Jagan

Software Engineering

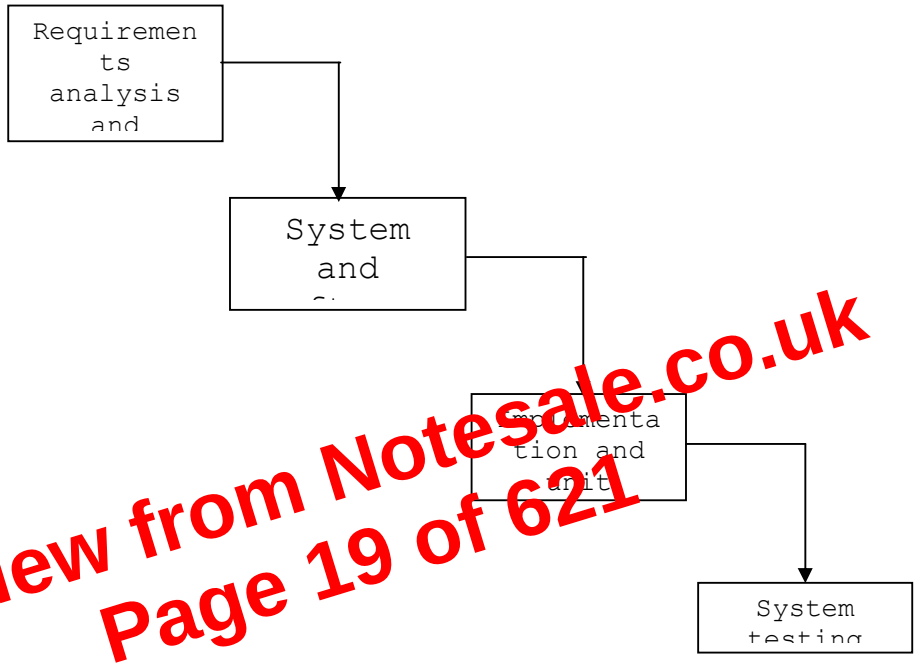


Figure 1.3 The Waterfall model

1.3.3 The prototyping-oriented life-cycle model

Developed in the 1990s [Pomberger 1991].

A prototyping-oriented software development strategy does not differ fundamentally from the classical phase-oriented development strategy. These strategies are more complementary than alternative.

New aspects:

- The phase model is seen not as linear but as iterative, and
- It specifies where and how these iterations are not only possible but necessary.

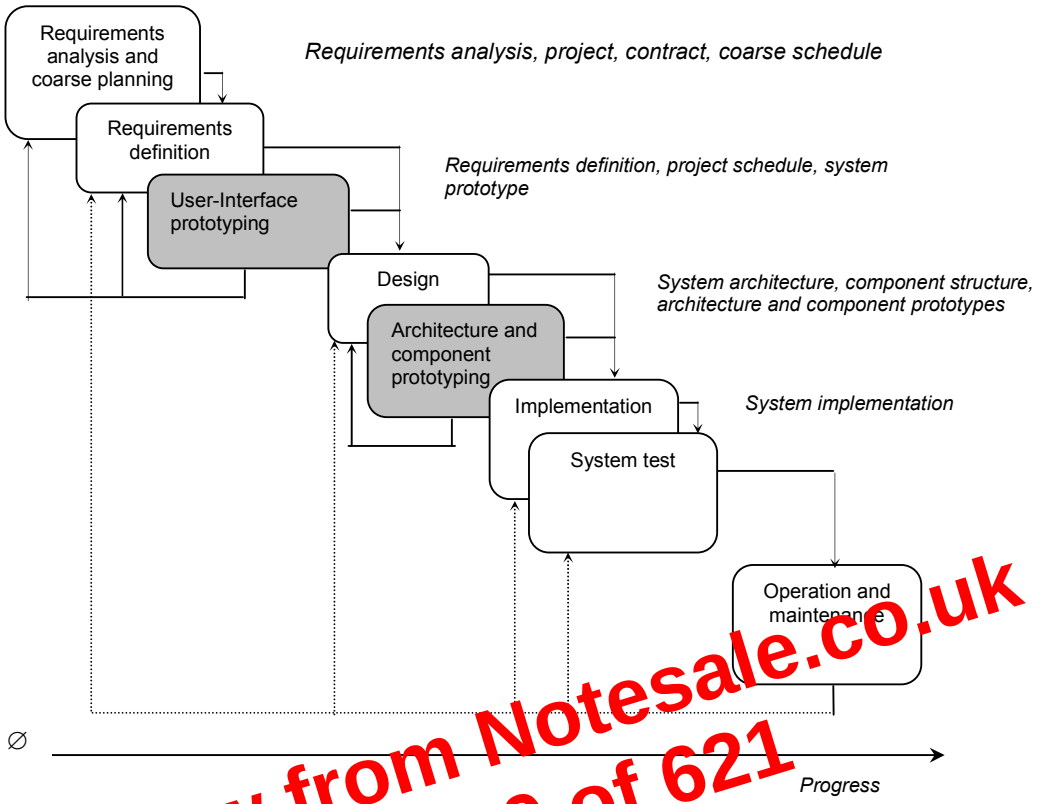


Figure 1.4 Prototyping-oriented software life cycle

Prototype construction is an iterative process (Figure 1.4). First a prototype is produced on the basis of the results of preceding activities. During the specification phase a prototype of the user interface is developed that encompasses significant parts of the functional requirements on the target software system. Experiments are conducted on this prototype that reflect real conditions of actual use in order to determine whether the client's requirements are fulfilled. Under conditions close to reality, software developers and users can test whether the system model contains errors, whether it meets the client's preconceptions, and whether modifications are necessary. This reduces the risk of erroneous or incomplete system specifications and creates a significantly better starting point for subsequent activities.

1.3.4 The spiral model

Developed in 1988 [Boehm 1988].

The spiral model is a software development model that combines the above models or includes them as special cases. The model makes it possible to choose the most suitable approach for a given project. Each cycle encompasses the same sequence of steps for each part of the target product and for each stage of completion.

A spiral cycle begins with the establishment of the following point:

reinvested in this maintenance. A new job title emerges, the class librarian, who is responsible for ensuring the efficient usability of the class library.

- During the test phase, the function of not only the new product but also of the reused components is tested. Any deficiencies in the latter must be documented exactly. The resulting modifications must be handled centrally in the class library to ensure that they impact on other projects, both current and future.
- Newly created classes must be tested for their general usability. If there is a chance that a component could be used in other projects as well, it must be included in the class library and documented accordingly. This also means that the new class must be announced and made accessible to other programmers who might profit from it. This places new requirements on the in-house communication structures.

The class library serves as a tool that extends beyond the scope of an individual project because classes provided by one project can increase productivity in subsequent projects.

The actual software life cycle recurs when new requirements arise in the company that initiate a new requirements analysis stage.

- Description of the prerequisites that must apply for the system to be used.

  Note:

  - *The description of all information that is necessary for the employment of the system, but not part of the implementation.*

- Specification of the number of users, the frequency of use, and the jobs of the users.

3. User interfaces

Contents:

- The human-machine interface.

Notes:

- This section is one of the most important parts of the requirements definition, documenting how the user communicates with the system.
- The quality of this section largely determines the acceptance of the software product.

4. Functional requirements

Contents:

- Definition of the system functionality expected by the user
- All necessary specifications about the type, amount and expected precision of the data associated with each system function

Notes:

- *Good specifications of system functionality contain only the necessary information about these functions.*
- *Any additional specifications, such as about the solution algorithm for a function, distracts from the actual specification task and restricts the flexibility of the subsequent system design.*
- *Only exact determination of value ranges for data permits a plausibility check to detect input errors.*

5. Nonfunctional requirements

Contents:

- Requirements of nonfunctional nature: reliability, portability, and response and processing times ...

Note:

- *For the purpose of the feasibility study, it is necessary to weight these requirements and to provide detailed justification.*

6. Exception handling

Contents:

- Description of the effects of various kinds of errors and the required system behavior upon occurrence of an error.

Note:

- *Developing a reliable system means considering possible errors in each phase of development and providing appropriate measures to prevent or diminish the effects of errors.*

7. Documentation requirements

Contents:

- Establish the scope and nature of the documentation.

Note:

- *The documentation of a system provides the basis for both the correct utilization of the software product and for system maintenance.*

8. Acceptance criteria

Contents:

- Establishing the conditions for inspection of the system by the client.

Notes:

- *The criteria refer to both functional and nonfunctional requirements*
- *The acceptance criteria must be established for each individual system requirement. If no respective acceptance criteria can be found for a given requirement, then we can assume that the client is unclear about the purpose and value of the requirement.*

9. Glossary and index

Contents:

- A *glossary* of terms
- An extensive *index*

Notes:

- *The requirements definition constitutes a document that provides the basis for all phases of a software project and contains preliminary considerations about the entire software life cycle*
- *The specifications are normally not read sequentially, but serve as a reference for lookup purposes.*

2.2 Quality criteria for requirements definition

- It must be *correct* and *complete*.

- It must be *consistent* and *unambiguous*.
- It should be *minimal*.
- It should be *readable* and *comprehensible*.
- It must be readily *modifiable*.

2.3 Fundamental problems in defining requirements

The fundamental problems that arise during system specification are [Keller 1989]:

- The goal/means conflict
- The determination and description of functional requirements
- The representation of the user interfaces

The goal/means conflict in system specification.

- The primary task of the specification process is to establish the *goal* of system development rather than to describe the *means* for achieving the goal.
- The requirements definition describes *what* a system must do, but not *how* the individual functions are to be realized.

Determining and describing the functional requirements.

- Describing functional requirements in the form of text is extremely difficult and leads to very lengthy specifications.
- A system model on the user interface level serving as an executable prototype supports the exploration of functional, nonfunctional and interaction-related requirements.
- It simplifies the determination of dependencies between system functions and abbreviates the requirements definition.
- A prototype that represents the most important functional aspects of a software system represents this system significantly better than a verbal description could do.

Designing the user interfaces.

- User interfaces represent a user-oriented abstraction of the functionality of a system.
- The graphical design of screen layouts requires particular effort and only affects one aspect of the user interface, its appearance.
- The much more important aspect, the dynamics behind a user interface, can hardly be depicted in purely verbal specifications.

Therefore the user interface components of the requirements definition should always be realized as an executable prototype.

2.4 Algebraic specification

Algebraic specification [Guttag 1977] is a technique whereby an object is specified in terms of the relationships between the operations that act on that object.

A specification is presented in four parts (Figure 2.1):

1. **Introduction** part where the sort of the entity being specified is introduced and the name of any other specifications which are required are set out
2. **Informal description** of the sort and its operations
3. **Signature** where the names of the operations on that object and the sorts of their parameters are defined
4. **Axioms** where the relationships between the sort operations are defined.

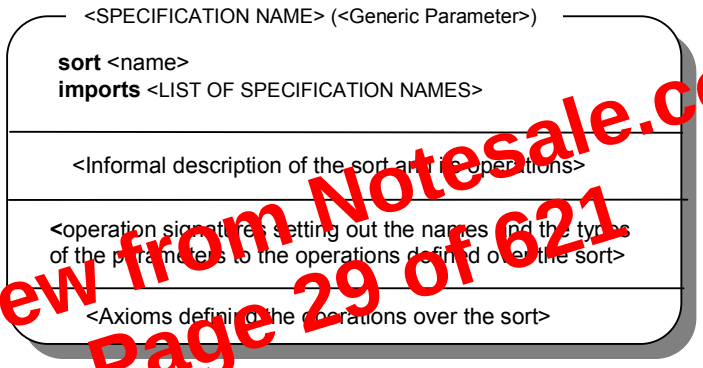


Figure 2.1 The format of an algebraic specification.

Note:

- *The introduction part of a specification also includes an **imports** part which names the other specifications which are required in a specification.*

Description part:

- Formal text with an informal description

Signature part:

- Names of the operations which are defined over the sort,
- Number and sort of their parameters, and
- Sort of the result of evaluating of the operation.

Axioms part:

- A great many software systems, particularly embedded real-time systems, are structured as a set of parallel communicating processes which is an outline design of a simple control system.
- With a fast processor, it may not be necessary to implement an embedded system as a parallel process. A sequential system which uses polling to interrogate and control hardware components may provide adequate performance.
- The advantage of avoiding a parallel systems design is that sequential programs are easier to design, implement, verify and test than parallel systems. Time dependencies between processes are hard to formalize, control and verify.

Note:

- *There are some applications, such as vector processing, where a parallel approach is a completely natural one. If n-element vectors have to be processed with the same operation carried out on each element, the natural implementation is for a set of n-processes carrying out the same operation at the same time.*

Design process:

Two-stage activity:

1. Identify the logical design structure, namely the components of a system and their inter-relationships
2. Realize this structure in a form which can be executed. This latter stage is sometimes considered as detailed design and sometimes as programming.

3.1.4 Design decomposition

- The design process is influenced not only by the design approach but also by the criteria used to decompose a system.
- Numerous decomposition principles have been proposed.

Classification of decomposition methods

1. *Function-oriented decomposition.* ([Wirth 1971], [Yourdon 1979]).
   - A function-oriented system perspective forms the core of the design.
   - Based on the functional requirements contained in the requirements definition, a task-oriented decomposition of the overall system takes place.
2. *Data-oriented decomposition*. ([Jackson 1975], [Warnier 1974], [Rechenberg 1984a])
   - The design process focuses on a data-oriented system perspective.
   - The design strategy orients itself to the data to be processed.
   - The decomposition of the system stems from the analysis of the data.
3. *Object-oriented decomposition*. ([Abbott 1983], [Meyer 1988], [Wirfs-Brock 1989], [Coad 1990], [Booch 1991], [Rumbaugh 1991])
   - An object-oriented system perspective provides the focus of the design.

The user can click on the menu bar with the mouse to display all the commands belonging to a menu and can select a command, likewise with the mouse.

Classification of menu commands

- Immediately executable commands
- Commands with parameters
- Commands for switching modes

Directly executable commands including all menu commands that require no parameters or that operate on the current selection.

Example 3.2
Commands *Cut* and *Paste are* elementary operations.

With a simple mouse click the user causes the system to carry out an action that normally involves processing data.

Commands with parameters are similar in effect to those in the first class. They differ primarily in the user actions that are required to execute them.

Example 3.3
Text editor commands *Find* and *Find Next:* locate certain characters in a text.

Find has an implicit parameter, the position at which searching is to begin. The execution of the command prompts the user to input additional parameters. Input prompting is normally handled via a dialog window. The execution of such a command thus requires several sequential inputs from the user.
To simplify the repeated execution of a command with the same parameters, it can be useful to use a dedicated, immediately executable menu command (*Find Next.)*

Instead of manipulating data, the menu commands of the third class cause a change in mode that affects subsequent commands or the way in which data are displayed.

Example 3.4
Switching between *insert* and *overwrite mode* and the command *Show Controls* in a text editor to display normally invisible control characters.

- A frequently neglected task in the design of a menu system is the choice of appropriate wording for the menu commands. Apply the rule that the command should be as short as possible, yet still meaningful.

- In the design of a menu system, similar commands should be grouped together under the same menu.
- The more frequently a command is used, the higher in the menu it should be placed to avoid unnecessary mouse motion.

The basic possibilities for handling the situation where a command is invoked in a mode where it cannot be executed are:

contain control structures. The interconnection of the subsolutions can and should already be specified at this stage by means of sequential, conditional and repeated execution.

- Treat details as late as possible. This means, after the decompositions. This means, after the decomposition of a task into appropriate subtasks, waste no thought yet on what the solutions of these subtasks could look like. First clarify the interrelationship of the subtasks; only then tackle the solution of each subtask independently. This ensures that critical decisions are made only after information is available about the interrelationship of the subtasks. This makes it easier to avoid errors with grave consequences.
- A continuous reduction of complexity accompanies the design process when using the principle of stepwise refinement. For example, if a task A is solved by the sequential solution of three subtasks B, C and D, there is a temptation to assess this decomposition as trivial and of little use.
- The subtasks become increasingly concrete and their solution requires increasingly detailed information. This means stepwise refinement not only of the algorithms but also the data with which they work. The concretization of the data should also be reflected in the interfaces of the subsolutions. In the initial design step, we recommend working with abstract data structures or abstract data types whose concrete structure is defined only when the last subtasks can no longer be solved without knowledge thereof.
- During stepwise refinement the designer must constantly check whether the current decomposition can be continued in the light of further refinement or whether it must be scrapped.

- The most important decomposition decisions are made at the very start, when the designer knows the least. Hence the designer is hardly able to apply stepwise refinement consistently.
- If the design falters, the designer is tempted to save the situation by incorporating special cases into already developed subalgorithms.

3.4 Object-oriented design

[Booch 1991], [Coad 1990], [Heitz 1988], [Rumbaugh 1991], [Wasseman 1990], [Wilso 1990], [Wirfs-Brock 1989], [Wirfs-Brock 1990].

***Function-oriented design and o*bject-oriented design**

- Function-oriented design focuses on the *verbs*
- Object-oriented design focuses on the *nouns*.
- Object-oriented design requires the designer to think differently than with function-oriented design.
- Since the focus is on the data, the algorithms are not considered at first; instead the objects and the relationships between them are studied.

actually exists and whether it is used correctly (i.e. whether the number of parameters and their data types are correct).

