequential energy cost (in contrast to printed output for instance).

## **Encoding and decoding methods (compared and contrasted)**

When data is encoded for any 'external' use, it is possible to do so in an almost unlimited variety of different formats that are sometimes conflicting. This content encoding (of the raw data) may be designed for:

- optimal readability – by humans
- optimal decoding speed – by other computer programs
- optimal compression – for archiving or data transmission
- optimal compatibility – with "legacy" or other existing formats or programming languages
- optimal security – using encryption

It is unlikely that all of these goals could be met with a single 'generic' encoding scheme and so a compromise will often be the desired goal and will often be compromised by the need for standardization and/or legacy and compatibility issues.

## **Encoding efficiently**

For data encoding whose destination is to be input for further computer processing, readability is not an issue – since the receiving processors algorithm can decode the data to any other desirable form including human readable. From the perspective of

algorithmic efficiency, **minimizing subsequent decoding** (with zero or minimal parsing) should take highest priority. The general rule of thumb is that any encoding system that 'understands' the underlying data structure - sufficiently to encode it in the first place - should be equally capable of easily encoding it in such a way that makes decoding it highly efficient. For variable length data with possibly omitted data values, for instance, this almost certainly means the utilization of declarative notation (i.e. including the length of the data item as a prefix to the data so that a de-limiter is not required and parsing completely eliminated). For keyword data items, tokenizing the key to an index (integer) after its first occurrence not only reduces subsequent data size but, furthermore, reduces future decoding overhead for the same items that are repeated.

Historically, optimal encoding was not only worthwhile from an efficiency standpoint but was also common practise to conserve valuable memory, external storage and processor resources. Once validated a country name for example could be held as a shorter sequential country code which could then also act as an index for subsequent 'decoding', using this code as an entry number within a table or record number within a file. If the table or file contained fixed length entries, the code could easily be converted to an absolute memory address or disk address for fast retrieval. The ISBN system for identifying books is a good example of a practical encoding method which also contains a built-in hierarchy.

According to recent articles in New Scientist and Scientific American; "TODAY'S telecommunications networks could use one ten-thousandth of the power they presently consume if smarter data-coding techniques were used", according to Bell Labs, based in Murray Hill, New Jersey It recognizes that this is only a theoretical limit but nevertheless sets itself a more realistic, practical goal of a 1,000 fold reduction within 5 years with future, as yet unidentified, technological changes.

### **Examples of several common encoding methods**

- Comma separated values (CSV - a list of data values separated by commas)
- Tab separated values (TSV) - a list of data values separated by 'tab' characters
- HyperText Markup Language (HTML) - the predominant markup language for web pages
- Extensible Markup Language (XML) - a generic framework for storing any amount of text or any data whose structure can be represented as a tree with at least one element - the root element.
- JavaScript Object Notation (JSON) - human-readable format for representing simple data structures

The last of these, (JSON) is apparently widely used for internet data transmission, primarily it seems because the data can be uploaded by a single JavaScript 'eval' statement - without the need to produce what otherwise would likely have been a more efficient purpose built encoder/decoder. There are in fact quite large amounts of repeated (and therefore redundant data) in this particular format, and also in HTML and XML source, that could quite easily be eliminated. XML is recognized as a verbose form of

encoding. Binary XML has been put forward as one method of reducing transfer and processing times for XML and, while there are several competing formats, none has been widely adopted by a standards organization or accepted as a de facto standard. It has also been criticized by Jimmy Zhang for essentially trying to solve the wrong problem

There are a number of freely available products on the market that partially compress HTML files and perform some or all of the following:

- merge lines
- remove unnecessary whitespace characters
- remove unnecessary quotation marks. For example, BORDER="0" will be replaced with BORDER=0)
- replace some tags with shorter ones (e.g. replace STRIKE tags with S, STRONG with B and EM with I)
- remove HTML comments (comments within scripts and styles are not removed)
- remove <!DOCTYPE.> tags
- remove named meta tags

The effect of this limited form of compression is to make the HTML code smaller and faster to load, but more difficult to read manually (so the original HTML code is usually retained for updating), but since it is predominantly meant to be processed only by a browser, this causes no problems. Despite these small improvements, HTML, which is the predominant language for the web still remains a predominantly *source* distributed, interpreted markup language, with high redundancy.

## **Kolmogorov complexity**

The study of encoding techniques has been examined in depth in an area of computer science characterized by a method known as Kolmogorov complexity where a value known as (**'K'**) is accepted as 'not a computable function'. The Kolmogorov complexity of any computable object is the length of the shortest program that computes it and then halts. The invariance theorem shows that it is not really important which computer is used. Essentially this implies that there is no automated method that can produce an optimum result and is therefore characterized by a requirement for human ingenuity or Innovation.

## ***Effect of programming paradigms***

The effect that different programming paradigms have on algorithmic efficiency is fiercely contested, with both supporters and antagonists for each new paradigm. Strong supporters of structured programming, such as Dijkstra for instance, who favour entirely goto-less programs are met with conflicting evidence that appears to nullify its supposed benefits. The truth is, even if the structured code itself contains no gotos, the optimizing compiler that creates the binary code almost certainly generates them (and not necessarily in the most efficient way). Similarly, OOP protagonists who claim their paradigm is superior are met with opposition from strong sceptics such as Alexander Stepanov who

suggested that OOP provides a mathematically limited viewpoint and called it, "almost as much of a hoax as Artificial Intelligence" In the long term, benchmarks, using real-life examples, provide the only real hope of resolving such conflicts - at least in terms of run-time efficiency.

## ***Optimization techniques***

The word optimize is normally used in relation to an existing algorithm/computer program (i.e. to improve upon completed code). Here, it is used both in the context of existing programs and also in the design and implementation of new algorithms, thereby avoiding the most common performance pitfalls. It is clearly wasteful to produce a working program - at first using an algorithm that ignores all efficiency issues - only to then have to redesign or rewrite sections of it if found to offer poor performance. Optimization can be broadly categorized into two domains:-

- Environment specific - that are essentially worthwhile only on certain platforms or particular computer languages
- General techniques - that apply irrespective of platform

### **Environment specific**

Optimization of algorithms frequently depends on the properties of the machine the algorithm will be executed on as well as the language the algorithm is written in and chosen data types. For example, a programmer might optimize code for time efficiency in an application for home computers (with sizable amounts of memory), but for code destined to be embedded in small, "memory-tight" devices, the programmer may have to accept that it will run more slowly, simply because of the restricted memory available for any potential software optimization.

### **General techniques**

- Linear search such as unsorted table look-ups in particular can be very expensive in terms of execution time but can be reduced significantly through use of efficient techniques such as indexed arrays and binary searches. Using a simple linear search on first occurrence and using a cached result thereafter is an obvious compromise.
- Use of indexed program branching, utilizing branch tables or "threaded code" to control program flow, (rather than using multiple conditional IF statements or unoptimized CASE/SWITCH) can drastically reduce instruction path length, simultaneously reduce program size and even also make a program easier to read and more easily maintainable (in effect it becomes a 'decision table' rather than repetitive spaghetti code).
- Loop unrolling performed manually, or more usually by an optimizing compiler, can provide significant savings in some instances. By processing 'blocks' of several array elements at a time, individually addressed, (for example, within a `while` loop), much pointer arithmetic and end of loop testing can be eliminated,

resulting in decreased instruction path lengths. Other Loop optimizations are also possible.

## Tunnel vision

There are many techniques for improving algorithms, but focusing on a single favorite technique can lead to a "tunnel vision" mentality. For example, in this X86 assembly example, the author offers loop unrolling as a reasonable technique that provides some 40% improvements to his chosen example. However, the same example would benefit significantly from both inlining and use of a trivial hash function. If they were implemented, either as alternative or complementary techniques, an even greater percentage gain might be expected. A combination of optimizations may provide ever increasing speed, but selection of the most easily implemented and most effective technique, from a large repertoire of such techniques, is desirable as a starting point.

## Dependency trees and spreadsheets

Spreadsheets are a 'special case' of algorithms that self-optimize by virtue of their dependency trees that are inherent in the design of spreadsheets to reduce re-calculations when a cell changes. The results of earlier calculations are effectively cached within the workbook and only updated if another cells changed value effects it directly.

## Table lookup

Table lookups can make many algorithms more efficient, particularly when used to bypass computations with a high time complexity. However, if a wide input range is required, they can consume significant storage resources. In cases with a sparse valid input set, hash functions can be used to provide more efficient lookup access than a full table.

## Hash function algorithms

A **hash function** is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**.

Hash functions are frequently used to speed up table lookups. The choice of a hashing function (to avoid a linear or brute force search) depends critically on the nature of the input data, and their probability distribution in the intended application.

## Trivial hash function

Sometimes if the datum is small enough, a "trivial hash function" can be used to effectively provide constant time searches at almost zero cost. This is particularly relevant for single byte lookups (e.g. ASCII or EBCDIC characters)

## **Searching strings**

Searching for particular text strings (for instance "tags" or keywords) in long sequences of characters potentially generates lengthy instruction paths. This includes searching for delimiters in comma separated files or similar processing which can be very simply and effectively eliminated (using declarative notation for instance).

Several methods of reducing the cost for general searching have been examined and the "Boyer–Moore string search algorithm" (or Boyer–Moore–Horspool algorithm, a similar but modified version) is one solution that has been proven to give superior results to repetitive comparisons of the entire search string along the sequence.

## **Hot spot analyzers**

Special system software products known as "performance analyzers" are often available from suppliers to help diagnose "hot spots" - during actual execution of computer programs - using real or test data - they perform a Performance analysis under generally repeatable conditions. They can pinpoint sections of the program that might benefit from specifically targeted programmer optimization without necessarily spending time optimizing the rest of the code. Using program re-runs, a measure of relative improvement can then be determined to decide if the optimization was successful and by what amount. Instruction Set Simulators can be used as an alternative to measure the instruction path length at the machine code level between selected execution paths, or on the entire execution.

Regardless of the type of tool used, the quantitative values obtained can be used in combination with anticipated reductions (for the targeted code) to estimate a relative or absolute overall saving. For example if 50% of the total execution time (or path length) is absorbed in a subroutine whose speed can be doubled by programmer optimization, an overall saving of around 25% might be expected (Amdahl law).

Efforts have been made at the University of California, Irvine to produce dynamic executable code using a combination of hot spot analysis and run-time program trace tree. A JIT like dynamic compiler was built by Andreas Gal and others, "in which relevant (i.e., frequently executed) control flows are ...discovered lazily during execution"

## **Benchmarking & competitive algorithms**

For new versions of software or to provide comparisons with competitive systems, benchmarks are sometimes used which assist with gauging an algorithms relative performance. If a new sort algorithm is produced for example it can be compared with its predecessors to ensure that at least it is efficient as before with known data - taking into consideration any functional improvements. Benchmarks can be used by customers when comparing various products from alternative suppliers to estimate which product will best suit their specific requirements in terms of functionality and performance. For example in the mainframe world certain proprietary sort products from independent software

companies such as Syncsort compete with products from the major suppliers such as IBM for speed. Some benchmarks provide opportunities for producing an analysis comparing the relative speed of various compiled and interpreted languages for example and *The Computer Language Benchmarks Game* compares the performance of implementations of typical programming problems in several programming languages. (Even creating "do it yourself" benchmarks to get at least some appreciation of the relative performance of different programming languages, using a variety of user specified criteria, is quite simple to produce as this "Nine language Performance roundup" by Christopher W. Cowell-Shah demonstrates by example)

## **Compiled versus interpreted languages**

A compiled algorithm will, in general, execute faster than the equivalent interpreted algorithm simply because some processing is required even at run time to 'understand' (i.e. interpret) the instructions to effect an execution. A compiled program will normally output an object or machine code equivalent of the algorithm that has already been processed by the compiler into a form more readily executed by microcode or the hardware directly. The popular Perl language is an example of an interpreted language and benchmarks indicate that it executes approximately 24 times more slowly than compiled C.

## **Optimizing compilers**

Many compilers have features that attempt to optimize the code they generate, utilizing some of the techniques outlined here and others specific to the compilation itself. Loop optimization is often the focus of optimizing compilers because significant time is spent in program loops and parallel processing opportunities can often be facilitated by automatic code re-structuring such as loop unrolling. Optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all program source code, the fastest (or smallest) possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem. Additionally, even optimizing compilers generally have no access to runtime metrics to enable them to improve optimization through 'learning'.

## **Just-in-time compilers**

'On-the-fly' processors known today as just-in-time or 'JIT' compilers combine features of interpreted languages with compiled languages and may also incorporate elements of optimization to a greater or lesser extent. Essentially the JIT compiler can compile small sections of source code statements (or bytecode) as they are newly encountered and (usually) retain the result for the next time the same source is processed. In addition, pre-compiled segments of code can be in-lined or called as dynamic functions that themselves perform equally fast as the equivalent 'custom' compiled function. Because the JIT processor also has access to run-time information (that a normal compiler can't have) it is also possible for it to optimize further executions depending upon the input and also perform other run-time introspective optimization as execution proceeds. A JIT

processor may, or may not, incorporate self modifying code or its equivalent by creating 'fast path' routes through an algorithm. It may also use such techniques as dynamic Fractional cascading or any other similar runtime device based on collected actual runtime metrics. It is therefore entirely possible that a JIT compiler might (counter intuitively) execute even faster than an optimally 'optimized' compiled program.

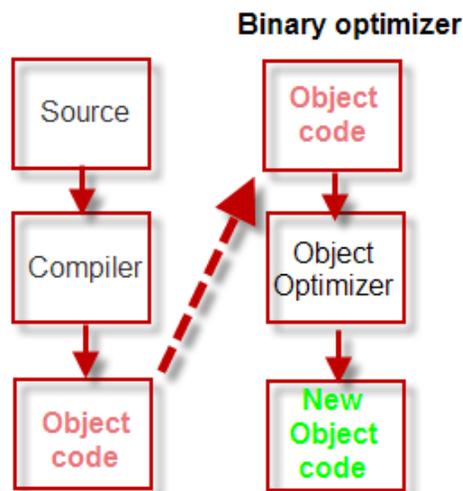
### Self-modifying code

As mentioned above, just-in-time compilers often make extensive use of self-modifying code to generate the actual machine instructions required to be executed. The technique can also be used to reduce instruction path lengths in application programs where otherwise repetitive conditional tests might otherwise be required within the main program flow. This can be particularly useful where a sub routine may have embedded debugging code that is either active (testing mode) or inactive (production mode) depending upon some input parameter. A simple solution using a form of dynamic dispatching is where the sub routine entry point is dynamically 'swapped' at initialization, depending upon the input parameter. Entry point A) includes the debugging code prologue and entry point B) excludes the prologue; thus eliminating all overhead except the initial 'test and swap' (even when test/debugging is selected, when the overhead is simply the test/debugging code itself).

### Genetic algorithm

In the world of performance related algorithms it is worth mentioning the role of genetic algorithms which compete using similar methods to the natural world in eliminating inferior algorithms in favour of more efficient versions.

### Object code optimizers



A binary optimizer takes the existing output from a compiler and produces a better execution file with the same functionality.

Some proprietary program optimizers such as the "COBOL Optimizer" developed by Capex Corporation in the mid 1970's for COBOL, actually took the unusual step of optimizing the Object code (or binary file) **after** normal compilation. This type of optimizer, recently sometimes referred to as a "post pass" optimizer or peephole optimizer, depended, in this case, upon knowledge of 'weaknesses' in the standard IBM COBOL compiler and actually replaced (or patched) sections of the object code with more efficient code.

A number of other suppliers have recently adopted the same approach.

### **Alignment of data**

Most processors execute faster if certain data values are aligned on word, doubleword or page boundaries. If possible design/specify structures to satisfy appropriate alignments. This avoids exceptions.

### **Locality of reference**

To reduce Cache miss exceptions by providing good spatial locality of reference, specify 'high frequency'/volatile working storage data within defined structure(s) so that they are also allocated from contiguous sections of memory (rather than possibly scattered over many pages). Group closely related data values also 'physically' close together within these structures. Consider the possibility of creating an 'artificial' structure to group some otherwise unrelated, but nevertheless frequently referenced, items together.

### **Choice of instruction or data type**

Particularly in an Assembly language (although also applicable to HLL statements), the choice of a particular 'instruction' or data type, can have a large impact on execution efficiency. In general, instructions that process variables such as signed or unsigned 16-bit or 32-bit integers are faster than those that process floating point or packed decimal. Modern processors are even capable of executing multiple 'fixed point' instructions in parallel with the simultaneous execution of a floating point instruction. If the largest integer to be encountered can be accommodated by the 'faster' data type, defining the variables as that type will result in faster execution - since even a non-optimizing compiler will, in-effect, be 'forced' to choose appropriate instructions that will execute faster than would have been the case with data types associated with 'slower' instructions. Assembler programmers (and optimizing compiler writers) can then also benefit from the ability to perform certain common types of arithmetic for instance - division by 2, 4, 8 etc. by performing the very much faster binary shift right operations (in this case by 1, 2 or 3 bits).

If the choice of input data type is not under the control of the programmer, although prior conversion (outside of a loop for instance) to a faster data type carries some overhead, it can often be worthwhile if the variable is then to be used as a loop counter, especially if the count could be quite a high value or there are many input values to process. As

mentioned above, choice of individual assembler instructions (or even sometimes just their order of execution) on particular machines can effect the efficiency of an algorithm.

## **Data granularity**

The greater the granularity of data definitions (such as splitting a geographic address into separate street/city/postal code fields) can have performance overhead implications during processing. Higher granularity in this example implies more procedure calls in Object-oriented programming and parallel computing environments since the additional objects are accessed via multiple method calls rather than perhaps one.

## **Subroutine granularity**

For structured programming and procedural programming in general, great emphasis is placed on designing programs as a hierarchy of (or at least a set of) subroutines. For object oriented programming, the method call (a subroutine call) is the standard method of testing and setting all values in objects and so increasing data granularity consequently causes increased use of subroutines. The greater the granularity of subroutine usage, the larger the proportion of processing time devoted to the mechanism of the subroutine linkages themselves. The presence of a (called) subroutine in a program contributes nothing extra to the functionality of the program. The extent to which subroutines (and their consequent memory requirements) influences the overall performance of the complete algorithm depends to a large extent on the actual implementation.

In assembly language programs, the invocation of a subroutine need not involve much overhead, typically adding just a couple of machine instructions to the normal instruction path length, each one altering the control flow either *to* the subroutine or returning *from* it (saving the state on a stack being optional, depending on the complexity of the subroutine and its requirement to reuse general purpose registers). In many cases, small subroutines that perform frequently used data transformations using 'general purpose' work areas can be accomplished without the need to save or restore any registers, including the return register.

By contrast, HLL programs typically always invoke a 'standard' procedure call (the *calling convention*), which involves saving the program state by default and usually allocating additional memory on the stack to save all registers and other relevant state data (the prologue and epilogue code). Recursion in a HLL program can consequently consume significant overhead in both memory and execution time managing the stack to the required depth.

Guy Steele pointed out in a 1977 paper that a *well-designed* programming language implementation *can* have very low overheads for procedural abstraction (but laments, in most implementations, that they seldom achieve this in practice - being "rather thoughtless or careless in this regard"). Steele concludes that "we should have a healthy respect for procedure calls" (because they are powerful) but he also cautioned "use them sparingly"

## Choice of language / mixed languages

Some computer languages can execute algorithms more efficiently than others. As stated already, interpreted languages often perform less efficiently than compiled languages. In addition, where possible, 'high-use', and time-dependent sections of code may be written in a language such as assembler that can usually execute faster and make better use of resources on a particular platform than the equivalent HLL code on the same platform. This section of code can either be statically called or dynamically invoked (external function) or embedded within the higher level code (e.g. Assembler instructions embedded in a 'C' language program). The effect of higher levels of abstraction when using a HLL has been described as the *Abstraction penalty*. Programmers who are familiar with assembler language (in addition to their chosen HLL) and are also familiar with the code that will be generated by the HLL, under known conditions, can sometimes use HLL language primitives of that language to generate code almost identical to assembler language. This is most likely to be possible only in languages that support pointers such as PL/1 or C. This is facilitated if the chosen HLL compiler provides an optional assembler listing as part of its printout so that various alternatives can be explored without also needing specialist knowledge of the compiler internals.

## Software validation versus hardware validation

An optimization technique that was frequently taken advantage of on 'legacy' platforms was that of allowing the hardware (or microcode) to perform validation on numeric data fields such as those coded in (or converted to) packed decimal (or packed BCD). The choice was to either spend processing time checking each field for a valid numeric content in the particular internal representation chosen or simply assume the data was correct and let the hardware detect the error upon execution. The choice was highly significant because to check for validity on multiple fields (for sometimes many millions of input records), it could occupy valuable computer resources. Since input data fields were in any case frequently built from the output of earlier computer processing, the actual probability of a field containing invalid data was exceedingly low and usually the result of some 'corruption'. The solution was to incorporate an 'event handler' for the hardware detected condition ('data exception') that would intercept the occasional errant data field and either 'report, correct and continue' or, more usually, abort the run with a core dump to try to determine the reason for the bad data.