- Languages may have independent compilation (e.g. C and FORTRAN), where this check takes place only upon invocation at run time (if at all)
- Alternatively, languages may have separate compilation (e.g. Ada and Modula-2), where each module has an interface description that provides the basis for checking its proper use already at compile time.

➢ ***Documentation value of a programming language***

- Affects the readability and thus the maintainability of programs.
- The importance of the documentation value rises for large programs and for software that the client continues to develop.
- High documentation value results, among other things, from explicit interface specifications with separate compilation (e.g. in Ada and Modula-2). Likewise the use of keywords instead of special characters (e.g. **begin** . . . **end** in Pascal rather than {. . .} in C) has a positive effect on readability because the greater redundancy gives less cause for careless errors in reading. Since programs are generally written only once but read repeatedly, the minimum additional effort in writing pays off no more so than in the maintenance phase. Likewise the language's scoping rules influence the readability of programs.
- Extensive languages with numerous specialized functions (e.g. Ada) are difficult to grasp in all their details, thus encouraging misinterpretations. Languages of medium size and complexity (e.g. Pascal and Modula-2) harbor significantly less such danger.

➢ ***Data structures in the programming language***

- Primarily when complex data must be processed, the availability of data structures in the programming language plays an important role.
- Older languages such as FORTRAN, BASIC, and COBOL offer solely the possibility to combine multiple homogeneous elements in array or heterogeneous elements in structures.
- Recursive data structures are difficult to implement in these languages.
- Languages like C permit the declaration of pointers to data structures. This enables data structures of any complexity, and their scope and structure can change at run time. However, the drawback of these data structures is that they are open and permit unrestricted access (but compare with Java [Heller 1997]).
- Primarily in large projects with multiple project teams, abstract data takes on particular meaning. Although abstract data structures can be emulated in any modular language, due to better readability, preference should be given to a language with its own elements supporting this concept.
- Object-oriented languages offer the feature of extensible abstract data types that permit the realization of complex software systems with elegance and little effort. For a flexible and extensible solution, object-oriented languages provide a particularly good option.

- For every program component, the declarations (of data types, constants, variables, etc.) should be distinctly separated form the statement section.
- The declaration sections should have a uniform structure when possible, e.g. using the following sequence: constant, data types, classes and modules, methods and procedures.
- The interface description (parameter lists for method and procedures) should separate input, output and input/output parameters.
- Keep comments and source code distinctly separate.
- The program structure should be emphasized with indentation.

4.3 Portability and reuse

The objective of this section is to describe the problems which can arise in writing portable high-level language programs and suggest how non-portable parts of a program may be isolated. The section is also concerned with software reuse. The advantages and disadvantages of reuse are discussed and guidelines given as to how reusable abstract data types can be designed.

4.3.1 Software portability

([Brown 1977], [Tanenbaum *et al.* 1978], [Wallis 1982], [Lösen 1985]).

- Can be achieved by one machine on another using microcode, compiling a program into some abstract machine language then implementing that abstract machine on a variety of computers, and using preprocessors to translate from one dialect of a programming language to another.
- A characteristic of a portable program is that it is self-contained. The program should not rely on the existence of external agents to supply required functions.
- In practice, complete self-containment is almost impossible to achieve and the programmer intending to produce a portable program must compromise by isolating necessary references to the external environment. When that external environment is changed those dependent parts of the program can be identified and modified.
- Even when a standard, widely implemented, high-level language is used for programming, it is difficult to construct a program of any size without some machine dependencies. These dependencies arise because feature of the machine and its operating system. Even the character set available on different machines may not be identical, with the result that programs written using one character set must be edited to reflect the alternative character set.
- Portability problems that arise when a standard high-level language is used can be classified under two headings:
  - problems caused by language features influenced by the machine architecture, and
  - problems caused by operating system dependencies.

- The re-use of existing software should be encouraged whenever possible as it reduces the amount of code which must be written, tested and documented. However, the use of subroutine libraries reduces the self-containedness of a program and hence may increase the difficulty of transferring that program from one installation to another.
- If use is made of standard subroutine libraries such as the NAG library, this will not cause any portability problems if the program is moved to another installation where the library is available. On the other hand, if the library is not available, transportation of the program is likely to be almost impossible.
- If use is made of local installation libraries, transporting the program either involves transporting the library with the program or supplementing the target system library to make it compatible with the host library. The user must trade off the productivity advantages of using libraries against the dependence on the external environment which this entails.
- One of the principal functions of an operating system is to provide a file system. Program access to this is via primitive operations which allow the user to name, create, access, delete, protect and share files. There are no standards governing how these operations should be provided. Each operating system supports them in different ways.
- As high-level language systems must provide file facilities, they interface with the file system. Normally, the file system operations provided in the high-level language are synonymous with the operating system primitives. Therefore, the least portable parts of a program are often those operations which involve access to files.

File system incompatibilities:

1. The convention for naming files may differ from system to system. Some systems restrict the number of characters in a file name, other systems impose restrictions on exactly which characters can make up a file name, and yet others impose no restrictions whatsoever.
2. The file system structure may differ from system to system. Some file systems are hierarchically structured. Users may create their own directories and sub-directories. Other systems are restricted to a two-level structure where all files belonging to a particular user must reside in the same directory.
3. Different systems utilize different schemes for protecting files. Some systems involve passwords, other systems use explicit lists of who may access what, and yet others grant permission according to the attributes of the user.
4. Some systems attempt to classify files as data files, program files, binary files or as files associated with the application that created them. Other systems consider all files to be untyped files of characters.
5. Most systems restrict the user to a maximum number of files which may be in use at any one time. If this number is different on the host machine from that on the target machine, there may be problems in porting programs which have many files open at the same time.

6.4 Document maintenance

As a software system is modified, the documentation associated with that system must also be modified to reflect the changes to the system.

All associated documents should be modified when a change is made to a program. Assuming that the change is transparent to the user, only documents describing the system implementation need be changed. If the system change is more than the correction of coding errors, this will mean revision of design and test documents and, perhaps, the higher level documents describing the system specification and requirements.

One of the major problems in maintaining documentation is keeping different representations of the system in step with each other. The natural tendency is to meet a deadline by modifying code with the intention of modifying other documents later. Often, pressure of work means that this modification is continually set aside until finding what is to be changed becomes very difficult indeed. The best solution to this problem is to support document maintenance with software tools which record document relationships, remind software engineers when changes to one document affect another, and record possible inconsistencies in the documentation.

If the system modification affects the user interface directly either by adding new facilities or by extending existing facilities, this should be intimated to the user immediately. In an on-line system this might be accomplished by providing a system noticeboard which each user may access. When a new item is added to the noticeboard, users can be informed of this when they log in to the system.

System changes can also be indicated on a real noticeboard and in a regular newsletter distributed to all system users. At periodic intervals, user documentation should be updated by supplying new pages which describe the changes made to the user interface.

Paragraphs which have been added or changed should be indicated to the reader.

New versions of documents should be immediately identifiable. The fact that a document has been updated should not be concealed on an inner page. Rather, the version number and date should be clearly indicated on the cover of the document and, if possible, different versions of each document should be issued with a different colour or design of cover.

6.6 Document portability

When a computing system is moved from one machine to another, the documentation associated with that system must be modified to reflect the new system. The work involved in this is comparable to the work involved in moving the programs themselves.

If portability is a system design objective, the documentation must also be designed and written with the same aim.

Program size ..... 5 : 1

Coding time..... 25 : 1

Required testing time..... 26 : 1

Required computation time..... 11 : 1

Execution time of finished program..... 13 : 1

- The time requirement for each task handled in a team consists of two basic components ([Brooks 1975]):

  (1) Productive work

  (2) Communication and mutual agreement of team members

  If no communication were necessary among team members, then the time requirement t for a project would decline with the number n of team members

  $$t \approx 1/n$$

  If each team member must exchange information with each other and that the average time for such communication is k, then the development time follows the formula:

  $$t \approx 1/n + k \cdot n^2/2$$

Important

"Adding manpower to a late software project makes it later." ([Brooks 1975])

- Most empirical values for cost estimation are in-house and unpublished. The literature gives few specifications on empirical data, and these often diverge pronouncedly. The values also depend greatly on the techniques and tools used.
- Distribution of the time invested in the individual phases of software development (including the documentation effort by share) according to the selected approach model and implementation technique ([Pomberger 1996]):

  Approach model: classical sequential software life cycle

  Implementation technique: module-oriented

  problem analysis and system specification..... 25%

  design..... 25%

  implementation..... 15%

  testing..... 35%

- The maintenance cost estimate may be refined by judging the importance of each factor which affects the cost and selecting the appropriate cost multiplier. The basic maintenance cost is then multiplied by each multiplier to give the revised cost estimate.

Example 7.2 *Say in the above system the factors having most effect on maintenance costs were reliability (RELY) which had to be very high, the availability of support staff with language and applications experience (AEXP and LEXP) which was also high, and the use of modern programming practices for system development (very high).*

From Boehm's table, we have:

RELY	1.10
AEXP	0.91
LEXP	0.95
MODP	0.72

By applying these multipliers to the initial cost estimate, a revised formula may be computed as follows:

AME = 35.4 * 1.10 * 0.91 * 0.95 * 0.72 = 24.2 person months.

- The reduction in estimated costs has come about partly because experienced staff are available for maintenance work but mostly because modern programming practices had been used during software development**.** As an illustration of their importance, the maintenance cost estimate if modern programming practices are not used at all and other factors (including the development cost!) are unchanged is as follows:

AME = 35.4 * 1.10 * 0.91 * 0.95 * 1.40 = 47.1 person-months.

- This is a gross estimate of the annual cost of maintenance for the entire software system. In fact, different parts of the system will have different ACTs so a more accurate formula can be derived by estimating initial development effort and annual change traffic for each software component. The total maintenance effort is then the sum of these individual component efforts.
- One. of the problems encountered when using an algorithmic cost estimation model for maintenance cost estimation is that it takes no account of the fact that the software structure degrades as the software ages. Using the original development time as a key factor in maintenance cost estimation introduces inaccuracies as the

software loses its resemblance to the original system. It is not clear whether this cost estimation model is valid for geriatric software systems.

- The existence of a cost estimation model which takes into account factors such as programmer experience, hardware constraints, software complexity, etc., allows decisions about maintenance to be made on a quantitative rather than a qualitative basis.

Example 7.3 *Say in the above example system that management decided that money might be saved by using less experienced staff for software maintenance. Assume that inexperienced staff cost $5000 per month compared to $6500 for more experienced software engineers.*

Using experienced staff, the total annual maintenance costs are:

AMC = 24.23 * 6500 = $157 495

Using inexperienced staff, the effort required for software maintenance is increased because the staff experience multipliers change:

AME = 35.24 * 1.10 * 1.07 * 1.13 * 0.72 = 33.89 person-months.

Thus, total costs using inexperienced staff are:

AMC = 33.89 * 5000 = $169 450

Therefore, it appears to be more expensive in this example to use inexperienced staff rather than experienced engineers.

Measuring program maintainability

- Maintainability metrics are based on the assumption that the maintainability of a program is related to its complexity.
- The metrics measure some aspects of the program complexity.
- It is suggested that high complexity values correlate with difficulties in maintaining a system component.
- The complexity of a program can be measured by considering ([Halstead 1977])
    - the number of unique operators,
    - the number of unique operands,
    - the total frequency of operators, and
    - the total frequency of operands in a program.

decisions made by maintenance management are overwhelmed by it. This law is a result of fundamental structural and organizational effects.

- The fourth law suggests that most large programming projects work in what he terms a 'saturated' state. That is, a change of resources or staffing has imperceptible effects on the long-term evolution of the system.
- The fifth law is concerned with the change increments in each system release.
- Lehman's laws are really hypotheses and it is unfortunate that more work has not been carried out to validate them. Nevertheless, they do seem to be sensible and maintenance management should not attempt to circumvent them but should use them as a basis for planning the maintenance process. It may be that business considerations require them to be ignored at any one time (say it is necessary to make several major system changes). In itself, this is not impossible but management should realize the likely consequences for future system change.

Preface

In most software-development organizations, the testing program functions as the final "quality gate" for an application, allowing or preventing the move from the comfort of the software-engineering environment into the real world. With this role comes a large responsibility: The success of an application, and possibly of the organization, can rest on the quality of the software product.

A multitude of small tasks must be performed and managed by the testing team—so many, in fact, that it is tempting to focus purely on the mechanics of testing a software application and pay little attention to the surrounding tasks required of a testing program. Issues such as the acquisition of proper test data, testability of the application's requirements and architecture, appropriate test-procedure standards and documentation, and hardware and facilities are often addressed very late, if at all, in a project's life cycle. For projects of any significant size, test scripts and tools alone will not suffice—a fact to which most experienced software testers will attest.

Knowledge of what constitutes a successful end-to-end testing effort is typically gained through experience. The realization that a testing program could have been much more effective had certain tasks been performed earlier in the project life cycle is a valuable lesson. Of course, at that point, it's usually too late for the current project to benefit from the experience.

Effective Software Testing provides experience-based practices and key concepts that can be used by an organization to implement a successful and efficient testing program. The goal is to provide a distilled collection of techniques and discussions that can be directly applied by software personnel to improve their products and avoid costly mistakes and oversights. This book details 50 specific software testing best practices, contained in ten parts that roughly follow the software life cycle. This structure itself illustrates a key concept in software testing: To be most effective, the testing effort must be integrated into the software-development process as a whole. Isolating the testing effort into one box in the "work flow" (at the end of the software life cycle) is a common mistake that must be avoided.

The material in the book ranges from process- and management-related topics, such as managing changing requirements and the makeup of the testing team, to technical aspects such as ways to improve the testability of the system and the integration of unit testing into the development process. Although some

Acknowledgments

My thanks to all of the software professionals who helped support the development of this book, including students attending my tutorials on Automated Software Testing, Quality Web Systems, and Effective Test Management; my co-workers on various testing efforts at various companies; and the co-authors of my various writings. Their valuable questions, insights, feedback, and suggestions have directly and indirectly added value to the content of this book. I especially thank Douglas McDiarmid for his valuable contributions to this effort. His input has greatly added to the content, presentation, and overall quality of the material.

My thanks also to the following individuals, whose feedback was invaluable: Joe Strazzere, Gerald Harrington, Karl Wiegers, Ross Collard, Bob Binder, Wayne Pagot, Bruce Katz, Larry Fellows, Steve Paulovich, and Tim Van Tongeren.

I want to thank the executives at Addison-Wesley for their support of this project, especially Debbie Lafferty, Mike Hendrickson, John Fuller, Chris Guzikowski, and Elizabeth Ryan.

Last but not least, my thanks to Eric Brown, who designed the interesting book cover.

Elfriede Dustin

Chapter 1. Requirements Phase

The most effective testing programs start at the beginning of a project, long before any program code has been written. The requirements documentation is verified first; then, in the later stages of the project, testing can concentrate on ensuring the quality of the application code. Expensive reworking is minimized by eliminating requirements-related defects early in the project's life, prior to detailed design or coding work.

The requirements specifications for a software application or system must ultimately describe its functionality in great detail. One of the most challenging aspects of requirements development is communicating with the people who are supplying the requirements. Each requirement should be stated precisely and clearly, so it can be understood in the same way by everyone who reads it.

If there is a consistent way of documenting requirements, it is possible for the stakeholders responsible for requirements gathering to effectively participate in the requirements process. As soon as a requirement is made visible, it can be **tested** and clarified by asking the stakeholders detailed questions. A variety of **requirement tests** can be applied to ensure that each requirement is relevant, and that everyone has the same understanding of its meaning.

Item 4: Ensure That Requirement Changes Are Communicated

When test procedures are based on requirements, it is important to keep test team members informed of changes to the requirements as they occur. This may seem obvious, but it is surprising how often test procedures are executed that differ from an application's implementation that has been changed due to updated requirements. Many times, testers responsible for developing and executing the test procedures are not notified of requirements changes, which can result in false reports of defects, and loss of required research and valuable time.

There can be several reasons for this kind of process breakdown, such as:

- *Undocumented changes.* Someone, for example the product or project manager, the customer, or a requirements analyst, has instructed the developer to implement a feature change, without agreement from other stakeholders, and the developer has implemented the change without communicating or documenting it. A process needs to be in place that makes it clear to the developer how and when requirements can be changed. This is commonly handled through a **Change Control Board,** an **Engineering Review Board,** or some similar mechanism, discussed below.
- *Outdated requirement documentation.* An oversight on the testers' part or poor configuration management may cause a tester to work with an outdated version of the requirement documentation when developing the test plan or procedures. Updates to requirements need to be documented, placed under configuration management control (**baselined**), and communicated to all stakeholders involved.
- *Software defects.* The developer may have implemented a requirement incorrectly, although the requirement documentation and the test documentation are correct.

data entry, for example, or business rules that could corrupt data or result in violation of regulations.