Similar event handlers are frequently utilized in today's web based applications to handle other exceptional conditions but repeatedly parsing data input, to ensure its validity before execution, has nevertheless become much more commonplace - partly because processors have become faster (and the perceived need for efficiency in this area less significant) but, predominantly - because data structures have become less 'formalized' (e.g. .csv and .tsv files) or uniquely identifiable (e.g. packed decimal). The potential savings using this type of technique may have therefore fallen into general dis-use as a consequence and therefore repeated data validations and repeated data conversions have become an accepted overhead. Ironically, one consequence of this move to less formalized data structures is that a corruption of say, a numeric binary integer value, will

not be detected at all by the hardware upon execution (for instance: is an ASCII hexadecimal value '20202020' a valid signed or unsigned binary value - or simply a string of blanks that has corrupted it?)

## **Algorithms for vector & superscalar processors**

Algorithms for vector processors are usually different than those for scalar processors since they can process multiple instructions and/or multiple data elements in parallel. The process of converting an algorithm from a scalar to a vector process is known as vectorization and methods for automatically performing this transformation as efficiently as possible are constantly sought. There are intrinsic limitations for implementing instruction level parallelism in Superscalar processors but, in essence, the overhead in deciding for certain if particular instruction sequences can be processed in parallel can sometimes exceed the efficiency gain in so doing. The achievable reduction is governed primarily by the (somewhat obvious) law known as Amdahl's law, that essentially states that the improvement from parallel processing is determined by its slowest sequential component. Algorithms designed for this class of processor therefore require more care if they are not to unwittingly disrupt the potential gains.

## **Avoiding costs**

- Defining variables as integers for indexed arrays instead of floating point will result in faster execution (see above).
- Defining structures whose structure length is a multiple of a power of 2 (2,4,8,16 etc.), will allow the compiler to calculate array indexes by shifting a binary index by 1, 2 or more bits to the left, instead of using a multiply instruction will result in faster execution. Adding an otherwise redundant short filler variable to 'pad out' the length of a structure element to say 8 bytes when otherwise it would have been 6 or 7 bytes may reduce overall processing time by a worthwhile amount for very large arrays.
- Storage defined in terms of bits, when bytes would suffice, may inadvertently involve extremely long path lengths involving bitwise operations instead of more efficient single instruction 'multiple byte' copy instructions. (This does not apply to 'genuine' intentional bitwise operations - used for example instead of multiplication or division by powers of 2 or for TRUE/FALSE flags.)
- Unnecessary use of allocated dynamic storage when static storage would suffice, can increase the processing overhead substantially - both increasing memory requirements and the associated allocation/deallocation path length overheads for each function call.
- Excessive use of function calls for very simple functions, rather than in-line statements, can also add substantially to instruction path lengths and stack/unstack overheads. For particularly time critical systems that are not also code size sensitive, automatic or manual inline expansion can reduce path length by eliminating all the instructions that call the function and return from it. (A conceptually similar method, loop unrolling, eliminates the instructions required to set up and terminate a loop by, instead; repeating the instructions inside the

loop multiple times. This of course eliminates the branch back instruction but may also increase the size of the binary file or, in the case of JIT built code, dynamic memory. Also, care must be taken with this method, that re-calculating addresses for each statement within an unwound indexed loop is not more expensive than incrementing pointers within the former loop would have been. If absolute indexes are used in the generated (or manually created) unwound code, rather than variables, the code created may actually be able to avoid generated pointer arithmetic instructions altogether, using offsets instead).

### **Memory management**

Whenever memory is *automatically* allocated (for example in HLL programs, when calling a procedure or when issuing a system call), it is normally released (or 'freed'/ 'deallocated'/ 'deleted' ) automatically when it is no longer required - thus allowing it to be re-used for another purpose *immediately*. Some memory management can easily be accomplished by the compiler, as in this example. However, when memory is *explicitly* allocated (for example in OOP when "new" is specified for an object), releasing the memory is often left to an asynchronous 'garbage collector' which does not necessarily release the memory at the earliest opportunity (as well as consuming some additional CPU resources deciding if it can be). The current trend nevertheless appears to be towards taking full advantage of this fully automated method, despite the tradeoff in efficiency - because it is claimed that it makes programming easier. Some functional languages are known as 'lazy functional languages' because of the significant use of garbage collection and can consume much more memory as a result.

- Array processing may simplify programming but use of separate statements to sum different elements of the same array(s) may produce code that is not easily optimized and that requires multiple passes of the arrays that might otherwise have been processed in a single pass. It may also duplicate data if array slicing is used, leading to increased memory usage and copying overhead.
- In OOP, if an object is known to be immutable, it can be copied simply by making a copy of a reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a boost in execution speed.

### **Readability, trade offs and trends**

One must be careful, in the pursuit of good coding style, not to over-emphasize efficiency. Frequently, a clean, readable and 'usable' design is much more important than a fast, efficient design that is hard to understand. There are exceptions to this 'rule' (such as embedded systems, where space is tight, and processing power minimal) but these are rarer than one might expect.

However, increasingly, for many 'time critical' applications such as air line reservation systems, point-of-sale applications, ATMs (cash-point machines), Airline Guidance systems, Collision avoidance systems and numerous modern web based applications -

operating in a real-time environment where speed of response is fundamental - there is little alternative.

## **Determining if optimization is worthwhile**

The essential criteria for using optimized code are of course dependent upon the expected use of the algorithm. If it is a new algorithm and is going to be in use for many years and speed is relevant, it is worth spending some time designing the code to be as efficient as possible from the outset. If an existing algorithm is proving to be too slow or memory is becoming an issue, clearly something must be done to improve it.

For the average application, or for one-off applications, avoiding inefficient coding techniques and encouraging the compiler to optimize where possible may be sufficient.

One simple way (at least for mathematicians) to determine whether an optimization is worthwhile is as follows: Let the original time and space requirements (generally in Big-O notation) of the algorithm be  $O_1$  and  $O_2$ . Let the new code require  $N_1$  and  $N_2$  time and space respectively. If  $N_1N_2 < O_1O_2$ , the optimization should be carried out. However, as mentioned above, this may not always be true.

## **Implications for algorithmic efficiency**

A recent report, published in December 2007, from Global Action Plan, a UK-based environmental organisation found that computer servers are "at least as great a threat to the climate as SUVs or the global aviation industry" drawing attention to the carbon footprint of the IT industry in the UK. According to an Environmental Research Letters report published in September 2008, "Total power used by information technology equipment in data centers represented about 0.5% of world electricity consumption in 2005. When cooling and auxiliary infrastructure are included, that figure is about 1%. The total data center power demand in 2005 is equivalent (in capacity terms) to about seventeen 1000 MW power plants for the world."

Some media reports claim that performing two Google searches from a desktop computer can generate about the same amount of carbon dioxide as boiling a kettle for a cup of tea, according to new research; however, the factual accuracy of this comparison is disputed, and the author of the study in question asserts that the two-searches-tea-kettle statistic is a misreading of his work.

Greentouch, a recently established consortium of leading Information and Communications Technology (ICT) industry, academic and non-governmental research experts, has set itself the mission of reducing energy consumption per user by a factor of 1000 from current levels. "A thousand-fold reduction is roughly equivalent to being able to power the world's communications networks, including the Internet, for three years using the same amount of energy that it currently takes to run them for a single day". The first meeting in February 2010 will establish the organization's five-year plan, first year deliverables and member roles and responsibilities. Intellectual property

issues will be addressed and defined in the forum's initial planning meetings. The conditions for research and the results of that research will be high priority for discussion in the initial phase of the research forum's development.

Computers having become increasingly more powerful over the past few decades, emphasis was on a 'brute force' mentality. This may have to be reconsidered in the light of these reports and more effort placed in future on reducing carbon footprints through optimization. It is a timely reminder that algorithmic efficiency is just another aspect of the more general thermodynamic efficiency. The genuine economic benefits of an optimized algorithm are, in any case, that more processing can be done for the same cost or that useful results can be shown in a more timely manner and ultimately, acted upon sooner.

### ***Criticism of the current state of programming***

- David May FRS a British computer scientist and currently Professor of Computer Science at University of Bristol and founder and CTO of XMOS Semiconductor, believes one of the problems is that there is a reliance on Moore's law to solve inefficiencies. He has advanced an 'alternative' to Moore's law (May's law) stated as follows:

Software efficiency halves every 18 months, compensating Moore's Law

He goes on to state

In ubiquitous systems, halving the instructions executed can double the battery life and big data sets bring big opportunities for better software and algorithms: Reducing the number of operations from  $N \times N$  to  $N \times \log(N)$  has a dramatic effect when  $N$  is large... for  $N = 30$  billion, this change is as good as 50 years of technology improvements

- Software author Adam N. Rosenburg in his blog "*The failure of the Digital computer*", has described the current state of programming as nearing the "Software event horizon", (alluding to the fictitious "*shoe event horizon*" described by Douglas Adams in his *Hitchhiker's Guide to the Galaxy* book). He estimates there has been a 70 dB factor loss of productivity or "99.99999 percent, of its ability to deliver the goods", since the 1980s - "When Arthur C. Clarke compared the reality of computing in 2001 to the computer HAL in his book 2001: A Space Odyssey, he pointed out how wonderfully small and powerful computers were but how disappointing computer programming had become".
- Conrad Weisert gives examples, some of which were published in ACM SIGPLAN (Special Interest Group on Programming Languages) Notices, December, 1995 in: "Atrocious Programming Thrives"

## Chapter-11

# Quality Assurance

**Quality assurance**, or **QA** for short, is the systematic monitoring and evaluation of the various aspects of a project, service or facility to maximize the probability that minimum standards of quality are being attained by the production process. QA cannot absolutely guarantee the production of *quality* products.

Two principles included in QA are: "Fit for purpose" - the product should be suitable for the intended purpose; and "Right first time" - mistakes should be eliminated. QA includes regulation of the quality of raw materials, assemblies, products and components, services related to production, and management, production and inspection processes.

*Quality* is determined by the product users, clients or customers, not by society in general. It is not the same as 'expensive' or 'high quality'. Low priced products can be considered as having high quality if the product users determine them as such.

### **Early efforts to control the quality of production**

During the Middle Ages, guilds adopted responsibility for quality control of their members, setting and maintaining certain standards for guild membership.

Royal governments purchasing material were interested in quality control as customers. For this reason, King John of England appointed William Wrotham to report about the construction and repair of ships. Centuries later, Samuel Pepys, Secretary to the British Admiralty, appointed multiple such overseers.

Prior to the extensive division of labor and mechanization resulting from the Industrial Revolution, it was possible for workers to control the quality of their own products. The Industrial Revolution led to a system in which large groups of people performing a similar type of work were grouped together under the supervision of a foreman who was appointed to control the quality of work manufactured.