- *Operational characteristics.* Some test requirements will rank high on the priority list because they apply to frequently-used functions or are based upon a lack of knowledge of the user in the area. Functions pertaining to technical resources or internal users, and those that are infrequently used, are ranked lower in priority.
- *User requirements.* Some test requirements are vital to user acceptance. If the test approach does not emphasize the verification of these requirements, the resulting product may violate contractual obligations or expose the company to financial loss. It is important that the impact upon the end user of any potential problem be assessed.
- *Available resources.* A factor in the prioritization of test requirements is the availability of resources. As previously discussed, the test program must be designed in the context of constraints including limited staff availability, limited hardware availability, and conflicting project requirements. Here is where the painful process of weighing trade-offs is performed.

Most risk is caused by a few factors:

- *Short time-to-market.* A short time-to-market schedule for the software product makes availability of engineering resources all the more important. As previously mentioned, testing budgets and schedules are often determined at the onset of a project, during proposal development, without inputs from testing personnel, reference to past experience, or other effective estimation techniques. A good test manager can quickly ascertain when a short-time-to-market schedule would prevent adequate testing. Test strategies must be adapted to fit the time available. It is imperative that this issue be pointed out immediately so schedules can be adjusted, or the risks of fast development can be identified and risk-mitigation strategies developed.
- *New design processes.* Introduction of new design processes, including new design tools and techniques, increases risk.
- *New technology.* If new technology is implemented, there may be a significant risk that the technology will not work as expected, will be misunderstood and implemented incorrectly, or will require patches.
- *Complexity.* Analyses should be performed to determine which functionality is most complex and error-prone and where failure would have high impact. Test-team resources should be focused on these areas.

Item 11: Plan the Test Environment

The **test environment** comprises all of the elements that support the physical testing effort, such as test data, hardware, software, networks, and facilities. Test-environment plans must identify the number and types of individuals who require access to the test environment, and specify a sufficient number of computers to accommodate these individuals. (For a discussion of test-team membership, see Chapter 3.) Consideration should be given to the number and kinds of environment-setup scripts and test-bed scripts that will be required.

In this chapter, the term **production environment** refers to the environment in which the final software will run. This could range from a single end-user computer to a network of computers connected to the Internet and serving a complete Web site.

While unit- and integration-level tests are usually performed within the development environment by the development staff, system tests and user-acceptance tests are ideally performed within a separate test-lab setting that represents a configuration identical to the production environment, or at least a scaled-down version of the production environment. The test-environment configuration must be representative of the production environment because the test environment must be able to replicate the baseline configuration of the production environment in order to uncover any configuration-related issues that may affect the application, such as software incompatibilities, clustering, and firewall issues. However, fully replicating the production environment is often not feasible, due to cost and resource constraints.

After gathering and documenting the facts as described above, the test team must compile the following information and resources preparatory to designing a test environment:

- Obtain descriptions of sample customer environments, including a listing of support software, COTS (commercial off-the-shelf) tools, computer hardware and operating systems. Hardware descriptions should include such elements as video resolution, hard-disk space, processing speed, and memory characteristics, as well as printer characteristics including type of printer, capacity, and whether the printer is dedicated to the user's machine or connected to a network server.
- Determine whether the test environment requires an archive mechanism, such as a tape drive or recordable CD (CD-R) drive, to allow the storage of

number of test procedures and number of tester hours expended, taking into account experience from similar historical projects. The result is then used to estimate the number of personnel hours (or full time equivalent personnel) needed to support the test effort on the new project.

For this estimation method to be most successful, the projects being compared must be similar in nature, technology, required expertise, problems solved, and other factors, as described in the section titled "Other Considerations" later in this Item.

Table 12.3 shows example figures derived using the Test Procedure Method, where a test team has estimated that a new project will require 1,120 test procedures. The test team reviews historical records of test efforts on two or more similar projects, which on average involved 860 test procedures and required 5,300 personnel-hours for testing. In these previous test efforts, the number hours per test procedure was approximately 6.16 over the entire life cycle of testing activities, from startup and planning to design and development to test execution and reporting. The 5,300 hours were expended over an average nine-month period, representing 3.4 full-time-equivalent test engineers for the project. For the new project, the team plans to develop 1,120 test procedures.

Table 12.3. Test Team Size Calculated Using the Test-Procedure Method

	Number of Test Procedures	Factor	Number of Person Hours	Performance Period	Number of Testers
Historical Record (Average of Two or More Similar Projects)	860	6.16	5,300	9 months (1,560 hrs)	3.4
New Project Estimate	1,120	6.16	6,900	12 months (2,080 hrs)	3.3

The factor derived using the Test Procedure Method is most reliable when the historical values are derived from projects undertaken after the testing culture of the organization has reached maturity.

Item 13: Define Roles and Responsibilities[1]

[1] Adapted from Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), Table 5.11, 183–186.

Test efforts are complex, and require that the test team possess a diversity of expertise to comprehend the scope and depth of the required test effort and develop a strategy for the test program.

In order for everyone on the test team to be aware of what needs to get done and who will take the lead on each task, it is necessary to define and document the roles and responsibilities of the test-team members. These should be communicated, both verbally and in writing, to everyone on the team. Identifying the assigned roles of all test-team members on the project enables everyone to clearly understand which individual is responsible for each area of the project. In particular, it allows new team members to quickly determine whom to contact if an issue arises.

In order to identify the individuals needed to perform a particular task, a task description should be created. Once the scope of the task is understood, it will be easier to assign particular team members to the task.

To help ensure successful execution of the task, **work packages** can be developed and distributed to the members of the test team. Work packages typically include the organization of the tasks, technical approach, task schedule, spending plan, allocation of hours for each individual, and a list of applicable standards and processes.

The number of test-engineering roles in a project may be greater than the number of test-team members. (The roles required depend on the task at hand, as discussed in Chapter 2.) As a result, a test engineer may "wear many hats," being responsible for more than one role.

Table 13.1 shows some example responsibilities and skills required for each test-program role.

Table 13.2. Example Test-Team Assignments

Position	Products	Duties / Skills	Roles and Responsibilities
Test Manager	Desktop Web	Responsible for test program, customer interface, test-tool introduction, and staff recruiting and supervision Skills: Management skills, MS Project, Winrunner, SQL, SQL Server, UNIX, VC++, Web applications, test-tool experience	Manage test program
Test Lead	Desktop Web	Staff supervision, cost/progress/test status reporting, and test planning, design, development, and execution Skills: TeamTest, Purify, Visual Basic, SQL, Winrunner, Robot, UNIX, MS Access, C/C++, SQL Server	[Reference the related testing requirements here] Develop automated test scripts for functional test procedures
Test Engineer	Desktop Web	Test planning, design, development, and execution Defect identification and tracking Skills: Test-tool experience, financial system experience	[Reference the related testing requirements here] Develop test harness
Test Engineer	Desktop Web	Test planning, design, development, and execution Defect identification and tracking Skills: Test-tool experience, financial system experience	Performance testing [Reference the related testing requirements here]
Test Engineer	Desktop	Test planning, design, development, and execution Defect identification and tracking Skills: Test-tool experience, financial system experience	Configuration testing, installation testing [Reference the related testing requirements here]
Test Engineer	Web	Responsible for test tool environment, network, and middleware testing Performs all other test activities Defect identification and tracking Skills: Visual Basic, SQL, CNE, UNIX, C/C++, SQL Server	Security testing [Reference the related testing requirements here]
Jr. Test Engineer	Desktop	Performs test planning, design, development, and execution Defect identification and tracking Skills: Visual Basic, SQL, UNIX, C/C++, HTML, MS Access	[Reference the related testing requirements here]

Table 13.2 identifies test-team positions and their assignments on the project, together with the products they are working on. The duties that must be performed by the person in each of the positions are outlined, as are the skills of the personnel

```
Message:   successfully connected to database
           [dbserver1, customer_db]

Function:  retrieveCustomer (customer.cpp line 20)
Machine:   testsrvr (PID=2201)
Timestamp: 1/10/2002 20:26:56.568
Message:   attempting to retrieve customer record
           for customer ID [A1000723]

Function:  retrieveCustomer (customer.cpp line 25)
Machine:   testsrvr (PID=2201)
Timestamp: 1/10/2002 20:26:57.12
Message:   ERROR: failed to retrieve customer record,
           message [customer record for ID A1000723
           not found]
```

This log-file excerpt demonstrates a few of the major aspects of application logging that can be used for effective testing.

In each entry, the function name is indicated along with the file name and the line number of the application source code that generated the entry. The host and process ID are also recorded, as well as the time when the entry was made. Each message contains useful information about the identities of components involved in the activity. For example, the database server is "dbserver1," the database is "customer_db," and the customer ID is "A1000723."

From this log, it is evident that the application was not able to successfully retrieve the specified customer record. In this situation, a tester could examine the database on dbserver1 and, using SQL tools, query the customer_db database for the customer record with ID A1000723 to verify its absence.

This information adds a substantial amount of defect-diagnosis capability to the testing effort, since the tester can now pass such detailed information along to the development staff as part of the defect report. The tester can report not only a "symptom"but also internal application behavior that pinpoints the cause of the problem.

methodology and standards to be followed; and the testing schedule. If a usable test plan (as discussed in Chapter 2) does not already exist, this information must be gathered from other sources.

To break down the testing tasks, the following "what," "when," "how," and "who" questions should be answered.

- ***What** should be tested?* During the test-planning phase, what to test and what not to test will have been determined and documented as part of the scope of testing.
- ***When** should test procedures be developed?* In Item 3 we suggest that test procedures be developed as soon as requirements are available. Once it has been determined *what* to test, the sequence of tests must be established. What needs to be tested first? The test planner should get to know the testing priorities, and should become familiar with the build and release schedule. Procedures for testing the high-priority items should be developed first. One exception: Certain functions may need to be run first to "prepare" the system for other functions. These **precursor functions** must be run early, whether they are high priority or not. (For more on prioritizing features, see Item 8.)

  Additionally, risk analysis (see Item 7) should be employed to help prioritize test procedures. If it is not possible to test everything, testers are forced to focus on the most critical elements. Risk analysis provides a mechanism for determining which these are.

- ***How** should test procedures be designed?* No single testing solution can effectively cover all parts of a system. Test procedures for the different parts of the system must be designed in the manner most appropriate for effectively testing each of those specific parts.

  In order to design the appropriate and most effective tests, it is necessary to consider the parts that make up the system and how they are integrated. For example, to verify functional behavior of the system via the user interface, test procedures will most likely be based on existing functional-requirements statements, with test cases that execute the various paths and scenarios. Another approach would be to begin by testing each field in the user interface with representative valid and invalid data, verifying the correct behavior for each input. This would involve following a sequence of execution paths, as, for example, when filling one field or screen produces another GUI screen that also requires data input.

Effective test design includes test procedures that rarely overlap, but instead provide effective coverage with minimal duplication of effort (although duplication sometimes cannot be entirely avoided in assuring complete testing coverage). It is not effective for two test engineers to test the same functionality in two different test procedures, unless this is necessary in order to get the required functional path coverage (as when two paths use duplicate steps at some points).

It is important to analyze test flow to ensure that, during test execution, tests run in proper order, efforts are not unnecessarily duplicated, testers don't invalidate one another's test results, and time is not wasted by producing duplicate or erroneous findings of defects. Such findings can be time consuming for developers to research and for testers to retest, and can skew the defect metrics if not tracked correctly. The test team should review the test plan and design in order to:

- Identify any patterns of similar actions or events used by several transactions. Given this information, test procedures should be developed in a modular fashion so they can be reused and recombined to execute various functional paths, avoiding duplication of test-creation efforts.
- Determine the order or sequence in which specific transactions must be tested to accommodate preconditions necessary to execute a test procedure, such as database configuration, or other requirements that result from control or work flow.
- Create a test procedure relationship matrix that incorporates the flow of the test procedures based on preconditions and postconditions necessary to execute a procedure. A test-procedure relationship diagram that shows the interactions of the various test procedures, such as the high-level test procedure relationship diagram created during test design, can improve the testing effort.

The analyses above help the test team determine the proper sequence of test design and development, so that modular test procedures can be properly linked together and executed in a specific order that ensures contiguous and effective testing.

Another consideration for effectively creating test procedures is to determine and review critical and high-risk requirements, in order to place a greater priority upon, and provide added depth for, testing the most important functions early in the development schedule. It can be a waste of time to invest efforts in creating test procedures that verify functionality rarely executed by the user, while failing to create test procedures for functions that pose high risk or are executed most often.

Item 25: Use Proven Testing Techniques when Designing Test-Case Scenarios

Item 10 discusses the importance of planning test data in advance. During the test-design phase, it will become obvious that the combinations and variations of test data that may be used as input to the test procedures can be endless. Since exhaustive testing is usually not possible, it is necessary to use testing techniques that narrow down the number of test cases and scenarios in an effective way, allowing the broadest testing coverage with the least effort. In devising such tests, it's important to understand the available test techniques.

Many books address the various white-box and black-box techniques.[1] While test techniques have been documented in great detail, very few test engineers use a structured test-*design* technique. An understanding of the most widely used test techniques is necessary during test design.

> [1] For example: Boris Beizer, *Software Testing Techniques* (Hoboken, N.J.: John Wiley & Sons, 1995).

Using a combination of available testing techniques has proven to be more effective than focusing on just one technique. When systems professionals are asked to identify an adequate set of test cases for a program they are testing, they are likely to identify, on average, only about half of the test cases needed for an adequate testing effort. When testers use guesswork to select test cases to execute, there is a high potential for unreliability, including inadequate test coverage.

Among the numerous testing techniques available to narrow down the set of test cases are functional analysis, equivalence partitioning, path analysis, boundary-value analysis, and orthogonal array testing. Here are a few points about each:

- *Functional analysis* is discussed in detail in Item 22. It involves analyzing the expected behavior of the system according to the functional specifications and generating one test procedure or more for each function or feature of the system. If the requirement is that the system provides function x, then the test case(s) must verify that the system provides function x in an adequate manner. One way of conducting functional test analyses is discussed in Item 22. After the functional tests have been defined and numerous testing paths through the application have been derived, additional techniques must be applied to narrow down the inputs for the functional steps to be executed during testing.

As an example, consider an application that checks an input to ensure that it is greater than 10.

- An in-bounds value would be 13, which is greater than 10.
- An out-of-bounds value would be 5, which is not greater than 10.
- The value of 10 is actually out-of-bounds, because it is not greater than 10.

In addition to values that lie in or on the boundary, such as endpoints, BV testing uses maximum/minimum values, or more than maximum or minimum, and one less than maximum and minimum, or zero and null inputs. For example, when defining the test-input values for a numeric input, one could consider the following:

- Does the field accept numeric values only, as specified, or does it accept alphabetic values?
- What happens if alphabetic values are entered? Does the system accept them? If so, does the system produce an error message?
- What happens if the input field accepts characters that are reserved by the application or by a particular technology, for example special characters such as ampersands in Web applications? Does the application crash when the user inputs these reserved characters?

The system should either not allow out-of-bounds characters to be entered, or instead should handle them gracefully by displaying an appropriate error message.

- *Orthogonal arrays* allow maximum test coverage from a minimum set of test procedures. They are useful when the amount of potential input data, or combinations of that input data, may be very large, since it is usually not feasible to create test procedures for every possible combination of inputs.[2]

  [2] 8. For more on orthogonal arrays, see Elfriede Dustin, "Orthogonally Speaking," *STQE Magazine* 3:5 (Sept.-Oct. 2001). Also available at http://www.effectivesoftwaretesting.com.

  The concept of orthogonal arrays is best presented with an example. Suppose there are three parameters (A, B, and C), each of which has one of three possible values (1, 2, or 3). Testing all possible combinations of the three pa rameters would require twenty-seven test cases (33). Are all twenty-

In addition to writing unit-test programs, the developer also must examine code and components with other tools, such as memory-checking software to find memory leaks. Having several developers examine the source code and unit-test results may increase the effectiveness of the unit-testing process.

In addition to writing the initial unit test, the developer of the component is in a good position to update the unit test as modifications are made to the code. These modifications could be in response to general improvements and restructuring, a defect, or a requirement change. Making the developer who is responsible for the code also responsible for the unit test is an efficient way to keep unit tests up to date and useful.

Depending on how unit tests are implemented, they could cause the build to halt—making it fail to compile or produce a working executable—if the unit-test program is part of the software build. For example, suppose a developer removes a function, or **method** from a component's C++ interface. If a unit test has not been updated and still requires the presence of this function to compile properly, it will fail to compile. This prevents continuing on to build other components of the system until the unit test is updated. To remedy the problem, the developer must adjust the unit-test program's code to account for the removal of the method from the interface. This example shows why it is important for the developer to perform any necessary updates to the unit test program whenever the code is changed.

Some software projects also require successful unit-test *execution*, not just compilation, for the build to be considered successful. See Item 30 for a discussion of this topic.

Unit tests must be written in an appropriate language capable of testing the code or component in question. For example, if the developer has written a set of pure C++ classes to solve a particular problem or need, the unit test most likely must also be written in C++ in order to exercise the classes. Other types of code, such as COM objects, could be tested using tests written in Visual Basic or possibly with scripts, such as VBScript, JScript, or Perl.