## **Wartime production**

At the time of the First World War, manufacturing processes typically became more complex with larger numbers of workers being supervised. This period saw the widespread introduction of mass production and piecework, which created problems as workmen could now earn more money by the production of extra products, which in turn occasionally led to poor quality workmanship being passed on to the assembly lines. To counter bad workmanship, full time inspectors were introduced into the to identify, quarantine and ideally correct product quality failures. Quality control by inspection in the 1920s and 1930s led to the growth of quality inspection functions, separately organised from production and large enough to be headed by superintendents.

The systematic approach to quality started in industrial manufacture during the 1930s, mostly in the USA, when some attention was given to the cost of scrap and rework. With the impact of mass production required during the Second World War made it necessary to introduce an improved form of quality control known as Statistical Quality Control, or SQC. Some of the initial work for SQC is credited to Walter A. Shewhart of Bell Labs, starting with his famous one-page memorandum of 1924.

SQC includes the concept that every production piece cannot be fully inspected into acceptable and nonacceptable batches. By extending the inspection phase and making inspection organizations more efficient, it provides inspectors with control tools such as sampling and control charts, even where 100 per cent inspection is not practicable. Standard statistical techniques allow the producer to sample and test a certain proportion of the products for quality to achieve the desired level of confidence in the quality of the entire batch or production run.

## **Postwar**

In the period following World War II, many countries' manufacturing capabilities that had been destroyed during the war were rebuilt. General Douglas MacArthur oversaw the re-building of Japan. During this time, General MacArthur involved two key individuals in the development of modern quality concepts: W. Edwards Deming and Joseph Juran. Both individuals promoted the collaborative concepts of quality to Japanese business and technical groups, and these groups utilized these concepts in the redevelopment of the Japanese economy.

Although there were many individuals trying to lead United States industries towards a more comprehensive approach to quality, the U.S. continued to apply the Quality Control (QC) concepts of inspection and sampling to remove defective product from production lines, essentially ignoring advances in QA for decades.

## **Steps for a typical quality assurance process**

There are many forms of QA processes, of varying scope and depth. The application of a particular process is often customized to the production process.

A typical process may include:

- test of previous articles
- plan to improve
- design to include improvements and requirements
- manufacture with improvements
- review new item and improvements
- test of the new item

## **Failure testing**

Valuable processes to perform on a whole consumer product is failure testing or stress testing. In mechanical terms this is the operation of a product until it fails, often under stresses such as increasing vibration, temperature, and humidity. This exposes many unanticipated weaknesses in a product, and the data are used to drive engineering and manufacturing process improvements. Often quite simple changes can dramatically improve product service, such as changing to mold-resistant paint or adding lock-washer placement to the training for new assembly personnel.

## **Statistical control**

Many organizations use statistical process control to bring the organization to Six Sigma levels of quality, in other words, so that the likelihood of an unexpected failure is confined to six standard deviations on the normal distribution. This probability is less than four one-millionths. Items controlled often include clerical tasks such as order-entry as well as conventional manufacturing tasks.

Traditional statistical process controls in manufacturing operations usually proceed by randomly sampling and testing a fraction of the output. Variances in critical tolerances are continuously tracked and where necessary corrected before bad parts are produced.

## **Total quality management**

The quality of products is dependent upon that of the participating constituents, some of which are sustainable and effectively controlled while others are not. The process(es) which are managed with QA pertain to Total Quality Management.

If the specification does not reflect the true quality requirements, the product's quality cannot be guaranteed. For instance, the parameters for a pressure vessel should cover not only the material and dimensions but operating, environmental, safety, reliability and maintainability requirements.

## ***QA in software development***

The following are examples of QA models relating to the software development process.

## **Models and standards**

ISO 17025 is an international standard that specifies the general requirements for the competence to carry out tests and or calibrations. There are 15 management requirements and 10 technical requirements. These requirements outline what a laboratory must do to become accredited. Management system refers to the organization's structure for managing its processes or activities that transform inputs of resources into a product or service which meets the organization's objectives, such as satisfying the customer's quality requirements, complying with regulations, or meeting environmental objectives.

The CMMI (Capability Maturity Model Integration) model is widely used to implement Quality Assurance (PPQA) in an organization. The CMMI maturity levels can be divided in to 5 steps, which a company can achieve by performing specific activities within the organization. (CMMI QA processes are excellent for companies like NASA, and may even be adapted for agile development style).

## ***Company quality***

During the 1980s, the concept of “company quality” with the focus on management and people came to the fore. It was realized that, if all departments approached quality with an open mind, success was possible if the management led the quality improvement process.

The company-wide quality approach places an emphasis on four aspects :-

1. Elements such as controls, job management, adequate processes, performance and integrity criteria and identification of records
2. Competence such as knowledge, skills, experience, qualifications
3. Soft elements, such as personnel integrity, confidence, organizational culture, motivation, team spirit and quality relationships.
4. Infrastructure (as it enhances or limits functionality)

The quality of the outputs is at risk if any of these aspects is deficient.

QA is not limited to the manufacturing, and can be applied to any business or non-business activity:

- Design work
- Administrative services
- Consulting
- Banking
- Insurance
- Computer software development
- Retailing
- Transportation
- Education

- Translation

It comprises a quality improvement process, which is generic in the sense it can be applied to any of these activities and it establishes a behavior pattern, which supports the achievement of quality.

This in turn is supported by quality management practices which can include a number of business systems and which are usually specific to the activities of the business unit concerned.

In manufacturing and construction activities, these business practices can be equated to the models for quality assurance defined by the International Standards contained in the ISO 9000 series and the specified Specifications for quality systems.

In the system of Company Quality, the work being carried out was shop floor inspection which did not reveal the major quality problems. This led to quality assurance or total quality control, which has come into being recently.

### ***Using contractors and/or consultants***

Consultants and contractors are sometimes employed when introducing new quality practices and methods, particularly where the relevant skills and expertise are not available within the organization or when allocating the available internal resources are not available. Consultants and contractors will often employ Quality Management Systems (QMS), auditing and procedural documentation writing CMMI, Six Sigma, Measurement Systems Analysis (MSA), Quality Function Deployment (QFD), Failure Mode and Effects Analysis (FMEA), and Advance Product Quality Planning (APQP).

### ***Quality assurance in European vocational education & training***

With the formulation of a joint quality strategy, the European Union seeks to fostering the overall attractiveness of vocational education & training (VET) in Europe. In order to promote this process, a set of new policy instruments were implemented, such as **CQAF** (Common Quality Assurance Framework) and **EQARF** (European Quality Assurance Reference framework), each of which shall contribute to the establishment of a common quality assurance policy and quality culture in VET throughout Europe. Furthermore the new policy instruments shall allow for an increased transparency and mutual trust between national VET systems.

In line with the European quality strategy, the member states subsequently have implemented national structures (QANRPs: reference points for quality assurance in VET), who closely collaborate with national stakeholders in order to meet the requirements and priorities of the national VET systems and support activities to training providers in order to guarantee the implementation and commitment at all levels. At European level, the cooperation between QANRPs will be ensured through the **EQAVET** network.

Over the past few years, with financial support of the European Union as well as the EU member states, numerous pilot initiatives have been developed, most of which are concerned with the promotion and development of quality in VET throughout Europe. Examples can be found in the project database **ADAM**, which keeps comprehensive information about innovation & transfer projects sponsored by the EU.

A practical example might be seen in the **BEQUAL** project, which has developed a benchmarking tool for training providers, who with the help of the online-tool can benchmark their quality performance in line with the CQAF quality process model. Furthermore the project offers a database with European good practice on quality assurance in the field of vocational education & training.

- Online Benchmarking Tool For Vocational Training Institutes
- European Quality Assurance in Vocational Education & Training

## Chapter-12

# Load Testing

**Load testing** is the process of putting demand on a system or device and measuring its response. Load testing is performed to determine a system's behavior under both normal and anticipated peak load conditions. It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. When the load placed on the system is raised beyond normal usage patterns, in order to test the system's response at unusually high or peak loads, it is known as stress testing. The load is usually so great that error conditions are the expected result, although no clear boundary exists when an activity ceases to be a load test and becomes a stress test.

There is little agreement on what the specific goals of load testing are. The term is often used synonymously with software performance testing, reliability testing, and volume testing. *Load testing* is a type of non-functional testing.

### ***Software load testing***

The term *load testing* is used in different ways in the professional software testing community. *Load testing* generally refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program concurrently. As such, this testing is most relevant for multi-user systems; often one built using a client/server model, such as web servers. However, other types of software systems can also be load tested. For example, a word processor or graphics editor can be forced to read an extremely large document; or a financial package can be forced to generate a report based on several years' worth of data. The most accurate load testing simulates actual use, as opposed to testing using theoretical or analytical modeling.

Load and performance testing analyzes software intended for a multi-user audience by subjecting the software to different amounts of virtual and live users while monitoring performance measurements under these different loads. Load and performance testing is usually conducted in a test environment identical to the production environment before the software system is permitted to go live.

As an example, a web site with shopping cart capability is required to support 100 concurrent users broken out into following activities:

- 25 Virtual Users (VUsers) log in, browse through items and then log off
- 25 VUsers log in, add items to their shopping cart, check out and then log off
- 25 VUsers log in, return items previously purchased and then log off
- 25 VUsers just log in without any subsequent activity

A test analyst can use various load testing tools to create these VUsers and their activities. Once the test has started and reached a steady state, the application is being tested at the 100 VUser load as described above. The application's performance can then be monitored and captured.

The specifics of a load test plan or script will generally vary across organizations. For example, in the bulleted list above, the first item could represent 25 VUsers browsing unique items, random items, or a selected set of items depending upon the test plan or script developed. However, all load test plans attempt to simulate system performance across a range of anticipated peak workflows and volumes. The criteria for passing or failing a load test (pass/fail criteria) are generally different across organizations as well. There are no standards specifying acceptable load testing performance metrics.

A common misconception is that load testing software provides record and playback capabilities like regression testing tools. Load testing tools analyze the entire OSI protocol stack whereas most regression testing tools focus on GUI performance. For example, a regression testing tool will record and playback a mouse click on a button on a web browser, but a load testing tool will send out hypertext the web browser sends after the user clicks the button. In a multiple-user environment, load testing tools can send out hypertext for multiple users with each user having a unique login ID, password, etc.

The popular load testing tools available also provide insight into the causes for slow performance. There are numerous possible causes for slow system performance, including, but not limited to, the following:

- Application server(s) or software
- Database server(s)
- Network – latency, congestion, etc.
- Client-side processing
- Load balancing between multiple servers

Load testing is especially important if the application, system or service will be subject to a service level agreement or SLA.

## Load testing tools

Tool Name	Company Name	Notes
IBM Rational Performance Tester	IBM	Eclipse based large scale performance testing tool primarily used for executing large volume performance tests to measure system response time for server based applications. Licensed.
JMeter	An Apache Jakarta open source project	Java desktop application for load testing and performance measurement.
Load Test (included with Soatest)	Parasoft	Performance testing tool that verifies functionality and performance under load. Supports SOAtest tests, JUnits, lightweight socket-based components. Detects concurrency issues.
LoadRunner	HP	Performance testing tool primarily used for executing large numbers of tests (or a large number or virtual users) concurrently. Can be used for unit and integration testing as well. Licensed.
OpenSTA	Open System Testing Architecture	Open source web load/stress testing application, licensed under the Gnu GPL. Utilizes a distributed software architecture based on CORBA. OpenSTA binaries available for Windows.
SilkPerformer	Micro Focus	Performance testing in an open and sharable model which allows realistic load tests for thousands of users running business scenarios across a broad range of enterprise application environments.
SLAMD		Open source, 100% Java web application, scriptable, distributed with Tomcat.
Visual Studio Load Test	Microsoft	Visual Studio includes a load test tool which enables a developer to execute a variety of tests (web, unit etc...) with a combination of configurations to simulate real user load.

## ***Mechanical load testing***

The purpose of a mechanical load test is to verify that all the component parts of a structure including materials, base-fixings are fit for task and loading it is designed for.

The *Supply of Machinery (Safety) Regulation 1992 UK* state that load testing is undertaken before the equipment is put into service for the first time.

Load testing can be either **Performance**, **Static** or **Dynamic**.

**Performance** testing is when the stated safe working load (SWL) for a configuration is used to determine that the item performs to the manufactures specification. If an item fails this test then any further tests are pointless.

**Static** testing is when a load at a factor above the SWL is applied. The item is not operated through all configurations as it is not a requirement of this test.

**Dynamic** testing is when a load at a factor above the SWL is applied. The item is then operated fully through all configurations and motions. Care must be taken during this test as there is a great risk of catastrophic failure if incorrectly carried out.

The design criteria, relevant legislation or the *Competent Person* will dictate what test is required.

Under the *Lifting Operations and Lifting Equipment Regulations 1998 UK* load testing after the initial test is required if a major component is replaced, if the item is moved from one location to another or as dictated by the *Competent Person*

The loads required for a test are stipulated by the item under test, but here are a few to be aware off. Powered lifting equipment **Static** test to 1.25 SWL and **dynamic** test to 1.1 SWL. Manual lifting equipment **Static** test to 1.5 SWL

For lifting accessories. 2 SWL for items up to 30 tonne capacity. 1.5 SWL for items above 30 tonne capacity. 1 SWL for items above 100 tonnes.

### ***Car charging system***

A load test can be used to evaluate the health of a car's battery. The tester consists of a large resistor that has a resistance similar to a car's starter motor and a meter to read the battery's output voltage both in the unloaded and loaded state. When the tester is used, the battery's open circuit voltage is checked first. If the open circuit voltage is below spec (12.6 volts for a fully charged battery), the battery is charged first. After reading the battery's open circuit voltage, the load is applied. When applied, it draws approximately the same current the car's starter motor would draw during cranking. Based on the specified cold cranking amperes of the battery, if the voltage under load falls below a certain point, the battery is bad. Load tests are also used on running cars to check the output of the car's alternator.

## Chapter-13

# Trent Accreditation Scheme

The **Trent Accreditation Scheme (TAS)**, now replaced *de facto* by QHA Trent Accreditation, was a British accreditation scheme formed with a mission to maintain and continually evaluate standards of quality, especially in health care delivery, through the surveying and accreditation of health care organisations, especially hospitals and clinics, both in the UK and elsewhere in the world.

The Trent UK Accreditation Scheme, or TAS UK, ceased to operate in May 2010, when a majority of Board members decided to end the scheme.

Subsequently, the British-based scheme QHA Trent Accreditation began operating.

### ***History of the scheme***



The logo of the former Trent Accreditation Scheme

Trent's basic mission resembled that of the USA's Joint Commission International, or JCI, and other major international healthcare accreditation groups, although there were some significant differences in the way the different groups work.

Apart from hospitals in the United Kingdom, Trent also surveyed a large number of private sector hospitals in Hong Kong and at the time of its demise had been developing links with hospitals in Cyprus.

The approach Trent took to Clinic and Hospital Accreditation was based on the axiom that no single healthcare system, whether European, American, Asian or otherwise in origin, has the right to claim a monopoly viewpoint over what represents acceptable quality and best clinical practice throughout the world, and no one country has the absolute right to tell another how their hospitals should be run. What is vital is that the

quality of care which patients receive should be of the highest possible standard, and also that the hospitals and clinics providing that care should be independently capable when it comes to working out how best to maintain those standards and how best to respond to any new challenges which will inevitably come along. If the overall standards of a hospital or clinic can be shown to be of acceptable quality, then it is desirable, and even ideal, that local differences related to culture and to legislation should be specifically discussed and incorporated into the assessment standards in an appropriate fashion. That said, Trent was very interested in the medical ethical standards of the hospitals it worked with.

To achieve all of this, Trent worked in close partnership with participating hospitals and clinics to generate an appropriate and mutually acceptable set of standards to survey against. Because the world of healthcare is constantly changing, the standards were constantly reviewed and up-dated through a system of working jointly with representatives of partner hospitals.

Trent developed various ways to ensure local participation, and even ownership, over the accreditation process in a locality. Trent utilised UK-sourced surveyors who were either working in the British National Health Service, or NHS, or had retired in recent times, and hence have valuable experience and insight "at the coal face", and in Hong Kong Trent also appoints locally-domiciled surveyors. Trent surveyors are drawn from a wide variety of professional backgrounds, but especially from the worlds of medicine, dentistry, nursing, the professions complementary to medicine (e.g. physiotherapy, pharmacy etc.) and healthcare management/administration, so as to ensure an appropriately broad portfolio of knowledge and skills are always present within the surveying teams and the wider organisation. Surveyors were all volunteer professionals rather than salaried employees.

Trent surveys were not just a matter of working through a "tick-list" of standards, a process which Trent believed might elevate standards to a certain level but nevertheless do little to inculcate a culture of "thinking for oneself" – instead, Trent surveys involved direct face-to-face conversation with all levels of staff, including clinical medical staff and senior management (for this reason, qualified medical doctors are included in all surveying teams organised by Trent) and Trent surveyors expected full freedom to go anywhere in the hospital or clinic under survey and to talk to anyone they choose to. Discussion and analysis of the data thus generated, not only by the Trent team but also by the hospital or clinic under survey, represented a major component of Trent's approach to hospital and clinic accreditation, and reflected an underlying philosophy that the whole process was about improving services to patients and the ability of an organisation to work effectively towards that aim.

Trent surveyors evaluated a vast range of modalities of a hospital's (or clinic's) activities and governance, including management, estates, equipment, clinical audit, research, education and training, as well as clinical/medical activity. In Hong Kong hospitals, survey teams always consisted of 2 or 3 surveyors from the UK working together with (usually) 2 based in Hong Kong and who were actively working in the local hospitals.

One surveyor will be nominated as the lead. The Hong Kong-based surveyors were nominated by the participating hospitals, and after receiving training they always surveyed hospitals other than their own. This approach led to unrivalled opportunity and potential for the sharing of ideas about best practice between hospitals working in the same locality, and the development of camaraderie. Also, patients were spoken to, and their views and experiences also sought.

At the end of a survey, the key findings were initially presented by the Lead Surveyor to the hospital or clinic undergoing the survey, this event taking place almost always on the last day. The findings were subsequently digested, analysed and put into a more detailed printed report, with positive virtues being highlighted as well as problems. However, because of the end-of-survey oral presentation, hospitals and clinics could start putting remedial action into place as soon as possible.

After a round of surveys, a joint meeting was held at which the printed reports of all the hospital and clinic surveys conducted in that particular round are discussed jointly and in depth by the Trent Board (which had both local and UK representation) together with senior representatives of the hospital or clinic being surveyed, and a decision was then taken as to whether or not accreditation would be granted unconditionally, or if it would be subject to conditions.

The Trent approach to accreditation ensured that the local hospitals and clinics enjoyed some ownership over the whole process, which would not be the case if all of the standards, all of the surveyors and all of the decisions regarding who was successful or not in achieving accreditation were imposed unilaterally from outside. It helped to build up the confidence of participating hospitals in their ability to develop ways to maintain and improve quality in a way that schemes which operate da more didactic approach to standards and their assessment would not. It also meant that there were Trent surveyors constantly present in the majority of the scheme's participating hospitals.

Trent was a member of the United Kingdom Accreditation Forum (UKAF) and an institutional member of ISQUA.

### ***International Healthcare Accreditation***

With the advent of medical tourism, international healthcare accreditation has increasingly grown in importance. A number of accreditation organisations sourced from a number of countries fulfil this internationally-orientated role, including:

- Accreditation Canada International (formerly "The Canadian Council on Health Services Accreditation", or CCHSA)
- Joint Commission International (JCI), in the USA
- The Australian Council on Healthcare Standards, or ACHS
- QHA Trent Accreditation (UK)

No single accreditation scheme enjoys exclusive rights to be seen as an overall world-wide-relevant scheme, and some hospitals are looking towards multiple accreditation to achieve performance credibility in different parts of the world.

The Trent Scheme was the first accreditation scheme to survey and accredit a hospital in Asia, in Hong Kong in 2000 . Since then others such as JCI have entered the market, with JCI first accrediting Bumrungrad International Hospital in Thailand in 2002.

### ***Closure of Trent Accreditation Scheme***

The UK-based Trent Accreditation Scheme (TAS UK) ceased surveying and accreditation activities in 2010. Subsequently, the British-based scheme QHA Trent Accreditation began operating.

## Chapter-14

# QCReporting and Software Assurance

## QCReporting

**QC Reporting** is a web based reporting tool that extracts data to produce reports from the database used by the test tool HP Quality Center and its predecessor HP Test Director. The resulting reporting media format can be, where possible:

- PDF
- Microsoft Excel
- Screen table
- Adobe Flash Graph
- Microsoft Word
- HTML

This product was formerly known as *TD Reports* which was based on the predecessor to Quality Center known as Test director during the former Mercury Interactive days.

QCReporting benefits testing resources who spending time writing and maintaining test reports by enabling them to produce reports quickly and accurately. QCReporting is usually installed as part of a company's intranet. This means that reports can be created and presented easily during meetings using the company Wifi whilst containing up-to-date data.