In a large system, code is usually developed in a modular fashion by dividing functionality into several **layers,** each responsible for a certain aspect of the system. For example, a system could be implemented in the following layers:

- **Database abstraction.** An abstraction for database operations **wraps up**[1] database interaction into a set of classes or components (depending on the

successfully unit-tested system. The software is always in a testable state, and does not contain major errors in the components that can be caught by the unit tests.

A major issue in unit testing is inconsistency. Many software engineers fail to employ a uniform, structured approach to unit testing. Standardizing and streamlining unit tests can reduce their development time and avoid differences in the way they are used. This is especially important if they are part of the build process, since it is easier to manage unit-test programs if they all behave the same way. For example, unit-test behavior when encountering errors or processing command-line arguments should be predictable. Employing standards for unit tests, such as that unit-test programs all return *zero* for success and *one* for failure, leads to results that can be picked up by the build environment and used as a basis for deciding whether the build should continue. If no standard is in place, different developers will probably use different return values, thus complicating the situation.

One way to achieve such standardization is to create a **unit-test framework.** This is a system that handles processing of command-line arguments (if any) and reporting of errors. Typically, a framework is configured at startup with a list of tests to run, and then calls them in sequence. For example:


```
Framework.AddTest(CreateOrderTest)
Framework.AddTest(CreateCustomerTest)
Framework.AddTest(CreateItemTest)
```

Each test (i.e., CreateOrderTest, CreateCustomerTest, and CreateItemTest) is a function somewhere in the unit-test program. The framework executes all of these tests by calling these functions one by one, and handles any errors they report, as well as returning the result of the unit test as whole, usually **pass** or **fail.** A framework can reduce unit-test development time, since only the individual tests need be written and maintained in each layer, not all of the supporting error-handling and other execution logic. The common unit-test functions are written only one time, in the framework itself. Each unit-test program simply implements the test functions, deferring to the framework code for all other functionality, such as error handling and command-line processing.

Since unit-test programs are directly related to the source code they test, each should reside in the project or workspace of its related source code. This allows for effective configuration management of the unit tests with the components being tested, avoiding "out-of-sync" problems. The unit tests are so dependent upon the

available tools, listed in Table 31.1, that support the various testing phases. Although other tools, such as defect-tracking tools and configuration-management tools, are also used in most software projects, the table lists only tools specific to test automation.

All of the tools listed in Table 31.1 may be valuable for improving the testing life cycle. However, before an organization can decide which tools to purchase, an analysis must be conducted to determine which of the tools, if any, will be most beneficial for a particular testing process. The capabilities and drawbacks of a tool are examined by comparing the current issues to be solved with a target solution, evaluating the potential for improvement, and conducting a cost/benefit analysis. Before purchasing any tool to support a software engineering activity, such an evaluation should be performed, similar to the automated test tool evaluation process described in Item 34.

Table 31.1. Test Tools

Type of Tool	Description
Test-Procedure Generators	Generate test procedures from requirements/design/object models
Code (Test) Coverage Analyzers and Code Instrumentors	Identify untested code and support dynamic testing
Memory-Leak Detection	Verify that an application is properly managing its memory resources
Metrics-Reporting Tools	Read source code and display metrics information, such as complexity of data flow, data structure, and control flow. Can provide metrics about code size in terms of numbers of modules, operands, operators, and lines of code.
Usability-Measurement Tools	User profiling, task analysis, prototyping, and user walk-throughs
Test-Data Generators	Generate test data
Test-Management Tools	Provide such test-management functions as test-procedure documentation and storage and traceability
Network-Testing Tools	Monitoring, measuring, testing, and diagnosing performance across entire network
GUI-Testing Tools	Automate GUI tests by recording user interactions

(Capture/Playback)	with online systems, so they may be replayed automatically
Load, Performance, and Stress Testing Tools	Load/performance and stress testing
Specialized Tools	Architecture-specific tools that provide specialized testing of specific architectures or technologies, such as embedded systems

Following are some key points regarding the various types of testing tools.

- *Test-procedure generators.* A requirements-management tool may be coupled with a specification-based test-procedure (case) generator. The requirements-management tool is used to capture requirements information, which is then processed by the test-procedure generator. The generator creates test procedures by statistical, algorithmic, or heuristic means. In statistical test-procedure generation, the tool chooses input structures and values in a statistically random distribution or a distribution that matches the usage profile of the software under test.

  Most often, test-procedure generators employ action, data, logic, event, and state-driven strategies. Each of those strategies is employed to probe for a different kind of software defect. When generating test procedures by heuristic or failure-directed means, the tool uses information provided by the test engineer. Failures the test engineer has discovered frequently in the past are entered into the tool. The tool then becomes knowledge-based, using the knowledge of historical failures to generate test procedures.

- *Code-coverage analyzers and code instrumentors* . Measuring structural coverage enables the development and test teams to gain insight into the effectiveness of tests and test suites. Tools in this category can quantify the complexity of the design, measure the number of integration tests required to qualify the design, help produce the integration tests, and measure the number of integration tests that have not been executed. Other tools measure multiple levels of test coverage, including segment, branch, and conditional coverage. The appropriate level of test coverage depends upon the criticality of a particular application.

For example, an entire test suite can be run through a code-coverage tool to measure branch coverage. The missing coverage of branches and logic can then be added to the test suite.

- *Memory-leak detection tools* . Tools in this category are used for a specific purpose: to verify that an application is properly using its memory resources. These tools ascertain whether an application is failing to release memory allocated to it, and provide runtime error detection. Since memory issues are involved in many program defects, including performance problems, it is worthwhile to test an application's memory usage frequently.
- *Usability-measurement tools* . Usability engineering is a wide-ranging discipline that includes user-interface design, graphics design, ergonomic concerns, human factors, ethnography, and industrial and cognitive psychology. Usability testing is largely a manual process of determining the ease of use and other characteristics of a system's interface. However, some automated tools can assist with this process, although they should never replace human verification of the interface.[2]

[2] Elfriede Dustin et al., "Usability," Chapter 7.5 in *Quality Web Systems: Performance, Security, and Usability* (Boston, Mass.: Addison-Wesley, 2002).

- *Test-data generators* . Test-data generators aid the testing process by automatically generating the test data. Many tools on the market support the generation of test data and populating of databases. Test-data generators can populate a database quickly based on a set of rules, whether data is needed for functional testing, data-driven load testing, or performance and stress testing.
- *Test-management tools* . Test-management tools support the planning, management, and analysis of all aspects of the testing life cycle. Some test-management tools, such as Rational's TestStudio, are integrated with requirement and configuration management and defect tracking tools, in order to simplify the entire testing life cycle.
- *Network-testing tools* . The popularity of applications operating in client-server or Web environments introduces new complexity to the testing effort. The test engineer no longer exercises a single, closed application operating on a single system, as in the past. Client-server architecture involves three separate components: the server, the client, and the network. Inter-platform connectivity increases potential for errors. As a result, the testing process must cover the performance of the server and the network, the overall

compatible with the application under test and devoting further resources to creating a work-around solution, it may be more beneficial to create a home-grown set of testing scripts or other custom tool.

- *Specialized testing needs* . For the most efficient testing, specialized, automated testing scripts are often required to augment formal, vendor-provided tool-based testing. Often a test harness must be developed as an enhancement to the GUI testing tool, to cover the automated testing for a complex, critical component that the GUI testing tool cannot reach.

If the decision is made to build a tool, there are important steps to follow. Assuming the task at hand is understood and the tests lend themselves to this type of automation, these steps include:

- Determine the resources, budgets, and schedules required for building the testing tool well in advance.
- Get buy-in and approval from management for this effort.
- Treat the development of the testing tool as a part of the software development effort.
- Manage the tool's source code in version control with the rest of system. If the tool isn't versioned, it will easily fall out of sync with the software and cease to function properly.
- Treat the development of the testing tool as a main objective. When building a tool is treated as a side project, it seldom is pursued with all of the best development practices that are important for producing a solid piece of software—and the tool itself may contain bugs or be difficult to implement and maintain.
- As with any piece of code, test the home-grown testing tool itself to verify that it works according to its requirements. It is critical that a testing tool not produce false negatives or false positives.

The process of building a tool can range from writing a simple batch file or Perl script to creating a complex C++ application. The appropriate language with which to build the tool depends on the task at hand and the function of the test. For example, if the function of the test is to thoroughly exercise a complex C++ calculation DLL using some or all of its possible inputs, a suitable solution may be another C++ program that directly calls the DLL, supplying the necessary combinations of test values and examining the results.

In addition to exploring testing tools on the market and considering building custom tools, it may also be worthwhile to investigate the many free or shareware

Surprising as it may seem, there is a good chance that the test effort will initially *increase* when an automated test tool is first brought in. Introducing an automated test tool to a new project adds a whole new level of complexity to the test program. And, in addition to accounting for the learning curve for the test engineers to become proficient in the use of the automated tool, managers must not forget that no tool will eliminate *all* need for manual testing in a project.

- *Test schedules do not decrease.* A related misconception about automated testing is that the introduction of an automated testing tool on a new project will immediately reduce the test schedule. Since the testing effort actually increases when an automated test tool is initially introduced, the testing schedule cannot be expected to decrease at first; rather, allowance must be made for schedule increases. After all, the current testing process must be augmented, or an entirely new testing process must be developed and implemented, to allow for the use of automated testing tools. An automated testing tool will provide additional testing coverage, but it will not generate immediate schedule reductions.
- *Automated testing follows the software development life cycle.* Initial introduction of automated testing requires careful analysis of the application under test to determine which sections of the application can be automated. It also requires careful attention to procedure design and development. The automated test effort can be viewed as having its own mini-development life cycle, complete with the planning and coordination issues attendant to any development effort.
- *A somewhat stable application is required.* An application must be somewhat stable in order to automate it effectively using a capture/playback tool. Often, it is infeasible or not possible for maintenance reasons, to automate against portions of the software that keep changing. Sometimes automated tests cannot be executed in their entirety, but must be executed only partway through, because of problems with the application.
- *Not all tests should be automated.* As previously mentioned, automated testing is an enhancement to manual testing, but it can't be expected that all tests on a project can be automated. It is important to analyze which tests lend themselves to automation. Some tests are impossible to automate, such as verifying a printout. The test engineer can automatically send a document to the printer —a message can even pop up that says, "printed successfully"—but the tester must verify the results by physically walking over to the printer to make sure the document really printed. (The printer could have been off line or out of paper. The printout could be misaligned,

being used, so scripts have to be repeatedly re-created, causing much wasted effort. Early training in use of the tool would eliminate much of this work.

- *Testing tools can be intrusive.* Some testing tools are intrusive; for the automated tool to work correctly, it may be necessary to insert special code into the application to integrate with the testing tool. Development engineers may be reluctant to incorporate this extra code. They may fear it will cause the system to operate improperly or require complicated adjustments to make it work properly.

  To avoid such conflicts, the test engineers should involve the development staff in selecting an automated tool. If the tool requires code additions (not all tools do), developers need to know that well in advance. To help reassure developers that the tool will not cause problems, they can be offered feedback from other companies that have experience using the tool, and can be shown documented vendor claims to that effect.

  Intrusive tools pose the risk that defects introduced by the testing hooks (code inserted specifically to facilitate testing) and instrumentation could interfere with the normal functioning of the system. Regression tests on the production-ready, cleaned-up code may be required to ensure that there are no tool-related defects.

- *Testing tools can be unpredictable.* As with all technologies, testing tools can be unpredictable. For example, repositories may become corrupt, baselines may not be restored, or tools may not always behave as expected. Often, much time must be spent tracking down the problem or restoring a back-up of a corrupted repository. Testing tools are also complex applications in themselves, so they may have defects that interfere with the testing effort and may require vendor-provided patches. All of these factors can consume additional time during automated testing.
- *Automaters may lose sight of the testing goal.* Often, when a new tool is used for the first time in a testing program, more time is spent on automating test scripts than on actual testing. Test engineers may become eager to develop elaborate automation scripts, losing sight of the real goal: to test the application. They must keep in mind that automating test scripts is *part* of the testing effort, but doesn't replace it. Not everything can or should be automated. As previously mentioned, it's important to evaluate which tests lend themselves to automation.

When planning an automated testing process, it's important to clearly define the division of duties. It's not necessary for the entire testing team to spend its time automating scripts; only some of the test engineers should spend their time automating scripts. The engineers selected for this work should have backgrounds in software development.

Item 34: Focus on the Needs of Your Organization

Anyone participating in test engineer user-group discussions[1] will frequently encounter the following questions: "Which testing tool is the best on the market? Which do you recommend?"

> [1] Two good examples of such discussions are the Web site http://www.qaforums.com and the Usenet newsgroup comp.software.testing.

Users will respond with as many different opinions as there are contributors to the testing forum. Often a user most experienced with a particular tool will argue that that specific tool is the best solution.

However, the most useful answer to this popular question is: "It depends." Which testing tool is best depends on the needs of the organization and the system-engineering environment—as well as the testing methodology, which will, in part, dictate how automation fits into the testing effort.

Following is a list of best practices to consider when choosing a testing tool:[2]

> [2] For additional information on tool evaluation, see Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), 67–103.

- *Decide on the type of testing life-cycle tool needed.* If the automation task is an organization-wide effort, gather input from all stakeholders. What do they want the automation to accomplish? For example, it may be that in-house users of the system under test would like to use the tool for user-acceptance testing. Determine what is expected from automation, so those expectations can be managed early on, as discussed in Item 33.

  Sometimes a test manager is instructed to find a tool that supports most of the organization's testing requirements, if feasible. Such a decision requires considering the systems-engineering environment and other organizational

- *Know the types of tests to be developed.* Since there are many types of test phases for any given project, it is necessary to select the types of testing of interest. The test strategy should be defined at this point so the test team can review the types of tests needed—regression testing, stress or volume testing, usability testing, and so forth. Questions to ask to help determine types of tests to employ include: What is the most important feature needed in a tool? Will the tool be used mainly for stress testing? Some test tools specialize in **source code coverage analysis.** That is, they identify all of the possible source-code paths that must be verified through testing. Is this capability required for the particular project or set of projects? Other test-tool applications to consider include those that support process automation and bulk data loading through input files. Consider what the test team hopes to accomplish with the test tool. What is the goal? What functionality is desired?
- *Know the schedule.* Another concern when selecting a test tool is its fit with and impact upon the project schedule. It is important to review whether there will be enough time for the necessary testers to learn the tool within the constraints of the schedule. When there is not enough time in the project schedule, it may be advisable not to introduce an automated test tool. By postponing the introduction of a test tool to a more opportune time, the test team may avoid the risk of rushing the introduction, perhaps selecting the wrong tool for the organization. In either case, the test tool likely will not be well received, and those who might otherwise become champions for automated testing may instead become the biggest opponents of such tools.
- *Know the budget.* Once the type of tool required has been determined, it may be tempting to go after the best of the breed. However, it is important to take into account the available budget. It's possible to spend months evaluating the most powerful tool, only to find out that its costs exceeds the budget. Additionally, a budget might be needed for training to bring people up to speed on the tool, or additional staffing resources may be required if there is no developer on the testing team.

Most importantly, testers should remember that there is no one best tool for all environments out there on the market. All tools have their pros and cons for different environments. Which tool is the best depends on the system-engineering environment and other organizational specific requirements and criteria.

To work around the limitations of an automated testing tool and allow deeper testing of core components, a test harness can be developed. Usually written in a robust programming language, as in a stand-alone C++ or VB program, a custom-built test harness typically is faster and more flexible than an automated test-tool script that may be constrained by the test tool's specific environment.

For an example of a testing task appropriate for a test harness, take an application whose purpose is to compute calculations based on user-supplied information and then generate reports based on those computations. The computations may be complex, and sensitive to different combinations of many possible input parameters. As a result, there could be millions of potential variations that produce different results, making comprehensive testing of the computations a significant undertaking.

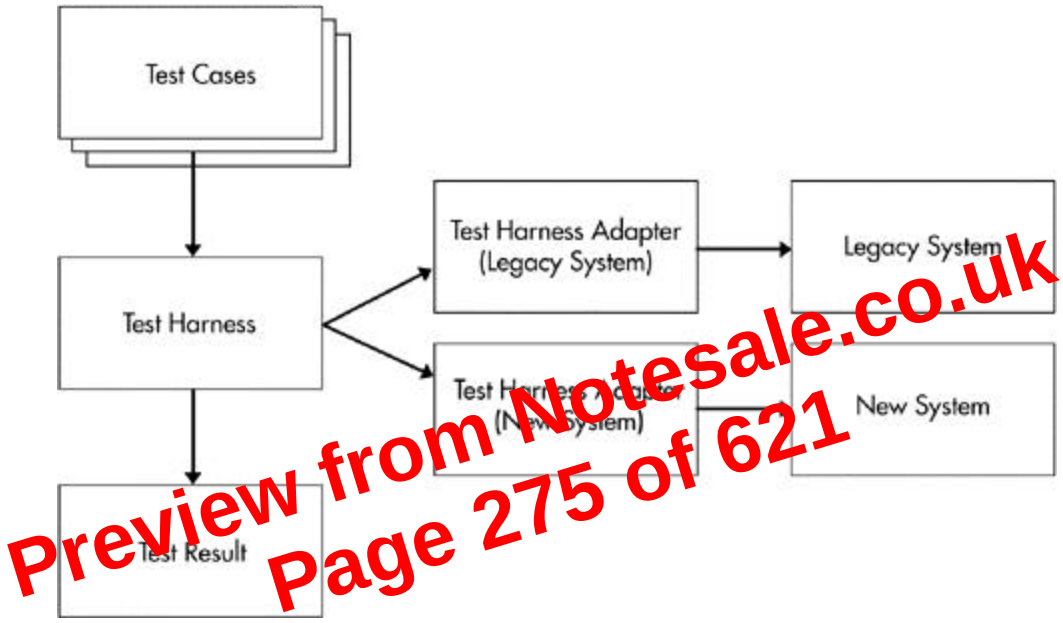
It is very time-consuming to develop and verify thousands of computational test cases by hand. In most cases, it would be far too slow to execute a large volume of test cases through the user interface. A more effective alternative may be to develop a test harness that executes test cases against the application's code, typically below the user-interface layer, directly against core components.