## Need for some QC reporting tool



### QCReporting Completion forecast

The testing product that is used with QCReporting is Quality Center. Quality Center comes with some reports. These reports can be built and set by the user but they often lead to very long HTML output. Over the years testers, usually the *test manager* of a test team would spend time creating and formatting reports for senior management. This process has been known to take a full working day to compile, usually involving multiple Excel report extracts. This in turn led to the writing of custom reports. These were often developed using one of two ways.

- By connecting to Quality Center using 'VBScript' report script which extracts the data and creates a formatted report. The format of the report is set in the code.
- By scripting with VBScript inside Quality Center in its workflow area. Quality Center allows adding custom buttons to its dashboard, which can then be used to trigger the custom report script.

The former option became the most popular with extracts also being written to Microsoft Excel which could then be used to create graphs.

The need for up-to-date information spurred on ever more ways to extract data from Quality Center and format a report.

## ***Example of a VB Script used to extract a QC report***

The following VBScript can be executed by typing **wscript scriptname.vbs** at the command prompt.

The tool helps make the need for scripts such as these a thing of the past. The following script is an example of what a test manager would have had to use to extract data from Quality Center.

```
Option Explicit
Dim QCConnection
'-----
Dim Excel, Sheet, CurrRow
Dim UserN, QCUserN, UserP, QCDomain, QCProj, QCAddr, XLSSavePos
Dim REQUsersTreeStr, WSHNetwork
'-----
QCUserN = "QC_User_Name"
UserP = "QC_User_Password"
QCDomain = "QC_Domain_Name"
QCProj = "QC_Project_Name"
QCAddr = "http://SomeServer:8080/qcbin"
'-----
WScript.Echo "Initialising...."
Set Excel = CreateObject("Excel.Application") 'Open Excel
Excel.ScreenUpdating = False
Set WSHNetwork = WScript.CreateObject("WScript.Network")
UserN = WshNetwork.UserName
Set WSHNetwork = Nothing
Set QCConnection = CreateObject("TDApiOle80.TDConnection")
QCConnection.InitConnectionEx QCAddr
QCConnection.login QCUserN, UserP
QCConnection.Connect QCDomain, QCProj
If (QCConnection.Connected <> True) Then
    WScript.Echo "QC Project Failed to Connect"
    WScript.Quit
End If
'-----
On Error Resume Next
WScript.Echo "Extract from QC here and format it out"
'-----
If Err.number <> 0 Then WScript.Echo Err.Description & ":" & Err.Number
'-----
'Note: This will attempt to save the XLS into UserN's DESKTOP. please
check the folder location first
XLSSavePos = "C:\Documents and Settings\" & UserN & "\Desktop\" &
QCProj & ".xls"
Excel.ScreenUpdating = True
WScript.Echo "Attempting to save to folder:" & XLSSavePos & ""
Excel.ActiveWorkbook.SaveAs (XLSSavePos)
Excel.Quit
Set Excel = Nothing
'-----
QCConnection.Disconnect
QCConnection.Logout
```

```
QCConnection.ReleaseConnection  
WScript.Echo "Report Complete"
```

## ***Marriage of QCReporting to Quality Center***

There are a number of ways where a marriage relationship is apparent between the reporting tool and Quality Center.

- Users are not created in QCReporting, instead they are synchronised from Quality Center. The synchronisation includes their user settings and access rights to domains and projects.
- Data managed and altered in Quality Center is reported on using a read-only user in QCReporting.
- Customisation, such as user fields, made in Quality Center are reported on through QCReporting.

## ***QCReporting tool support***

The tool offers user support and functionality to manage itself.

### User Support

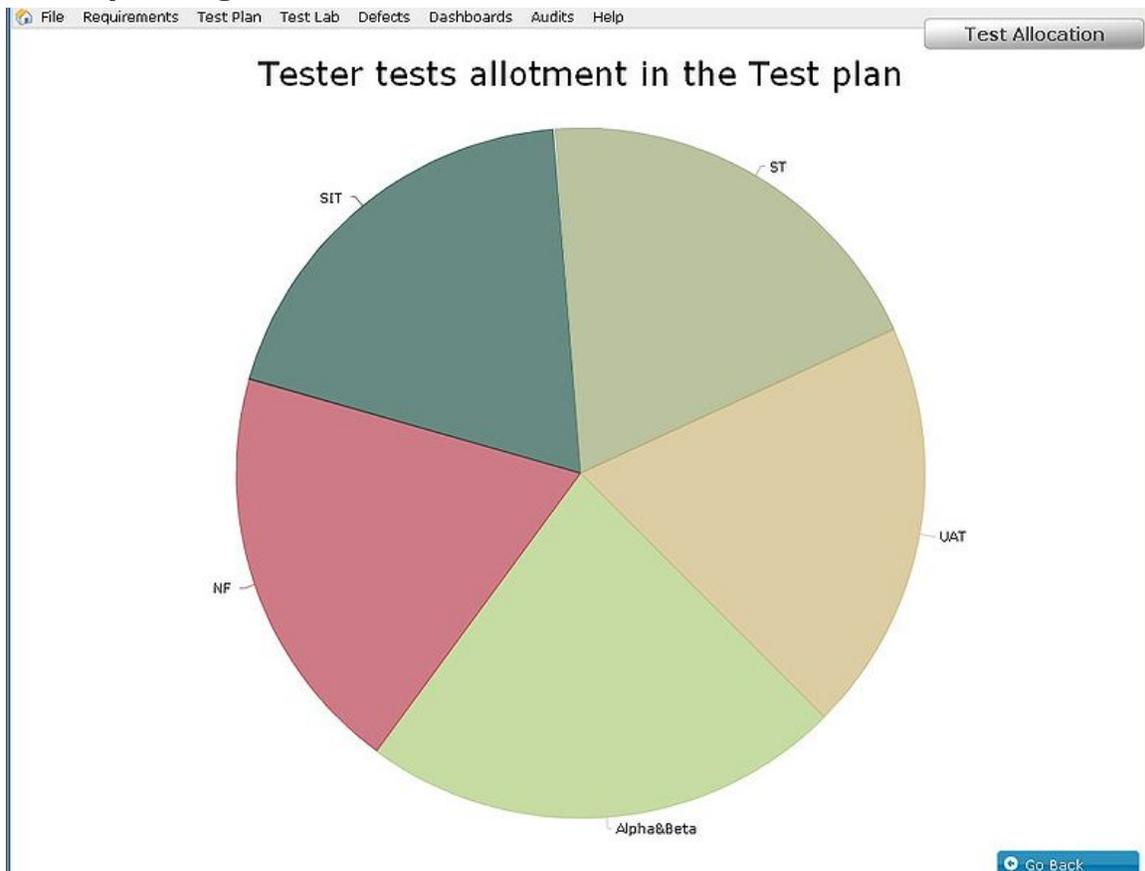
User support is primarily managed through a web based ticket system off the clients own installed instance of the tool.

- Tickets can be user or group owned
- Tickets can be escalated
- Simple requests such as wanting a new report can be raised as a ticket
- Fixes to tickets appear through the automatic update system

### Tool maintenance

- An **auto update** facility downloads and installs the latest updates from the support site. The update is encrypted over https.
- The tool verifies each transmission with its unique client license information

## QCReporting was born



QCReporting Test Allocation

Quality Center is a testing tool that is very common in the testing world. The test tool is divided into four main categories. The QCReporting follows those categories. They are:

- Requirements & coverage assurance report. These can be business, design, functional and non-functional requirements.
- Test plan
- Test Lab
- Defects
- Dashboards. These are overview cross-project and cross-domain reports. Most popular with upper management.
- Test Exit reports - at the end of every phase and cycle issue the report to summarise the finding, the coverage and the outcome.
- The additional category is Audit reports which follow the Sarbanes–Oxley legislation.

The tool delivers fantastic time saving and greater levels of accuracy: producing the Test Exit report, which includes all the customised fields of QC is just one button click away; or a few if you would like to exclude certain parts of the default settings.

The tool provides comprehensive overview of the test coverage reports showing horizontal and vertical traceability between all of the project requirements and test scripts whilst highlighting where there may be gaps, such as missing requirements or orphaned test scripts.

"How are we doing?" Many Project and Test Managers have heard this never-aging question. How do you explain it in not so many words, as there is little patience, time or desire to listen to a lengthy and wordy explanation. With QCReporting the answer is just one click away - a comprehensive and yet easy-to-understand dashboard spread across several projects and even domains can appear within mere seconds. It tells you where you are with each project and across the function or the domain and provides some vital indicators of the project "health" and its testing phase status and maturity levels.

The tool plays a vital role in the Impact Assessment procedure, providing conscious and unambiguous views of the real impact of the change or of the delay to the defect fix. There is no longer a need to guess - one click away is the information on which requirements were affected and to which extent, which scripts need to be re-written and which ones will have to be amended; what the regression implication is and, if related to the already tested piece of work, how long did it take to execute in the past.

There are several additional features which will be available shortly.

What's even better is that one of the tool's add-ins such as the Auto-Updater is used for its self-maintenance and to provide instant user support.

## **Software assurance**

**Software Assurance** (SwA) is defined as “the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle, and that the software functions in the intended manner.”

### ***Alternate definitions***

#### **Department of Homeland Security (DHS)**

According to the DHS, software assurance addresses:

- Trustworthiness - No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended;
- Conformance - Planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements, standards/ procedures.

Contributing SwA disciplines, articulated in Bodies of Knowledge and Core Competencies: Software Engineering, Systems Engineering, Information Systems Security Engineering, Information Assurance, Test and Evaluation, Safety, Security, Project Management, and Software Acquisition.

Software Assurance is a strategic initiative of the U.S. Department of Homeland Security (DHS) to promote integrity, security, and reliability in software. The SwA Program is based upon the National Strategy to Secure Cyberspace - Action/Recommendation 2-14:

“DHS will facilitate a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development.”

### **United States Department of Defense (DoD)**

According to the DoD, software assurance relates to "the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software."

### **Software Assurance Metrics and Tool Evaluation (SAMATE) project**

According to the NIST SAMATE project, software assurance is "the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures to help achieve:

- Trustworthiness - No exploitable vulnerabilities exist, either of malicious or unintentional origin, and
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended."

### **National Aeronautics and Space Administration (NASA)**

According to NASA, Software Assurance is a "planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. It includes the disciplines of Quality Assurance, Quality Engineering, Verification and Validation, Nonconformance Reporting and Corrective Action, Safety Assurance, and Security Assurance and their application during a software life cycle." The NASA Software Assurance Standard also states: "The application of these disciplines during a software development life cycle is called Software Assurance."

### **Object Management Group (OMG)**

According to the OMG, Software Assurance is “justifiable trustworthiness in meeting established business and security objectives.”

OMG's SwA Special Interest Group (SIG), works with Platform and Domain Task Forces and other software industry entities and groups external to the OMG, to coordinate the establishment of a common framework for analysis and exchange of information related to software trustworthiness by facilitating the development of a specification for a Software Assurance Framework that will:

- Establish a common framework of software properties that can be used to represent any/all classes of software so software suppliers and acquirers can represent their claims and arguments(respectively), along with the corresponding evidence, employing automated tools (to address scale)
- Verify that products have sufficiently satisfied these characteristics in advance of product acquisition, so that system engineers/integrators can use these products to build (compose) larger assured systems with them
- Enable industry to improve visibility into the current status of software assurance during development of its software
- Enable industry to develop automated tools that support the common framework.

### **Software Assurance Forum for Excellence in Code (SAFECode)**

According to SAFECode, Software Assurance is “confidence that software, hardware and services are free from intentional and unintentional vulnerabilities and that the software functions as intended.”

### **Webopedia**

According to Webopedia, Software Quality Assurance, abbreviated as SQA, and also called "software assurance", is a level of confidence that software is free from vulnerabilities, either intentionally designed into the software or inserted at anytime during its lifecycle, and that the software functions in the intended manner."

As indicated in the Webopedia definition, the term "software assurance" has been used as a shorthand for Software Quality Assurance (SQA) when not necessarily considering security or trustworthiness. SQA is defined in the *Handbook of Software Quality Assurance* as: "the set of systematic activities providing evidence of the ability of the software process to produce a software product that is fit to use."

## Chapter-15

# 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 (errors or other defects).

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 other performance, behavior under certain constraints, or security. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

## 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 is a part of the software quality assurance (SQA) process. In SQA, software process specialists and auditors are concerned for the software development process rather than just the artefacts such as documentation, code and systems. 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, 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. The main levels during the development process as defined by the SWEBOOK guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective.

## Test target

### 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.

### Objectives of testing

#### 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 is 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. It approach towards the use of the application.

## **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.

- Technical terminology may become inconsistent if the project is translated by several people without proper coordination or if the translator is imprudent.
- 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 crash or malfunction.
- 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).

- A regression testing technique is to have a standard set of tests, which cover existing functionality that result in persistent tabular data, and to compare pre-change data to post-change data, where there should not be differences, using a tool like diffkit. Differences detected indicate unexpected functionality changes or "regression".

## 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 use this methodology but also the 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-16

# 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 not 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.

## 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, Citrix, and 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

## **Methodology**

### **Performance Testing Web Applications Methodology**

According to the Microsoft Developer Network the Performance Testing Methodology consists of the following activities:

- **Activity 1. Identify the Test Environment.** Identify the physical test environment and the production environment as well as the tools and resources available to the test team. The physical environment includes hardware, software, and network configurations. Having a thorough understanding of the entire test environment at the outset enables more efficient test design and planning and helps you identify testing challenges early in the project. In some situations, this process must be revisited periodically throughout the project's life cycle.
- **Activity 2. Identify Performance Acceptance Criteria.** Identify the response time, throughput, and resource utilization goals and constraints. In general, response time is a user concern, throughput is a business concern, and resource utilization is a system concern. Additionally, identify project success criteria that may not be captured by those goals and constraints; for example, using performance tests to evaluate what combination of configuration settings will result in the most desirable performance characteristics.
- **Activity 3. Plan and Design Tests.** Identify key scenarios, determine variability among representative users and how to simulate that variability, define test data, and establish metrics to be collected. Consolidate this information into one or more models of system usage to be implemented, executed, and analyzed.
- **Activity 4. Configure the Test Environment.** Prepare the test environment, tools, and resources necessary to execute each strategy as features and components become available for test. Ensure that the test environment is instrumented for resource monitoring as necessary.
- **Activity 5. Implement the Test Design.** Develop the performance tests in accordance with the test design.
- **Activity 6. Execute the Test.** Run and monitor your tests. Validate the tests, test data, and results collection. Execute validated tests for analysis while monitoring the test and the test environment.
- **Activity 7. Analyze Results, Tune, and Retest.** Analyse, Consolidate and share results data. Make a tuning change and retest. Improvement or degradation? Each improvement made will return smaller improvement than the previous improvement. When do you stop? When you reach a CPU bottleneck, the choices then are either improve the code or add more CPU.

## Chapter-17

# Graphical user Interface Testing

In software engineering, **graphical user interface testing** is the process of testing a product's 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 relatively 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.

## ***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.

## Chapter-18

# Evidence-based Medicine

**Evidence-based medicine (EBM)** or **evidence-based practice (EBP)** aims to apply the best available evidence gained from the scientific method to clinical decision making. It seeks to assess the strength of evidence of the risks and benefits of treatments (including lack of treatment) and diagnostic tests. Evidence quality can range from meta-analyses and systematic reviews of double-blind, placebo-controlled clinical trials at the top end, down to conventional wisdom at the bottom.

EBM/EBP recognizes that many aspects of health care depend on individual factors such as quality- and value-of-life judgments, which are only partially subject to scientific methods. EBP, however, seeks to clarify those parts of medical practice that are in principle subject to scientific methods and to apply these methods to ensure the best *prediction* of outcomes in medical treatment, even as debate continues about which outcomes are desirable.

Because this approach is used in allied related fields, including dentistry, nursing, and psychology, *evidenced-based practice* is a more encompassing term.

### **Classification**

Two types of evidence-based practice have been proposed.

### **Evidence-based guidelines**

Evidence-based guidelines (EBG) is the practice of evidence-based medicine at the organizational or institutional level. This includes the production of guidelines, policy, and regulations. This approach has also been called evidence based healthcare.

### **Evidence-based individual decision making**

Evidence-based individual decision (EBID) making is evidence-based medicine as practiced by the individual health care provider. There is concern that current evidence-based medicine focuses excessively on EBID. The American Academy of Family

Physicians (AAFP) says that DynaMed may be of assistance to family physicians in answering clinical questions with high-quality evidence.

## ***Process and progress***

Using techniques from science, engineering, and statistics, such as the systematic review of medical literature, meta-analysis, risk-benefit analysis, and randomized controlled trials (RCTs), EBM aims for the ideal that healthcare professionals should make "conscientious, explicit, and judicious use of current best evidence" in their everyday practice. *Ex cathedra* statements by the "medical expert" are considered to be least valid form of evidence. All "experts" are now expected to reference their pronouncements to scientific studies.

The systematic review of published research studies is a major method used for evaluating particular treatments. The Cochrane Collaboration is one of the best-known, respected examples of systematic reviews. Like other collections of systematic reviews, it requires authors to provide a detailed and repeatable plan of their literature search and evaluations of the evidence. Once all the best evidence is assessed, treatment is categorized as "likely to be beneficial", "likely to be harmful", or "evidence did not support either benefit or harm".

A 2007 analysis of 1016 systematic reviews from all 50 Cochrane Collaboration Review Groups found that 44% of the reviews concluded that the intervention was "likely to be beneficial", 7% concluded that the intervention was "likely to be harmful", and 49% concluded that evidence "did not support either benefit or harm". 96% recommended further research. A 2001 review of 160 Cochrane systematic reviews (excluding complementary treatments) in the 1998 database revealed that, according to two readers, 41.3% concluded positive or possibly positive effect, 20% concluded evidence of no effect, 8.1% concluded net harmful effects, and 21.3% of the reviews concluded insufficient evidence. A review of 145 alternative medicine Cochrane reviews using the 2004 database revealed that 38.4% concluded positive effect or possibly positive (12.4%) effect, 4.8% concluded no effect, 0.69% concluded harmful effect, and 56.6% concluded insufficient evidence.

Generally, there are three distinct, but interdependent, areas of EBM. The first is to treat individual patients with acute or chronic pathologies by treatments supported in the most scientifically valid medical literature. Thus, medical practitioners would select treatment options for specific cases based on the best research for each patient they treat. The second area is the systematic review of medical literature to evaluate the best studies on specific topics. This process can be very human-centered, as in a journal club, or highly technical, using computer programs and information techniques such as data mining. Increased use of information technology turns large volumes of information into practical guides. Finally, evidence-based medicine can be understood as a medical "movement" in which advocates work to popularize the method and usefulness of the practice in the public, patient communities, educational institutions, and continuing education of practicing professionals.

## ***Ranking the quality of evidence***

Evidence-based medicine categorizes different types of clinical evidence and rates or grades them according to the strength of their freedom from the various biases that beset medical research. For example, the strongest evidence for therapeutic interventions is provided by systematic review of randomized, triple-blind, placebo-controlled trials with allocation concealment and complete follow-up involving a homogeneous patient population and medical condition. In contrast, patient testimonials, case reports, and even expert opinion have little value as proof because of the placebo effect, the biases inherent in observation and reporting of cases, difficulties in ascertaining who is an expert, and more.

### **US Preventive Services Task Force**

Systems to stratify evidence by quality have been developed, such as this one by the U.S. Preventive Services Task Force for ranking evidence about the effectiveness of treatments or screening:

- Level I: Evidence obtained from at least one properly designed randomized controlled trial.
- Level II-1: Evidence obtained from well-designed controlled trials without randomization.
- Level II-2: Evidence obtained from well-designed cohort or case-control analytic studies, preferably from more than one center or research group.
- Level II-3: Evidence obtained from multiple time series with or without the intervention. Dramatic results in uncontrolled trials might also be regarded as this type of evidence.
- Level III: Opinions of respected authorities, based on clinical experience, descriptive studies, or reports of expert committees.

### **National Health Service**

The UK National Health Service uses a similar system with categories labeled A, B, C, and D. The above Levels are only appropriate for treatment or interventions; different types of research are required for assessing diagnostic accuracy or natural history and prognosis, and hence different "levels" are required. For example, the Oxford Centre for Evidence-based Medicine suggests levels of evidence (LOE) according to the study designs and critical appraisal of prevention, diagnosis, prognosis, therapy, and harm studies:

- Level A: Consistent Randomised Controlled Clinical Trial, cohort study, all or none, clinical decision rule validated in different populations.
- Level B: Consistent Retrospective Cohort, Exploratory Cohort, Ecological Study, Outcomes Research, case-control study; or extrapolations from level A studies.
- Level C: Case-series study or extrapolations from level B studies.

- Level D: Expert opinion without explicit critical appraisal, or based on physiology, bench research or first principles.

## **GRADE Working Group**

A newer system is by the GRADE Working Group and takes in account more dimensions than just the quality of medical evidence. "Extrapolations" are where data is used in a situation which has potentially clinically important differences than the original study situation. Thus, the quality of evidence to support a clinical decision is a combination of the quality of research data and the clinical 'directness' of the data.