Another way to use a test harness is to compare a new component against a legacy component or system. Often, two systems use different data-storage formats, and have different user interfaces with different technologies. In such a case, any automated test tool would require a special mechanism, or duplicate automated test scripts, in order to run identical test cases on the two systems and generate comparable results. In the worst case, a single testing tool is not compatible with both systems, so duplicate test scripts must be developed using two different automated testing tools. A better alternative would be to build a custom, automated test harness that encapsulates the differences between the two systems into separate modules and allows targeted testing to be performed against both systems. An automated test harness could take the test results generated by a legacy system as a baseline, and automatically verify the results generated by the new system by comparing the two result sets and outputting any differences.

One way to implement this is with a test harness adapter pattern. A **test-harness adapter** is a module that translates or "adapts" each system under test to make it compatible with the test harness, which executes pre-defined test cases against systems through the adapters, and stores the results in a standard format that can be automatically compared from one run to the next. For each system to be tested, a specific adapter must be developed that is capable of interacting with the system—

directly against its DLLs or COM objects, for example—and executing the test cases against it. Testing two systems with a test harness would require two different test adapters and two separate invocations of the test harness, one for each system. The first invocation would produce a test result that would be saved and then compared against the test result for the second invocation. Figure 37.1 depicts a test harness capable of executing test cases against both a legacy system and a new system.

Figure 37.1. Basic Test-Harness Architecture



Identical test cases can be run against multiple systems using a test harness adapter for each system. The adapter for a legacy system can be used to establish a set of baseline results against which the results for the new system can be compared.

The test-harness adapter works by taking a set of test cases and executing them in sequence directly against the application logic of each system under test, bypassing the user interface. Bypassing the user interface optimizes performance, allowing for maximum throughput of the test cases. It also provides greater stability. If the test harness relied upon the user interface, any change to the interface (which often undergoes extensive revision during the development life cycle) could cause the test harness to report false positives. Reviewing such results would waste precious time.

Results from each test case are stored in one or more results files, in a format (such as XML) that is the same regardless of the system under test. Results files can be

retained for later comparison to results generated in subsequent test runs. The comparisons can be performed by a custom-built result-comparison tool programmed to read and evaluate the results files and output any errors or differences found. It is also possible to format the results so they can be compared with a standard "file diff" (file-difference comparison) tool.

As with any type of testing, test harness test cases may be quite complex, especially if the component tested by the harness is of a mathematical or scientific nature. Because there are sometimes millions of possible combinations of the various parameters involved in calculations, there are also potentially millions of possible test cases. Given time and budget constraints, it is unlikely that all possible cases will be tested; however, many thousands of test cases can feasibly be executed using a test harness.

With thousands of different test cases to be created and executed, test-case management becomes a significant effort. Detailed below is a general strategy for developing and managing test cases for use with a test harness. This strategy is also applicable to other parts of the testing effort.

1. *Create test cases.* Test cases for a test harness are developed in the same fashion as for manual testing, using various **test techniques.** A test technique is a formalized approach to choosing test conditions that give a high probability of finding defects. Instead of guessing at which test cases to choose, test techniques help testers derive test conditions in a rigorous and systematic way. A number of books on testing describe testing techniques such as equivalence partitioning, boundary-value analysis, cause-effect graphing, and others. These are discussed in detail in Item 25, but a brief overview is provided here:
    - **Equivalence partitioning** identifies the ranges of inputs and initial conditions expected to produce the same results. Equivalence relies on the commonality and variances among the different situations in which a system is expected to work.
    - **Boundary-value testing** is used mostly for testing input-edit logic. Boundary conditions should always be part of the test scenarios, because many defects occur on the boundaries. Boundaries define three sets or classes of data: good, bad, and on-the-border (in-bound, out-of-bound, and on-bound).
    - **Cause-effect graphing** provides concise representations of logical conditions and corresponding actions, shown in graph form with causes on the left and effects on the right.

- **Orthogonal-array testing** enables the selection of the combinations of test parameters that provide maximum coverage using a minimum number of test cases. Orthogonal-array test cases can be generated in an automated fashion.

2. *Establish a common starting point.* All testing must begin at a well-defined starting point that is the same every time the test harness is executed. Usually, this means the data used in each system during each test must be the same so that the results can be properly compared. When each modular test component is reused, it will be able to set the application state back to the way it found it for the next test component to run. Were this not the case, the second test component would always fail, because the assumed starting point would be incorrect.
3. *Manage test results.* Test scripts produce results for every transaction set they execute. These results are generally written to a file. A single test script can write results to as many files as desired, though in most cases a single file should be sufficient. When running a series of test cases, several test-results files are created. Once baselined, any given test case should produce the same results every time it is executed, test-results files can be compared directly via simple file-comparison routines or by using a custom-developed test-results comparison tool. Any differences found during comparisons must be evaluated in order to identify, document, and track to closure the defects causing those differences.

A custom-built test harness can provide a level of testing above and beyond that of automated test-tool scripts. Although creating a test harness can be time-consuming, it offers various advantages, including deeper coverage of sensitive application areas and ability to compare two applications that cannot be tested using a single off-the-shelf test tool.

Item 38: Use Proven Test-Script Development Techniques

Test-script development can be regarded as a software development project of its own. Proven techniques should be used to produce efficient, maintainable, and reusable test scripts. This will require some additional work on the part of the test team, but in the end the testing effort will profit from a more effective automated testing product.

Consider the following techniques when developing test scripts using functional testing tools.

success. The argument may also be made that nonfunctional issues can be addressed at a later time, such as with a version upgrade or patch.

Unfortunately, this approach can lead to problems in the application's implementation, and even increased risk of failure in production. For example, ignoring security on a Web application for the sake of implementing functionality may leave it vulnerable to attack from malicious Internet users, which in turn can result in downtime, loss of customer data, and negative public attention to the site, ultimately resulting in loss of revenue. As another example, consider an application that is functionally complete, but unable to process a sufficient amount of customer data. Although the application provides the proper functions, it is useless because it does not meet the customer's needs. Again, problems like these can lead to negative publicity for the application and lost customers. These kinds of problems can often undo the effort spent to implement functionality, and can take an enormous amount of effort to correct.

Nonfunctional considerations ideally are investigated early in an application's architecture and design phases. Without early attention to these aspects of the implementation, it may be difficult or impossible later to modify or add components to satisfy the nonfunctional requirements. Consider the following examples:

- *Web application performance.* Web applications are typically developed in small environments, such as one consisting of a single Web server and a single database server. In addition, when the system is first placed into production, it is most cost effective to use the minimum hardware necessary to service the initially small number of users. Over time, however, the load on the Web application may increase, requiring a corresponding increase in the site's hardware capacity to handle the load. If the hardware capacity is not increased, users will experience performance problems, such as excessive time for loading pages and possibly even time-outs at the end-user's browser. Typically, Web-site capacity is increased by adding several machines to the site to **scale** the Web application to achieve higher performance. If this type of expansion was not considered in the application's original architecture and implementation, considerable design and implementation changes may be required to achieve scalability. This results in higher costs, and, perhaps worst of all, considerable delay as engineers work to develop and roll out the improved production site.
- *Use of an incompatible third-party control.* One way to increase an application's functionality while reducing development time is to use third-

requirements. This eliminates the need to repeatedly state the same nonfunctional concerns in every requirements document.

Nonfunctional requirements are usually documented in two ways:

1. A system-wide specification is created that defines nonfunctional requirements for all use cases in the system. An example: "The user interface of the Web system must be compatible with Netscape Navigator 4.x or higher and Microsoft Internet Explorer 4.x or higher."
2. Each requirement description contains a section titled "Nonfunctional Requirements," which documents any specific nonfunctional needs of that particular requirement that differ from the system-wide specifications.

Item 42: Conduct Performance Testing with Production-Sized Databases

Testing teams responsible for an application that manages data must be cognizant that application performance typically degrades as the amount of data stored by the application increases. Database and application optimization techniques can greatly reduce this degradation. It is critical, therefore, to test the application to ensure that optimization has been successfully employed.

To chart application performance across data sets of different sizes, it is usually necessary to test with a variety of data sets. For example, an application may be tested with 1, 100, 500, 1,000, 5,000, 10,000, 50,000, and 100,000 records to investigate how the performance changes as the data quantity grows. This type of testing also makes it possible to find the "upper bound" of the application's data capability, meaning the largest database with which the application performs acceptably.

It is critical to begin application performance testing as soon as possible in the development life cycle. This allows performance improvements to be incorporated while the application is still under development, rather than after significant portions of the application have been developed and tested. Early on, it is acceptable to focus on general performance testing, as opposed to performance fine-tuning. During the early stages, any glaring performance problems should be corrected; finer tuning and optimization issues can be addressed later in the development cycle.

their own security-related requirements, such as maximum lengths for user-supplied inputs.[1]

> [1] Input-length checking is vital for preventing buffer-overflow attacks on an application. For more information, see Elfriede Dustin et al., *Quality Web Systems* (Boston, Mass.: Addison-Wesley, 2002), 76–79.

With the security-related requirements properly documented, test procedures can be created to verify that the system meets them. Some security requirements can be verified through the application's user interface, as in the case of input-length checking. In other cases, it may be necessary to use gray-box testing, as described in Item 16, to verify that the system meets the specified requirement. For example, the requirements for the log-on feature may specify that user name and password must be transmitted in encrypted form. A network-monitoring program must be used to examine the data packets sent between the client and the server to verify that the credentials are in fact encrypted. Still other requirements may require analysis of database tables or of files on the server's hard disk.

In addition to security concerns that are directly related to particular requirements, a software project has security issues that are global in nature, and therefore are related to the application's architecture and overall implementation. For example, a Web application may have a global requirement that all private customer data of any kind is stored in encrypted form in the database. Because this requirement will undoubtedly apply to many functional requirements throughout the system, it must be examined relative to each requirement. Another example of a system-wide security requirement is a requirement to use SSL (Secure Socket Layer) to encrypt data sent between the client browser and the Web server. The testing team must verify that SSL is correctly used in all such transmissions. These types of requirements are typically established in response to assessments of risk, as discussed in Item 41.

Many systems, particularly Web applications, make use of third-party resources to fulfill particular functional needs. For example, an e-commerce site may use a third-party payment-processing server. These products must be carefully evaluated to determine whether they are secure, and to ensure that they are not employed improperly, in ways that could result in security holes. It is particularly important for the testing team to verify that any information passed through these components adheres to the global security specifications for the system. For example, the testing team must verify that a third-party payment-processing server

user can see and change the data. Once changes are made, the data is sent back to the server, and the database record for that order is updated. Now, if two users simultaneously have the editing dialog open for the same record, they both have copies of the data in their local machines' memory, and can make changes to it. What happens if they both choose to save the data?

The answer depends on how the application is designed to deal with concurrency. Managing multiuser access to a shared resource is a challenge that dates back to the introduction of multiuser mainframes. Any resource that can be accessed by more than one user requires software logic to protect that resource by managing the way multiple users can access and change it at the same time. This problem has only become more common since the advent of network file sharing, relational databases, and client-server computing.

There are several ways for a software application to deal with concurrency. Among these are the following:

- *Pessimistic.* This concurrency model places **locks** on data. If one user has a record open and any other users attempt to read that data in a context that allows editing, the system denies the request. In the preceding example, the first user to open the order for editing gets the lock on the order record. Subsequent users attempting to open the order will be sent a message advising that the order is currently being edited by another user, and will have to wait until the first user saves the changes or cancels the operation. This concurrency model is best in situations when it is highly likely that more than one user will attempt to edit the same data at the same time. The downside with this model is that others users are prevented from accessing data that any one user has open, which makes the system less convenient to use. There is also a certain amount of implementation complexity when a system must manage record locks.
- *Optimistic.* In the optimistic concurrency model, users are always allowed to read the data, and perhaps even to update it. When the user attempts to save the data, however, the system checks to see if the data has been updated by anyone else since the user first retrieved it. If it has been changed, the update fails. This approach allows more users to view data than does the pessimistic model, and is typically used when it is unlikely that several users will attempt to edit the same data at the same time. However, it is inconvenient when a user spends time updating a record only to find that it cannot be saved. The record must be retrieved anew and the changes made again.

the organization should be standardized where possible, and based upon criteria that have been proven in several projects.

It may be determined that the system can ship with some defects to be addressed in a later release or a patch. Before going into production, test results can be analyzed to help identify which defects must be fixed immediately versus which can be deferred. For example, some "defect" repairs may be reclassified as enhancements, and then addressed in later software releases. The project or software development manager, together with the other members of the change-control board, are the likely decision-makers to determine whether to fix a defect immediately or risk shipping the product with the defect.

Additional metrics must be evaluated as part of the exit criteria. For example:

- What is the rate of defect discovery in regression tests on previously working functions—in other words, how often are defect fixes breaking previously working functionality?
- How often are defect corrections failing, meaning that a defect thought to be fixed actually wasn't?
- What is the trend in the rate of discovering new defects as this testing phase proceeds? The defect-opening rate should be declining as testing proceeds.

Testing can be considered complete when the application is in an acceptable state to ship or to go live, meeting the exit criteria, even though it most likely contains defects yet to be discovered.

In a world of limited budgets and schedules, there comes a time when testing must halt and the product must be deployed. Perhaps the most difficult decision in software testing is when to stop. Establishing quality guidelines for the completion and release of software will enable the test team to make that decision.

Item 48: Isolate the Test Environment from the Development Environment

It is important that the test environment be set up by the time the testing team is ready to execute the test strategy.

The test environment must be separated from the development environment to avoid costly oversights and untracked changes to the software during testing. Too often, however, this is not the case: To save costs, a separate test environment is not made available for the testing team.

information the program displayed, as well as the correct information (expected result).

- *Retest failure.* If the defect still appears upon retesting, provide details describing the failure during retest.
- *Category.* The category of behavior being reported may be "defect," "enhancement request," or "change request." The priority of a "change request" or "enhancement request" remains N/A until it is actively being reviewed.
- *Resolution.* Developers are to indicate the corrective action taken to fix the defect.

In reporting defects, **granularity** should be preserved. Each report should cover just one defect. If several defects are interrelated, each should have its own entry, with a cross-reference to the others.

2 Prioritization

The process must define how to assign a level of priority to each defect. The test engineer initially must assess how serious the problem is to the successful operation of the system. The most critical defects cause software to fail and prevent test activity from continuing. Defects are commonly referred to a change-control board (CCB) for further evaluation and disposition, as discussed in Item 4.

A common defect priority classification scheme is provided below.

1. Showstopper— Testing cannot continue because the defect causes the application to crash, expected functionality is not implemented, and so on.
2. Urgent— Incident is extremely important and requires immediate attention.
3. High— Incident is important and should be resolved as soon as possible after Urgent items.
4. Medium— Incident is important but can be resolved in a reasonably longer time frame because a work-around exists.
5. Low— Incident is not critical and can be resolved as time and resources allow.
6. N/A— Priority is not applicable (e.g., for change and enhancement requests).

Defect priority levels must to be tailored to the project and organization. For some projects, a simple "High/Low" or "High/Medium/Low" system may suffice. Other projects may need many more levels of defect prioritization. However, having too

I. Executive Summary

This overview has been prepared by NSTL, Inc., the world's leading mobile testing and quality assurance services organization. It is being offered to provide insight into mobile application development challenges and beneficial testing methodologies.

The international mobile marketplace is growing across all market segments. According to a report from Strategy Analytics, early mobile adopters alone will account for nearly $88 billion dollars in mobile service revenues in 2004 and worldwide revenues from mobile data services will increase from $61 billion in 2004 to $189 billion in 2009. Mobile entertainment applications are expected to account for 28% of those revenues in 2009. In terms of worldwide growth, In-Stat/MDR reports that the mobile handset market will see an increase of 14.5% in total subscribers from 2003.

What all of these statistics indicate is that, first, worldwide usage of mobile products is going to continue to grow. As mobile handsets embrace new technologies and more capabilities and as consumer comfort with using mobile products continues to grow, the mobile applications marketplace should continue to be a profitable one. To help realize this potential, mobile application developers, handset manufacturers and operators must understand the need to require testing and quality assurance standards across the industry. This movement toward standardized mobile application testing requirements has already begun to unfold. By doing so, there is much to gain for all.