Despite the differences between systems, the purposes are the same: to guide users of clinical research information about which studies are likely to be most valid. However, the individual studies still require careful critical appraisal.

Note: The all or none principle is met when all patients died before the Rx became available, but some now survive on it; or when some patients died before the Rx became available, but none now die on it.

## **Categories of recommendations**

In guidelines and other publications, recommendation for a clinical service is classified by the balance of risk versus benefit of the service *and* the level of evidence on which this information is based. The U.S. Preventive Services Task Force uses:

- Level A: Good scientific evidence suggests that the benefits of the clinical service substantially outweigh the potential risks. Clinicians should discuss the service with eligible patients.
- Level B: At least fair scientific evidence suggests that the benefits of the clinical service outweighs the potential risks. Clinicians should discuss the service with eligible patients.
- Level C: At least fair scientific evidence suggests that there are benefits provided by the clinical service, but the balance between benefits and risks are too close for making general recommendations. Clinicians need not offer it unless there are individual considerations.
- Level D: At least fair scientific evidence suggests that the risks of the clinical service outweighs potential benefits. Clinicians should not routinely offer the service to asymptomatic patients.
- Level I: Scientific evidence is lacking, of poor quality, or conflicting, such that the risk versus benefit balance cannot be assessed. Clinicians should help patients understand the uncertainty surrounding the clinical service.

## **Statistical measures**

Evidence-based medicine attempts to express clinical benefits of tests and treatments using mathematical methods. Tools used by practitioners of evidence-based medicine include:

### **Likelihood ratio**

The pretest odds of a particular diagnosis, multiplied by the likelihood ratio, determines the post-test odds. (Odds can be calculated from, and then converted to, the [more familiar] probability.) This reflects Bayes' theorem. The differences in likelihood ratio between clinical tests can be used to prioritize clinical tests according to their usefulness in a given clinical situation.

### **AUC-ROC**

The area under the receiver operating characteristic curve (AUC-ROC) reflects the relationship between sensitivity and specificity for a given test. High-quality tests will have an AUC-ROC approaching 1, and high-quality publications about clinical tests will provide information about the AUC-ROC. Cutoff values for positive and negative tests can influence specificity and sensitivity, but they do not affect AUC-ROC.

### **Number needed to treat / harm**

Number needed to treat or Number needed to harm are ways of expressing the effectiveness and safety of an intervention in a way that is clinically meaningful. In general, NNT is always computed with respect to two treatments A and B, with A typically a drug and B a placebo (in our example above, A is a 5-year treatment with the hypothetical drug, and B is no treatment). A defined endpoint has to be specified (in our example: the appearance of colon cancer in the 5 year period). If the probabilities  $p_A$  and  $p_B$  of this endpoint under treatments A and B, respectively, are known, then the NNT is computed as  $1/(p_B - p_A)$ . The NNT for breast mammography is 285; that is, 285 mammograms need to be performed to diagnose one breast cancer. As another example, an NNT of 4 means if 4 patients are treated, only one would respond.

An NNT of 1 is the most effective and means each patient treated responds, e.g., in comparing antibiotics with placebo in the eradication of *Helicobacter pylori*. An NNT of 2 or 3 indicates that a treatment is quite effective (with one patient in 2 or 3 responding to the treatment). An NNT of 20 to 40 can still be considered clinically effective.

### **Quality of clinical trials**

Evidence-based medicine attempts to objectively evaluate the quality of clinical research by critically assessing techniques reported by researchers in their publications.

- Trial design considerations. High-quality studies have clearly defined eligibility criteria, and have minimal missing data.
- Generalizability considerations. Studies may only be applicable to narrowly defined patient populations, and may not be generalizable to clinical practice.
- Followup. Sufficient time for defined outcomes to occur can influence the study outcomes and the statistical power of a study to detect differences between a treatment and control arm.
- Power. A mathematical calculation can determine if the number of patients is sufficient to detect a difference between treatment arms. A negative study may reflect a lack of benefit, or simply a lack of sufficient quantities of patients to detect a difference.

## **Limitations**

Although evidence-based medicine is becoming regarded as the "gold standard" for clinical practice there are a number of limitations and criticisms of its use.

## **Ethics**

In some cases, such as in open-heart surgery, conducting randomized, placebo-controlled trials is commonly considered to be unethical, although observational studies may address these problems to some degree.

## **Cost**

The types of trials considered "gold standard" (i.e. large randomized double-blind placebo-controlled trials) are expensive, so that funding sources play a role in what gets investigated. For example, public authorities may tend to fund preventive medicine studies to improve public health, while pharmaceutical companies fund studies intended to demonstrate the efficacy and safety of particular drugs.

## **Generalizability**

Furthermore, evidence-based guidelines do not remove the problem of extrapolation to different populations or longer timeframes. Even if several top-quality studies are available, questions always remain about how far, and to which populations, their results are "generalizable". Furthermore, skepticism about results may always be extended to areas not explicitly covered: for example, a drug may influence a "secondary endpoint" such as test result (blood pressure, glucose, or cholesterol levels) without having the power to show that it decreases overall mortality or morbidity in a population.

The quality of studies performed varies, making it difficult to compare them and generalize about the results.

Certain groups have been historically under-researched (racial minorities and people with many co-morbid diseases), and thus the literature is sparse in areas that do not allow for generalizing.

## **Publication bias**

It is recognised that not all evidence is made accessible, that this can limit the effectiveness of any approach, and that efforts to reduce publication bias and retrieval bias is required.

Failure to publish negative trials is the most obvious gap, and moves to register all trials at the outset, and then to pursue their results, are underway. Changes in publication methods, particularly related to the Web, should reduce the difficulty of obtaining publication for a paper on a trial that concludes it did not prove anything new, including its starting hypothesis.

Treatment effectiveness reported from clinical studies may be higher than that achieved in later routine clinical practice due to the closer patient monitoring during trials that leads to much higher compliance rates.

The studies that are published in medical journals may not be representative of all the studies that are completed on a given topic (published and unpublished) or may be unreliable due to conflicts of interest. Thus the array of evidence available on particular therapies may not be well-represented in the literature. A 2004 statement by the International Committee of Medical Journal Editors (that they will refuse to publish clinical trial results if the trial was not recorded publicly at its outset) may help with this, although this has not yet been implemented.

## **Ghost writers**

## **Populations, clinical experience, and dubious diagnoses**

EBM applies to groups of people but this does not preclude clinicians from using their personal experience in deciding how to treat the person in front of them. In *The limits of evidence-based medicine*, Tonelli advises that "the knowledge gained from clinical research does not directly answer the primary clinical question of what is best for the patient at hand." and suggests that evidence-based medicine should not discount the value of clinical experience.

David Sackett writes that "the practice of evidence based medicine means integrating individual clinical expertise with the best available external clinical evidence from systematic research".

## **Political criticism**

There is a good deal of criticism of evidence based medicine, which is suspected of being — as against what the phrase suggests — in essence a tool not so much for medical science as for health managers, who want to introduce managerialist techniques into medical administration. Thus Dr Michael Fitzpatrick writes: "To some of its critics, in its disparagement of theory and its crude number-crunching, EBM marks a return to 'empiricist quackery' in medical practice . Its main appeal, as Singh and Ernst suggest, is to health economists, policymakers and managers, to whom it appears useful for measuring performance and rationing resources."

## ***In psychiatry***

Standard knowledge about mental illnesses, such as the Diagnostic and Statistical Manual of Mental Disorders, have been criticized as incompletely justified by evidence. In many cases, it is unknown whether a particular "disease" has one, several, or no underlying biological causes (controversy arising over whether some diseases are merely an artifact of the attempt to construct a unified classification scheme, rather than a "real" disease).

While some experts point to statistics in support of the idea that a lack of adoption of research findings results in suboptimal treatment for many patients, others emphasize the importance of the skill of the practitioner and the customization of the treatment to fit individual needs. There is some controversy over whether mental illnesses is too complex for broad population studies to be helpful.

## ***History***

Traces of evidence-based medicine's origin can be found in ancient Greece, Although testing medical interventions for efficacy has existed since the time of Avicenna's *The Canon of Medicine* in the 11th century, it was only in the 20th century that this effort evolved to impact almost all fields of health care and policy. Professor Archie Cochrane, a Scottish epidemiologist, through his book *Effectiveness and Efficiency: Random Reflections on Health Services* (1972) and subsequent advocacy, caused increasing acceptance of the concepts behind evidence-based practice. Cochrane's work was honoured through the naming of centres of evidence-based medical research — *Cochrane Centres* — and an international organization, the Cochrane Collaboration. The explicit methodologies used to determine "best evidence" were largely established by the McMaster University research group led by David Sackett and Gordon Guyatt. Guyatt later coined the term "evidence-based" in 1990. The term "evidence-based medicine" first appeared in the medical literature in 1992 in a paper by Guyatt *et al.* Relevant journals include the British Medical Journal's *Clinical Evidence*, the *Journal Of Evidence-Based Healthcare* and *Evidence Based Health Policy*. All of these were co-founded by Anna Donald, an Australian pioneer in the discipline.

## ***EBM and ethics of experimental or risky treatments***

Insurance companies in the United States and public insurers in other countries usually wait for drug use approval based on evidence-based guidelines before funding a treatment. Where approval for a drug has been given, and subsequent evidence based findings indicating that a drug may be less safe than originally anticipated, some insurers in the U.S. have reacted very cautiously and withdrawn funding. For example, an older generic statin drug had been shown to reduce mortality, but a newer and much more expensive statin drug was found to lower cholesterol more effectively. However, evidence came to light about safety concerns with the new drug which caused some insurers to stop funding it even though marketing approval was not withdrawn. Some people are willing to take their chances to gamble their health on the success of new drugs or old drugs in new situations which may not yet have been fully tested in clinical trials. However insurance companies are reluctant to take on the job of funding such treatments, preferring instead to take the safer route of awaiting the results of clinical testing and leaving the funding of such trials to the manufacturer seeking a license.

Sometimes caution errs in the other direction. Kaiser Permanente did not change its methods of evaluating whether or not new therapies were too "experimental" to be covered until it was successfully sued twice: once for delaying in vitro fertilization treatments for two years after the courts determined that scientific evidence of efficacy and safety had reached the "reasonable" stage; and in another case where Kaiser refused to pay for liver transplantation in infants when it had already been shown to be effective in adults, on the basis that use in infants was still "experimental." Here again, the problem of induction plays a key role in arguments.

## ***Application of the evidence based model on other public policy matters***

There has been discussion of applying what has been learned from EBM to public policy. In his 1996 inaugural speech as President of the Royal Statistical Society, Adrian Smith held out evidence-based medicine as an exemplar for *all* public policy. He proposed that "evidence-based policy" should be established for education, prisons and policing policy and all areas of government.