While there will always be a need for diversification and platforms, the increasingly popular sentiment is that there exists a need for standardization in the testing requirements of operators and within each of the individual platforms. This overview of the mobile application marketplace, takes a brief look into current challenges and benefits of creating these industry standards for mobile application testing for developers, operators, handset manufacturers and consumers too.

With better quality control, **consumers** benefit from increased confidence and more powerful handset capabilities and functionality. With increased application support and accountability, consumers will have more comfort adopting these new mobile technologies.

Operators gain assurances that the quality of applications being tested will be held to a higher set of standards in terms of both quality and network security, Further, application interoperability means lower QA costs and less user downtime.

There are benefits of testing and QA throughout the global mobile marketplace. Standardization in testing requirements across operator networks and within platform environments will help to create a mobile environment that is truly without boundaries.

II. Challenges for Mobile Application Developers

Historically, application developers have been able to create software and get it to the market the way most products come to market. There were traditional channels, such as retail, VARs, online and printed catalogs. In the mobile environment, most applications are developed for inclusion on storefronts that enable end users to download new applications specific to their handset model and/or operator. The advent of mobile platforms capable of running applications has opened vast new markets for developers, while the nature of the cellular industry has created new obstacles to market. Now, device manufacturers, platform owners, and mobile operators each have a "gatekeeper's" share of the road to market. Each stakeholder has specific requirements of which developers must be aware and to which they must adhere. The most widely available path to market is not through those applications that are embedded into a mobile device prior to its market launch.

For developers considering the mobile market, there are many unique – and distinct – challenges that must be successfully navigated before an application can be brought to market. To begin to understand the environment in which developers must operate, the equation begins by accounting for the challenges that all software developers must address. Add to that the challenges of a crucial need for timely market launch, cost-effective management of the testing process and a constant need to refresh platform, operator and handset expertise. The tools required to navigate the "maze to market" and solve the equation quickly and effectively are crucial to the success of each and every mobile application developer.

Severity

The severity tells the reader of the bug how bad the problem is. Or in other words, say what the results of the bug are. Here's a common list for judging the severity of bugs. There is sometimes disagreement about how bad a bug is. This list takes the guess work out of assigning a severity to bugs.

Rating	Value
Blue screen	1
Loss without a work around	2
Loss with a work around	3
Inconvenient	4
Enhancement	5

Likelihood

Put yourself in the average user's place. How likely is a user to encounter this bug? While the tester may encounter this bug every day with every build, if the user isn't likely to see it, how bad can the bug be?

Rating	Value
Always	1
Usually	2
Sometimes	3
Rarely	4
Never	5

Severity * Likelihood = Rating

Computing the rating of a bug is done by multiplying the numeric value given to the severity and likelihood status'. Do the math by hand or let your defect tracker do it for you.

The trick is to remember that the lower the number, the more severe the bug is. The highest rating is a 25 (5 X 5), the lowest is 1 (1 X 1). The bug with a 1 rating should be fixed first while the bug with a 25 rating may never get fixed.

Looking at a list of these bugs ordered by rating means the most important ones will be at the top of the list to be dealt with first. Sorting bugs this way also lets management know whether the product is ready to ship or not. If the number of severe (1) bugs is zero, the product can ship. If there are any severe bugs, then bug fixing must continue.

Other useful information

Who's the bug *Assigned to;* who's going to be responsible for the bug and do the work on the bug?

What *Platform* was the bug found on – Windows, Linux, etc. Is the bug specific to one platform or does it occur on all platforms?

What *Product* was the bug found in? If your company is doing multiple products this is a good way to track those products.

What *Company* would be concerned about this bug? If your company is working with multiple companies either as an OEM or as customer this is a good way to track that information.

Whatever else you want or need to keep track of. Some of these fields will also have value to marketing and sales. It's a useful way to track information about companies and clients.

An example of a bug report:

so if the bug has to go back to him, it will make back onto his list. This procedure ensures that bugs don't fall between the cracks.

The following is a list of status' that a developer can assign to a bug.

Fixed

The *Fixed* status indicates that a change was made to the code and will be available in the next build. Testers search the database on a daily basis looking for all Fixed status bugs. Then the bug reporter or tester assigned to the feature retests the bug duplicating the original circumstances. If the bug is fixed and it is now working properly, another test with slightly different circumstances is performed to confirm the fix. If the bug passes both tests, it gets a *Tested* status.

If the bug doesn't pass the test, the bug is given a *Verified* status and sent back to the developer. Notice here that since the bug's Assigned To field has retained the developer's name, it's an easy process for the tester to send the bug back by simply changing the status to Submitted.

Duplicate

The *Duplicate* status bug is the same as a previously reported bug. Sometimes only the developer or person looking at the code can tell that the bug is a duplicate. It's not always obvious from the surface. A note indicating the previous bug number is placed on the duplicate bug. A note is also placed on the original bug indicating that a duplicate bug exists. When the original bug is fixed and tested, the duplicate bug will be tested also. If the bug really is a duplicate of previous bug then the when the previous bug is fixed, the duplicate bug will also be fixed. If this the case then both bugs get a *Tested* status.

If the duplicate is still a bug, while the original bug is working properly, the duplicate bug is no longer has a duplicate status. It gets a Submitted status and is sent back to the developer. This is a "fail-safe" built into the bug life cycle. It's a check and balance that prevents legitimate bugs from being swept under the carpet or falling between the cracks.

A note of warning. Writing lots of duplicate bugs will get a tester a reputation for being an "airhead". It pays to set time aside daily to read all the new bugs written the previous day.

Resolved

Resolved means that the problem has been taken care of but no code has been changed. For example, bugs can be resolved by setting new device drivers or third party software. Resolved bugs are tested to make sure that the problem really has been resolved with the new situation. If the problem no longer occurs, the bug gets a Tested status. If the Resolved bug still occurs, it is sent back to the developer with a Submitted status.

Need More Information

Need More Information or "NMI" indicates that the bug verifier or developer does not have enough information to duplicate or fix the bug; for example, the steps to duplicate the bug may be unclear or incomplete. The developer changes the status to 'Need More Information' and includes a question or comments to the reporter of the bug. This status is a flag to the bug reporter to supply the necessary information or a demonstration of the problem. After updating the bug information (in the Notes field), the status is put back to Verified so the developer can continue working on the bug. If the bug reporter can not duplicate the bug, it is given a *Can't Duplicate* status along with a note indicating the circumstances.

The only person who can put "Can't Duplicate" on a bug is the person who reported it (or the person testing it). The developer can NOT use this status, he must put Need More Information on it to give the bug reporter a chance to work on the bug.

This is another example of a "fail-safe" built into the database. It is vital at this stage that the bug be given a second chance. The developer should never give a bug a 'Can't Duplicate' status. The bug reporter needs an opportunity to clarify or add information to the bug or to retire it.

Working as Designed

The developer has examined the bug, the product requirements and the design documents and determined that the bug is not a bug, it is *Working as Designed*. What the product or code is doing is intentional as per the design. Or as someone more aptly pointed out it's "working as coded"! It's doing exactly what the code said to do.

8) The tester retests the bug and the same problem persists, so the tester after confirmation from test leader reopens the bug and marks it with 'Reopen' status. And the bug is passed back to the development team for fixing.

< V > Cycle V:
1) A tester finds a bug and reports it to Test Lead.
2) The Test lead verifies if the bug is valid or not.
3) The bug is verified and reported to development team with status as 'New'.
4) The developer tries to verify if the bug is valid but fails in replicate the same scenario as was at the time of testing, but fails in that and asks for help from testing team.
5) The tester also fails to re-generate the scenario in which the bug was found. And developer rejects the bug marking it 'Rejected'.

< VI > Cycle VI:
1) After confirmation that the data is unavailable or certain functionality is unavailable, the solution and retest of the bug is postponed for indefinite time and it is marked as 'Postponed'.

< VII > Cycle VII:
1) If the bug does not stand importance and can be/needed to be postponed, then it is given a status as 'Deferred'.

This way, any bug that is found ends up with a status of Closed, Rejected, Deferred or Postponed.

The main purpose behind any Software Development process is to provide the client (Final/End User of the software product) with a complete solution (software product), which will help him in managing his business/work in cost effective and efficient way. A software product developed is considered successful if it satisfies all the requirements stated by the end user.

Any software development process is incomplete if the most important phase of Testing of the developed product is excluded. Software testing is a process carried out in order to find out and fix previously undetected bugs/errors in the software product. It helps in improving the quality of the software product and makes it ready for client to use.

What is a bug/error?

A bug or error in software product is any exception that can hinder the functionality of either the whole software or part of it.

How do I find out a BUG/ERROR?

Basically, test cases/scripts are run in order to find out any unexpected behavior of the software product under test. If any such unexpected behavior or exception occurs, it is called as a bug.

What is a Test Case?

A test case is a noted/documented set of steps/activities that are carried out or executed on the software in order to confirm its functionality/behavior to certain set of inputs.

What do I do if I find a bug/error?

In normal terms, if a bug or error is detected in a system, it needs to be communicated to the developer in order to get it fixed.

Right from the first time any bug is detected till the point when the bug is fixed and closed, it is assigned various statuses which are New, Open, Postpone, Pending Retest, Retest, Pending Reject, Reject, Deferred, and Closed.

chapter).

Software Production Process Models

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact that the operational models can be viewed as computational scripts or programs: programs that implement a particular regimen of software engineering and development. Non-operational models on the other hand denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification or automated processing.

Non-Operational Process Models

There are two classes of non-operational software process models of the great interest. These are the spiral model and the continuous transformation models. There is also a wide selection of other non-operational models, which for brevity we label as miscellaneous models. Each is examined in turn.

The Spiral Model. The spiral model of software development and evolution represents a risk-driven approach to software process analysis and structuring (Boehm 1987, Boehm *et al*, 1998). This approach, developed by Barry Boehm, incorporates elements of specification-driven, prototype-driven process methods, together with the classic software life cycle. It does so by representing iterative development cycles as an expanding spiral, with inner cycles denoting early system analysis and prototyping, and outer cycles denoting the classic software life cycle. The radial dimension denotes cumulative development costs, and the angular dimension denotes progress made in accomplishing each development spiral. See Figure 3.

Risk analysis, which seeks to identify situations that might cause a development effort to fail or go over budget/schedule, occurs during each spiral cycle. In each cycle, it represents roughly the same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out only so far as needed according to the risk that must be managed. Alternatively, the spiral model indicates that the classic software life cycle model need only be followed when risks are greatest, and after early system prototyping as a way of reducing these risks, albeit at increased cost. The insights that the Spiral Model offered has in turned influenced the standard software life cycle process models, such as ISO12207 noted earlier. Finally, efforts are now in progress to integrate computer-based support for stakeholder negotiations and capture of trade-off rationales into an operational form of the WinWin Spiral Model (Boehm et al, 1998). (see Risk Management in Software Development)

Miscellaneous Process Models. Many variations of the non-operational life cycle and process models have been proposed, and appear in the proceedings of the international software process workshops sponsored by the ACM, IEEE, and Software Process Association. These include fully

executable models. Three classes of operational software process models can be identified and examined. Following this, we can also identify a number of emerging trends that exploit and extend the use of operational process models for software engineering.

Operational specifications for rapid prototyping. The operational approach to software development assumes the existence of a formal specification language and processing environment that supports the evolutionary development of specifications into an prototype implementation (Bauer 1976, Balzer 1983, Zave 1984). Specifications in the language are coded, and when computationally evaluated, constitute a functional prototype of the specified system. When such specifications can be developed and processed incrementally, the resulting system prototypes can be refined and evolved into functionally more complete systems. However, the emerging software systems are always operational in some form during their development. Variations within this approach represent either efforts where the prototype is the end sought, or where specified prototypes are kept operational but refined into a complete system.

The specification language determines the power underlying operational specification technology. Simply stated, if the specification language is a conventional programming language, then nothing new in the way of software development is realized. However, if the specification incorporates (or extends to) syntactic and semantic language constructs that are specific to the application domain, which usually are not part of conventional programming languages, then domain-specific rapid prototyping can be supported.

An interesting twist worthy of note is that it is generally within the capabilities of many operational specification languages to specify "systems" whose purpose is to serve as a model of an arbitrary abstract process, such as a software process model. In this way, using a prototyping language and environment, one might be able to specify an abstract model of some software engineering processes as a system that produces and consumes certain types of documents, as well as the classes of development transformations applied to them. Thus, in this regard, it may be possible to construct operational software process models that can be executed or simulated using software prototyping technology. Humphrey and Kellner describe one such application and give an example using the graphic-based state-machine notation provided in the STATECHARTS environment (Humphrey 1989).

Software automation. Automated software engineering (also called knowledge-based software engineering) attempts to take process automation to its limits by assuming that process specifications can be used directly to develop software systems, and to configure development environments to support the production tasks at hand. The common approach is to seek to automate some form of the continuous transformation model (Bauer 1976, Balzer 1985). In turn, this implies an automated environment capable of recording the formalized development of operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by incorporating the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation (Balzer 1983b, Balzer 1985). However, current progress has been limited to demonstrating such mechanisms and specifications on software coding, maintenance, project communication and management tasks (Balzer 1983b, Balzer 1985, Sathi 1985, Mi 1990, Scacchi and Mi 1997), as well as to software component catalogs and formal models of software

Boehm, B., A Spiral Model of Software Development and Enhancement, *Computer*, 20(9), 61-72, 1987.

Boehm, B., A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, Using the WinWin Spiral Model: A Case Study, *Computer*, 31(7), 33-44, 1998.

Bolcer, G.A., R.N. Taylor, Advanced workflow management technologies, *Software Process--Improvement and Practice*, 4,3, 125-171, 1998.

Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, *Approaches to Prototyping*, Springer-Verlag, New York, 1984.

Chatters, B.W., M.M. Lehman, J.F. Ramil, and P. Werwick, Modeling a Software Evolution Process: A Long-Term Case Study, Software Process-Improvement and Practice, 5(2-3), 91-102, 2000.

Cook, J.E., and A.Wolf, Discovering models of software processes from event-based data, *ACM Trans. Softw. Eng. Methodol*. 7, 3 (Jul. 1998), 215 - 249

B. Curtis, H. Krasner, V. Shen, and N. Iscoe, On Building Software Process Models Under the Lamppost, *Proc. 9th. Intern. Conf. Software Engineering*, IEEE Computer Society, Monterey, CA, 96-103, 1987.

Curtis, B., H. Krasner, and N. Iscoe, A Field Study of the Software Design Process for Large Systems, *Communications ACM*, 31, 11, 1268-1287, November, 1988.

Cusumano, M. and D. Yoffie, Software Development on Internet Time, *Computer*, 32(10), 60-69, 1999.

Distasco, J., Software Management-A Survey of Practice in 1980, *Proceedings IEEE*, 68,9,1103-1119, 1980.

DiBona, C., S. Ockman and M. Stone, *Open Sources: Voices from the Open Source Revolution*, O'Reilly Press, Sebastopol, CA, 1999.

Fogel, K., *Open Source Development with CVS*, Coriolis Press, Scottsdale, AZ, 1999.

Garg, P.K. and M. Jazayeri (eds.), *Process-Centered Software Engineering Envir*onment, IEEE Computer Society, pp. 131-140, 1996.

Garg, P.K., P. Mi, T. Pham, W. Scacchi, and G. Thunquest, The SMART approach for software process engineering, *Proc. 16th. Intern. Conf. Software Engineering,* 341 - 350,1994.

Garg, P.K. and W. Scacchi, ISHYS: Design of an Intelligent Software Hypertext Environment, *IEEE Expert*, 4, 3, 52-63, 1989.

Garg, P.K. and W. Scacchi, A Hypertext System to Manage Software Life Cycle Documents, *IEEE Software*, 7, 2, 90-99, 1990.

Goguen, J., Reusing and Interconnecting Software Components, *Computer*, 19,2, 16-28, 1986.

Graham, D.R., Incremental Development: Review of Non-monolithic Life-Cycle Development Models, *Information and Software T*echnology, 31, 1, 7-20, January,1989.

Grundy, J.C.; Apperley, M.D.; Hosking, J.G.; Mugridge, W.B. A decentralized architecture for software process modeling and enactment, *IEEE Internet Computing* , Volume: 2 Issue: 5 , Sept.-Oct. 1998, 53 -62.

Grinter, R., Supporting Articulation Work Using Software Configuration Management*, J. Computer Supported Cooperative Work*,5, 447-465, 1996.

Heineman, G., J.E. Botsford, G. Caldiera, G.E. Kaiser, M.I. Kellner, and N.H. Madhavji., Emerging Technologies that Support a Software Process Life Cycle. *IBM Systems J.*, 32(3):501-529, 1994.

Hekmatpour, S., Experience with Evolutionary Prototyping in a Large Software Project, *ACM Software Engineering Notes*, 12,1, 38-41 1987

Hoffnagel, G. F., and W. Beregi, Automating the Software Development Process, *,IBM Systems J.*,24 ,2 1985 ,102-120

Horowitz, E. and R. Williamson, SODOS: A Software Documentation Support Environment--Its Definition, *IEEE Trans. Software Engineering*, 12, 8, 1986.

Horowitz, E., A. Kemper, and B. Narasimhan, A Survey of Application Generators, *IEEE Software*, 2,1 ,40-54, 1985.

Hosier, W. A., Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming, *IRE Trans. Engineering Management,* EM-8, June, 1961.

Humphrey, W. S., The IBM Large-Systems Software Development Process: Objectives and Direction, *,IBM Systems J.*, 24,2, 76-78, 1985.

Humphrey, W.S. and M. Kellner, Software Process Modeling: Principles of Entity Process Models, *Proc. 11th. Intern. Conf. Software Engineering*, IEEE Computer Society, Pittsburgh, PA, 331-342, 1989.

Kaiser, G., P. Feiler, and S. Popovich, Intelligent Assistance for Software Development and Maintenance, *IEEE Software*, 5, 3, 1988.

Kling, R., and W. Scacchi, The Web of Computing: Computer Technology as Social Organization, *Advances in Computers*, 21, 1-90, Academic Press, New York, 1982.

Lehman, M. M., Process Models, Process Programming, Programming Support, *Proc. 9th. Intern. Conf. Software Engineering*, 14-16, IEEE Computer Society, 1987.

Lehman, M. M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, New York, 1985

information/documentation is available and up-to-date - preferably electronic, not paper; promote teamwork and cooperation; use protoypes and/or continuous communication with end-users if possible to clarify expectations.

What is software 'quality'?

Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable. However, quality is obviously a subjective term. It will depend on who the 'customer' is and their overall influence in the scheme of things. A wide-angle view of the 'customers' of a software development project might include end-users, customer acceptance testers, customer contract officers, customer management, the development organization's management/accountants/testers/salespeople, future software maintenance engineers, stockholders, magazine columnists, etc. Each type of 'customer' will have their own slant on 'quality' - the accounting department might define quality in terms of profits, while an end-user might define quality as user-friendly and bug-free.

What is 'good code'?

'Good code' is code that works, is bug free, and is readable and maintainable. Some organizations have coding 'standards' that all developers are supposed to adhere to, but everyone has different ideas about what's best, or what is too many or too few rules. There are also various theories and metrics, such as McCabe Complexity metrics. It should be kept in mind that excessive use of standards and rules can stifle productivity and creativity. 'Peer reviews', 'buddy checks' code analysis tools, etc. can be used to check for problems and enforce standards.

For C and C++ coding, here are some typical ideas to consider in setting rules/standards; these may or may not apply to a particular situation:

- minimize or eliminate use of global variables.
- use descriptive function and method names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- use descriptive variable names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- function and method sizes should be minimized; less than 100 lines of code is good, less than 50 lines is preferable.
- function descriptions should be clearly spelled out in comments preceding a function's code.

Critical. (Note that documentation can be electronic, not necessarily paper, may be embedded in code comments, etc.) QA practices should be documented such that they are repeatable. Specifications, designs, business rules, inspection reports, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. should all be documented in some form. There should ideally be a system for easily finding and obtaining information and determining what documentation will have a particular piece of information. Change management for documentation should be used if possible.

What's the big deal about 'requirements'?

One of the most reliable methods of ensuring problems, or failure, in a large, complex software project is to have poorly documented requirements specifications. Requirements are the details describing an application's externally-perceived functionality and properties. Requirements should be clear, complete, reasonably detailed, cohesive, attainable, and testable. A non-testable requirement would be, for example, 'user-friendly' (too subjective). A testable requirement would be something like 'the user must enter their previously-assigned password to access the application'. Determining and organizing requirements details in a useful and efficient way can be a difficult effort; different methods are available depending on the particular project. Many books are available that describe various approaches to this task.

Care should be taken to involve ALL of a project's significant 'customers' in the requirements process. 'Customers' could be in-house personnel or out, and could include end-users, customer acceptance testers, customer contract officers, customer management, future software maintenance engineers, salespeople, etc. Anyone who could later derail the project if their expectations aren't met should be included if possible.

Organizations vary considerably in their handling of requirements specifications. Ideally, the requirements are spelled out in a document with statements such as 'The product shall.....'. 'Design' specifications should not be confused with 'requirements'; design specifications should be traceable back to the requirements.

In some organizations requirements may end up in high level project plans, functional specification documents, in design documents, or in other documents at various levels of detail. No matter what they are called, some type of documentation with detailed requirements will be needed by testers in order to properly plan and execute tests. Without such documentation, there will be no clear-cut way to determine if a software application is performing correctly.

'Agile' methods such as XP use methods requiring close interaction and cooperation between programmers and customers/end-users to iteratively develop requirements. In the XP 'test first' approach developmers create automated unit testing code before the application code, and these automated unit tests essentially embody the requirements.

What steps are needed to develop and run software tests?

The following are some of the steps to consider:

- Obtain requirements, functional design, and internal design specifications and other necessary documents
- Obtain budget and schedule requirements
- Determine project-related personnel and their responsibilities, reporting requirements, required standards and processes (such as release processes, change processes, etc.)
- Determine project context, relative to the existing quality culture of the organization and business, and how it might impact testing scope, aproaches, and methods.
- Identify application's higher-risk aspects, set priorities, and determine scope and limitations of tests
- Determine test approaches and methods - unit, integration, functional, system, load, usability tests, etc.
- Determine test environment requirements (hardware, software, communications, etc.)
- Determine testware requirements (record/playback tools, coverage analyzers, test tracking, problem/bug tracking, etc.)
- Determine test input data requirements
- Identify tasks, those responsible for tasks, and labor requirements
- Set schedule estimates, timelines, milestones
- Determine input equivalence classes, boundary value analyses, error classes
- Prepare test plan document and have needed reviews/approvals
- Write test cases
- Have needed reviews/inspections/approvals of test cases
- Prepare test environment and testware, obtain needed user manuals/reference documents/configuration guides/installation guides, set up test tracking processes, set up logging and archiving processes, set up or obtain test input data
- Obtain and install software releases
- Perform tests
- Evaluate and report results
- Track problems/bugs and fixes
- Retest as needed
- Maintain and update test plans, test cases, test environment, and testware through life cycle

What's a 'test plan'?

A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product. The completed document will help people outside the test group understand the 'why' and 'how' of product validation. It should be thorough enough to be useful but not so thorough that no one outside the test group will read it. The following are some of the items that might be included in a test plan, depending on the particular project:

- Title
- Identification of software including version/release numbers
- Revision history of document including authors, dates, approvals
- Table of Contents
- Purpose of document, intended audience
- Objective of testing effort
- Software product overview
- Relevant related document list, such as requirements, design documents, other test plans, etc.
- Relevant standards or legal requirements
- Traceability requirements
- Relevant naming conventions and identifier conventions
- Overall software project organization and personnel/contact-info/responsibilties
- Test organization and personnel/contact-info/responsibilities
- Assumptions and dependencies
- Project risk analysis
- Testing priorities and focus
- Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
- Outline of data input equivalence classes, boundary value analysis, error classes
- Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
- Test environment validity analysis - differences between the test and production systems and their impact on test validity.
- Test environment setup and configuration issues
- Software migration processes
- Software CM processes
- Test data setup requirements
- Database setup requirements
- Outline of system-logging/error-logging/other capabilities, and tools such as screen capture software, that will be used to help describe and report bugs
- Discussion of any specialized software or hardware tools that will be used by testers to help track the cause or source of bugs
- Test automation - justification and overview
- Test tools to be used, including versions, patches, etc.
- Test script/test code maintenance processes and version control
- Problem tracking and resolution - tools and processes
- Project test metrics to be used
- Reporting requirements and testing deliverables
- Software entrance and exit criteria
- Initial sanity testing period and criteria
- Test suspension and restart criteria
- Personnel allocation
- Personnel pre-training needs
- Test site/location

- Outside test organizations to be utilized and their purpose, responsibilties, deliverables, contact persons, and coordination issues
- Relevant proprietary, classified, security, and licensing issues.
- Open issues
- Appendix - glossary, acronyms, etc.

What's a 'test case'?

- A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly. A test case should contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results.
- Note that the process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's useful to prepare test cases early in the development cycle if possible.

What should be done after a bug is found?

The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere. If a problem-tracking system is in place, it should encapsulate these processes. A variety of commercial problem-tracking/management software tools are available :

- Complete information such that developers can understand the bug, get an idea of it's severity, and reproduce it if necessary.
- Bug identifier (number, ID, etc.)
- Current bug status (e.g., 'Released for Retest', 'New', etc.)
- The application name or identifier and version
- The function, module, feature, object, screen, etc. where the bug occurred
- Environment specifics, system, platform, relevant hardware specifics
- Test case name/number/identifier
- One-line bug description
- Full bug description
- Description of steps needed to reproduce the bug if not covered by a test case or if the developer doesn't have easy access to the test case/test script/test tool
- Names and/or descriptions of file/data/messages/etc. used in test
- File excerpts/error messages/log file excerpts/screen shots/test tool logs that would be helpful in finding the cause of the problem
- Severity estimate (a 5-level range such as 1-5 or 'critical'-to-'low' is common)
- Was the bug reproducible?
- Tester name
- Test date

the application as efficiently as possible while meeting the test organizations testing mandate.

Test Automation Engineer

The Role of the Test Automation Engineer to is to create automated test case scripts that perform the tests as designed by the Test Designer. To fulfill this role the Test Automation Engineer must develop and maintain an effective test automation infrastructure using the tools and techniques available to the testing organization. The Test Automation Engineer must work in concert with the Test Designer to ensure the appropriate automation solution is being deployed.

Test Methodologist or Methodology Specialist

The Role of the Test Methodologist is to provide the test organization with resources on testing methodologies. To fulfill this role the Methodologist works with Quality Assurance to facilitate continuous quality improvement within the testing methodology and the testing organization as a whole. To this end the methodologist: evaluates the test strategy, provides testing frameworks and templates and ensures effective implementation of the appropriate testing techniques.

Testing Techniques

Over time the IT industry at large and the testing discipline have developed several techniques for analyzing and testing applications.

Black-box Tests

Black-box tests are derived from an understanding of the purpose of the code; knowledge on or about the actual internal program structure is not required when using this approach. The risk involved with this type of approach is that .hidden. (functions unknown to the tester) will not be tested and may not been even exercised.

White-box Tests or Glass-box tests

White-box tests are derived from an intimate understanding of the purpose of the code and the code itself; this allows the tester to test .hidden. (undocumented functionality) within the body of the code. The challenge with any white-box testing is to find testers that are comfortable with reading and understanding code.

by facilitating the determination of current process capabilities and identification of the issues most critical to software quality and process improvement. [SEI/CMU-93-TR-25]

Capture-replay tools. - Tools that gives testers the ability to move some GUI testing away from manual execution by 'capturing' mouse clicks and keyboard strokes into scripts, and then 'replaying' that script to re-create the same sequence of inputs and responses on subsequent test.[Scott Loveland, 2005]

Cause Effect Graphing. (1) [NBS] Test data selection technique. The input and output domains are partitioned into classes and analysis is performed to determine which input classes cause which effect. A minimal set of inputs is chosen which will cover the entire effect set. (2)A systematic method of generating test cases representing combinations of conditions. See: testing, functional.[G. Myers]

Clean test. A test whose primary purpose is validation; that is, tests designed to demonstrate the software`s correct working.(syn. positive test)[B. Beizer 1995]

Clear-box testing. See White-box testing.

Code audit. An independent review of source code by a person, team, or tool to verify compliance with software design documentation and programming standards. Correctness and efficiency may also be evaluated. (IEEE)

Code Inspection. A manual [formal] testing [error detection] technique where the programmer reads source code, statement by statement, to a group who ask questions analyzing the program logic, analyzing the code with respect to a checklist of historically common programming errors, and analyzing its compliance with coding standards. Contrast with code audit, code review, code walkthrough. This technique can also be applied to other software and configuration items. [G.Myers/NBS] Syn: Fagan Inspection

Code Walkthrough. A manual testing [error detection] technique where program [source code] logic [structure] is traced manually [mentally] by a group with a small set of test cases, while the state of program variables is manually monitored, to analyze the programmer's logic and assumptions.[G.Myers/NBS]

Coexistence Testing. Coexistence isn't enough. It also depends on load order, how virtual space is mapped at the moment, hardware and software configurations, and the history of what took place hours or days before. It's probably an exponentially hard problem rather than a square-law problem. [from Quality Is Not The Goal. By Boris Beizer, Ph. D.]

Comparison testing. Comparing software strengths and weaknesses to competing products

Compatibility bug A revision to the framework breaks a previously working feature: a new feature is inconsistent with an old feature, or a new feature breaks an unchanged application rebuilt with the new framework code. [R. V. Binder, 1999]

function used in the previous example. The square root function has two input partitions and two output partitions, as shown in table 3.2.

Input Partitions		Output Partitions	
i	<0	a	>=0
ii	>=0	b	Error

Table 3.2 - Partitions for Square Root

These four partitions can be tested with two test cases:

Test Case 1: Input 4, Return 2
- Exercises the >=0 input partition (ii)
- Exercises the >=0 output partition (a)

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.
- Exercises the <0 input partition (i)
- Exercises the "error" output partition (b)

For a function like square root, we can see that equivalence partitioning is quite simple.

One test case for a positive number and a real result, and a second test case for a negative number and an error result. However, as software becomes more complex, the identification of partitions and the inter-dependencies between partitions becomes much more difficult, making it less convenient to use this technique to design test cases. Equivalence partitioning is still basically a positive test case design technique and needs to be supplemented by negative tests.

Boundary Value Analysis

Boundary value analysis uses the same analysis of partitions as equivalence partitioning. However, boundary value analysis assumes that errors are most likely to exist at the boundaries between partitions. Boundary value analysis consequently incorporates a degree of negative testing into the test design, by anticipating that errors will occur at or near the partition boundaries. Test cases are designed to exercise the software on and at either side of boundary values. Consider the two input partitions in the square root example, as illustrated by figure 3.2.

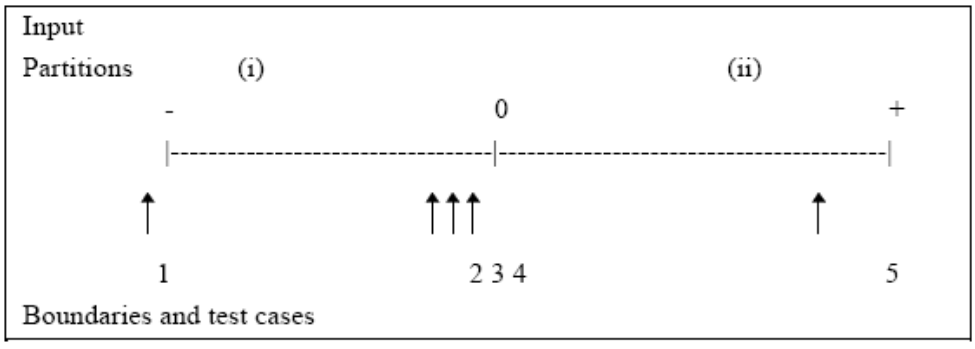


Figure 3.2 - Input Partition Boundaries in Square Root

The zero or greater partition has a boundary at 0 and a boundary at the most positive real number. The less than zero partition shares the boundary at 0 and has another boundary at the most negative real number. The output has a boundary at 0, below which it cannot go.

Test Case 1: Input {the most negative real number}, Return 0, Output "Square root error - illegal negative input" using Print_Line - Exercises the lower boundary of partition (i).

Test Case 2: Input {just less than 0}, Return 0, Output "Square root error – illegal negative input" using Print_Line - Exercises the upper boundary of partition (i).

Test Case 3: Input 0, Return 0
- Exercises just outside the upper boundary of partition (i), the lower boundary of partition (ii) and the lower boundary of partition (a).

Test Case 4: Input {just greater than 0}, Return {the positive square root of the input}
- Exercises just inside the lower boundary of partition (ii).

Test Case 5: Input {the most positive real number}, Return {the positive square root of the input}
- Exercises the upper boundary of partition (ii) and the upper boundary of partition (a).

As for equivalence partitioning, it can become impractical to use boundary value analysis thoroughly for more complex software. Boundary value analysis can also be meaningless for non scalar data, such as enumeration values. In the example, partition (b) does not really have boundaries. For purists, boundary value analysis requires knowledge of the underlying representation of the numbers. A more pragmatic approach is to use any small values above and below each boundary and suitably big positive and negative numbers.

3.4. State-Transition Testing

State transition testing is particularly useful where either the software has been designed as a state machine or the software implements a requirement that has been modeled as a state machine. Test cases are designed to test the transitions between states by creating the events which lead to transitions.

When used with illegal combinations of states and events, test cases for negative testing can be designed using this approach. Testing state machines is addressed in detail by the IPL paper "Testing State Machines with AdaTEST and Cantata".

3.5. Branch Testing

In branch testing, test cases are designed to exercise control flow branches or decision points in a unit. This is usually aimed at achieving a target level of Decision Coverage. Given a functional specification for a unit, a "black box" form of branch testing is to "guess" where branches may be coded and to design test cases to follow the branches. However, branch testing is really a "white box" or structural test case design technique. Given a structural specification for a unit, specifying the control flow within the unit, test cases can be designed to exercise branches. Such a structural unit specification will typically include a flowchart or PDL.

Returning to the square root example, a test designer could assume that there would be a branch between the processing of valid and invalid inputs, leading to the following test cases:

Test Case 1: Input 4, Return 2
- Exercises the valid input processing branch

functionality of a unit that is important, and that branch testing is a means to an end, not an end in itself. Another consideration is that branch testing is based solely on the outcome of decisions. It makes no allowances for the complexity of the logic which leads to a decision.

3.6. Condition Testing

There are a range of test case design techniques which fall under the general title of condition testing, all of which endeavor to mitigate the weaknesses of branch testing when complex logical conditions are encountered. The object of condition testing is to design test cases to show that the individual components of logical conditions and combinations of the individual components are correct.

Test cases are designed to test the individual elements of logical expressions, both within branch conditions and within other expressions in a unit. As for branch testing, condition testing could be used as a "black box" technique, where the test designer makes intelligent guesses about the implementation of a functional specification for a unit. However, condition testing is more suited to "white box" test design from a structural specification for a unit.

The test cases should be targeted at achieving a condition coverage metric, such as Modified Condition Decision Coverage (available as Boolean Operand Effectiveness in AdaTEST). The IPL paper entitled "Structural Coverage Metrics" provides more detail of condition coverage metrics.

To illustrate condition testing, consider the example specification for the square root function which uses successive approximation (figure 3.3(d) - Specification 1). Suppose that the designer for the unit made a decision to limit the algorithm to a maximum of 10 iterations, on the grounds that after 10 iterations the answer would be as close as it would ever get. The PDL specification for the unit could specify an exit condition like that given in figure 3.4.


```
:
:
:
EXIT_LOOP WHEN (error<desired accuracy) or (iterations=10)
:
:
:
```

Figure 3.4 - Loop Exit Condition

If the coverage objective is Modified Condition Decision Coverage, test cases have to prove that both error<desired accuracy and iterations=10 can independently affect the outcome of the decision.

Test Case 1: 10 iterations, error>desired accuracy for all iterations.
- Both parts of the condition are false for the first 9 iterations. On the tenth iteration, the first part of the condition is false and the second part becomes true, showing that the iterations=10 part of the condition can independently affect its outcome.

Test Case 2: 2 iterations, error>=desired accuracy for the first iteration, and error<desired accuracy for the second iteration. - Both parts of the condition are false for the first iteration. On

the second iteration, the first part of the condition becomes true and the second part remains false, showing that the error<desired accuracy part of the condition can independently affect its outcome. Condition testing works best when a structural specification for the unit is available. It provides a thorough test of complex conditions, an area of frequent programming and design error and an area which is not addressed by branch testing. As for branch testing, it is important for test designers to beware that concentrating on conditions could distract a test designer from the overall functionality of a unit.

3.7. Data Definition-Use Testing

Data definition-use testing designs test cases to test pairs of data definitions and uses. A data definition is anywhere that the value of a data item is set, and a data use is anywhere that a data item is read or used. The objective is to create test cases which will drive execution through paths between specific definitions and uses.

Like decision testing and condition testing, data definition-use testing can be used in combination with a functional specification for a unit, but is better suited to use with a structural specification for a unit.

Consider one of the earlier PDL specifications for the square root function which sent every input to the maths co-processor and used the co-processor status to determine the validity of the result. (Figure 3.3(c) - Specification 3). The first step is to list the pairs of definitions and uses. In this specification there are a number of definition-use pairs, as shown in table 3.3.

	Definition	Use
1	Input to routine	By the maths co-processor
2	Co-processor status	Test for status=error
3	Error message	By Print_Line
4	RETURN 0	By the calling unit
5	Answer by co-processor	RETURN the answer
6	RETURN the answer	By the calling unit

Table 3.3 - Definition-Use pairs

These pairs of definitions and uses can then be used to design test cases. Two test cases are required to test all six of these definition-use pairs:

Test Case 1: Input 4, Return 2
- Tests definition-use pairs 1, 2, 5, 6

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input"
using Print_Line. - Tests definition-use pairs 1, 2, 3, 4

The analysis needed to develop test cases using this design technique can also be useful for identifying problems before the tests are even executed; for example, identification of situations where data is used without having been defined. This is the sort of data flow analysis that some static analysis tool can help with. The analysis of data definition-use pairs can become very complex, even for relatively simple units. Consider what the definition-use pairs would be for the successive approximation version of square root!

ORGANIZATIONAL APPROACHES FOR UNIT TESTING

Introduction
Unit testing is the testing of individual components (units) of the software. Unit testing is usually conducted as part of a combined code and unit test phase of the software lifecycle, although it is not uncommon for coding and unit testing to be conducted as two distinct phases.

The basic units of design and code in Ada, C and C++ programs are individual subprograms (procedures, functions, member functions). Ada and C++ provide capabilities for grouping basic units together into packages (Ada) and classes (C++). Unit testing for Ada and C++ usually tests units in the context of the containing package or class.

When developing a strategy for unit testing, there are three basic organizational approaches that can be taken. These are **top down**, **bottom up** and **isolation**.

The concepts of **test drivers** and **stubs** are used throughout this paper. A **test driver** is software which executes software in order to test it, providing a framework for setting input parameters, executing the unit, and reading the output parameters. A **stub** is an imitation of a unit, used in place of the real unit to facilitate testing.

An AdaTEST or Cantata test script comprises a test driver and an (optional) collection of stubs.

2. Top Down Testing

2.1. Description
In **top down** unit testing, individual units are tested by using them from the units which call them, but in isolation from the units called. The unit at the top of a hierarchy is tested first, with all called units replaced by stubs. Testing continues by replacing the stubs with the actual called units, with lower level units being stubbed. This process is repeated until the lowest level units have been tested. Top down testing requires test stubs, but no test drivers.

Figure 2.1 illustrates the test stubs and tested units needed to test unit D, assuming that units A, B and C have already been tested in a top down approach.

A unit test plan for the program shown in figure 2.1, using a strategy based on the top down organisational approach, could read as follows:

Step (1)
Test unit A, using stubs for units B, C and D.
Step (2)
Test unit B, by calling it from tested unit A, using stubs for units C and D.
Step (3)
Test unit C, by calling it from tested unit A, using tested units B and a stub for unit D.
Step (4)
Test unit D, by calling it from tested unit A, using tested unit B and C, and stubs for units E, F and G. (Shown in figure 2.1).
Step (5)
Test unit E, by calling it from tested unit D, which is called from tested unit A, using tested units B and C, and stubs for units F, G, H, I and J.
Step (6)
Test unit F, by calling it from tested unit D, which is called from tested unit A, using tested units B, C and E, and stubs for units G, H, I and J.
Step (7)
Test unit G, by calling it from tested unit D, which is called from tested unit A, using tested units B, C, E and F, and stubs for units H, I and J.

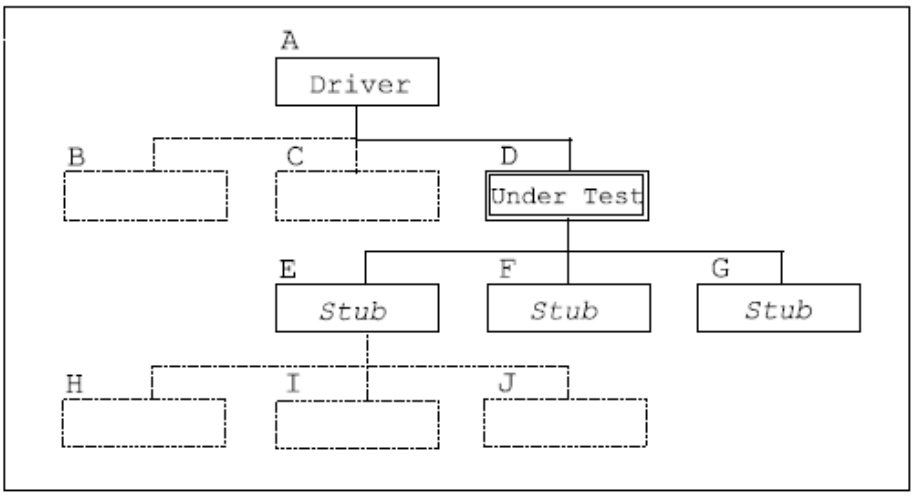


Figure 4.1 - Isolation Testing

A unit test plan for the program shown in figure 4.1, using a strategy based on the isolation organisational approach, need contain only one step, as follows:

Step (1)
(Note that there is only one step to the test plan. The sequence of tests is unimportant, all tests could be executed in parallel.)

Test unit A, using a driver to start the test and stubs in place of units B, C and D;
Test unit B, using a driver to call it in place of unit A;
Test unit C, using a driver to call it in place of unit A;
Test unit D, using a driver to call it in place of unit A and stubs in place of units E, F and G, (Shown in Figure 3.1);

Test unit E, using a driver to call it in place of unit D and stubs in place of units H, I and J;
Test unit F, using a driver to call it in place of unit D;
Test unit G, using a driver to call it in place of unit D;
Test unit H, using a driver to call it in place of unit E;
Test unit I, using a driver to call it in place of unit E;
Test unit J, using a driver to call it in place of unit E.

Advantages
It is easier to test an isolated unit thoroughly, where the unit test is removed from the complexity of other units. Isolation testing is the easiest way to achieve good structural coverage, and the difficulty of achieving good structural coverage does not vary with the position of a unit in the unit hierarchy.
Because only one unit is being tested at a time, the test drivers tend to be simpler than for bottom up testing, while the stubs tend to be simpler than for top down testing. With an isolation approach to unit testing, there are no dependencies between the unit tests, so the unit test phase can overlap the detailed design and code phases of the software lifecycle. Any number of units can be tested in parallel, to give a 'short and fat' unit test phase. This is a useful way of using an increase in team size to shorten the overall time of a software development.

A further advantage of the removal of interdependency between unit tests, is that changes to a unit only require changes to the unit test for that unit, with no impact on other unit tests. This results in a lower cost than the bottom up or top down organisational approaches, especially

you can that will convince others also that this is indeed a valid problem. Evidence may take the form of documentation from user guides, specifications, requirements, and designs. It may be past comments from customers, de-facto standards from competing products, or results from previous versions of the product. Don't assume everyone sees things the same way you do. Don't expect people to read between the lines and draw the same conclusions as you. Don't assume that 3 weeks from now you will remember why you thought this was a bug. Think about what it is that convinced you that this is a bug and include that in the report. You will have to provide even more evidence if you think there is a chance that this situation may not be readily accepted by all as a valid bug.

Mental Checklist

It is important that you develop an easily accessible mental checklist that you go over in your mind each time you write a defect report. Inspections have proven to be the least expensive and most effective means of improving software quality. It stands to reason, that the least expensive most effective means of improving the quality of your defect reports is an inspection, even if it is an informal self-inspection. It is important that using whatever memory techniques work for you that these checklist items get implanted into your memory. In most cases, inadequate defect reports are not due to an inability to write a good report. Usually, we just didn't think about and answer the right questions.

This mental checklist takes us through the process of thinking about and answering the right questions. You may find it useful to apply a mnemonic to the checklist. If you look at the first letter of each item on the checklist it spells CAN PIG RIDE? This is just short enough and obnoxious enough that hopefully it will stick with you. If you spend about 20-30 minutes using this phrase and associating it with the defect inspection checklist, you will probably have that mental checklist implanted in your memory. If ten items are too much to remember, then concentrate on PIG. If you do a good job on these three items (Precise, Isolate, and Generalize) it will guide you to adequate and more effective defect reports in most cases.

Template

A defect remark template can prove useful in making sure that the remarks provide the correct information and answer the right questions. Some defect tracking tools may allow a template to automatically be displayed whenever it prompts for defect remarks. Otherwise, you may have to use cut and paste to insert a template into your remarks.

emphasis is on verification to ensure that the design and programs accomplish the defined requirements. During the test and installation phases, the emphasis is on inspection to determine that the implemented system meets the system specification.

The chart below describes the Life Cycle verification activities.

Life Cycle Phase	Verification Activities
Requirements	• Determine verification approach. • Determine adequacy of requirements. • Generate functional test data. • Determine consistency of design with requirements.
Design	• Determine adequacy of design. • Generate structural and functional test data. • Determine consistency with design
Program (Build)	• Determine adequacy of implementation • Generate structural and functional test data for programs.
Test	• Test application system.
Installation	• Place tested system into production.
Maintenance	• Modify and retest.

Throughout the entire lifecycle, neither development nor verification is a straight-line activity. Modifications or corrections to a structure at one phase will require modifications or re-verification of structures produced during previous phases.

2.0 Verification and Validation Testing Strategies

2.1 Verification Strategies

The Verification Strategies, persons / teams involved in the testing, and the deliverable of that phase of testing is briefed below:

Verification Strategy	Performed By	Explanation	Deliverable
Requirements Reviews	Users, Developers, Test Engineers.	Requirement Review's help in base lining desired requirements to build a system.	Reviewed and approved statement of requirements.
Design Reviews	Designers, Test Engineers	Design Reviews help in validating if the design meets the requirements and build an effective system.	System Design Document, Hardware Design Document.
Code Walkthroughs	Developers, Subject Specialists, Test Engineers.	Code Walkthroughs help in analyzing the coding techniques and if the code is meeting the coding standards	Software ready for initial testing by the developer.
Code Inspections	Developers, Subject Specialists, Test Engineers.	Formal analysis of the program source code to find defects as defined by meeting system	Software ready for testing by the testing team.

		during the Installation at the user place.	
Beta Testing	Users.	Testing of the application after the installation at the client place.	Successfully installed and running application.

3.0 Testing Types

There are two types of testing:

1. Functional or Black Box Testing,
2. Structural or White Box Testing.

Before the Project Management decides on the testing activities to be performed, it should have decided the test type that it is going to follow. If it is the Black Box, then the test cases should be written addressing the functionality of the application. If it is the White Box, then the Test Cases should be written for the internal and functional behavior of the system.

Functional testing ensures that the requirements are properly satisfied by the application system. The functions are those tasks that the system is designed to accomplish.

Structural testing ensures sufficient testing of the implementation of a function.

3.1 White Box Testing

White Box Testing; also know as glass box testing is a testing method where the tester involves in testing the individual software programs using tools, standards etc.

Using white box testing methods, we can derive test cases that
1) Guarantee that all independent paths within a module have been exercised at lease once,
2) Exercise all logical decisions on their true and false sides,
3) Execute all loops at their boundaries and within their operational bounds, and
4) Exercise internal data structures to ensure their validity.

Advantages of White box testing:
1) Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
2) Often, a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.
3) Typographical errors are random.

White Box Testing Types

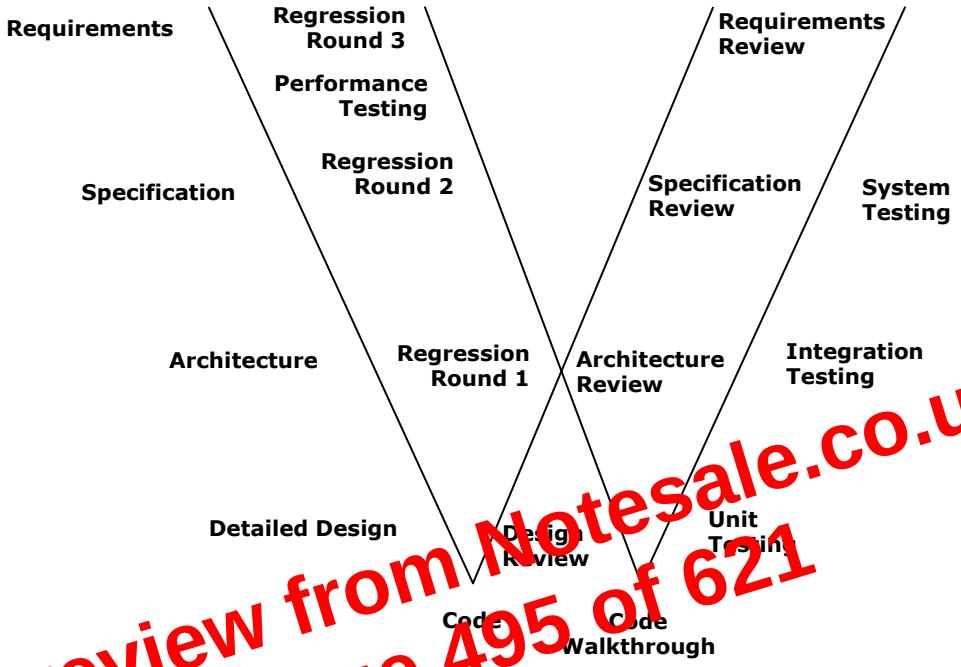
There are various types of White Box Testing. Here in this framework I will address the most common and important types.

3.1.1 Basis Path Testing

Basis path testing is a *white box testing* technique first proposed by Tom McCabe. The Basis path method enables to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test Cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

6.2 The 'W' Model

The following diagram depicts the 'W' model:



The 'W' model depicts that the testing starts from day one of the initiation of the project and continues till the end. The following table will illustrate the phases of activities that happen in the 'W' model:

SDLC Phase	The first 'V'	The second 'V'
1. Requirements	1. Requirements Review	1. Build Test Strategy. 2. Plan for Testing. 3. Acceptance (Beta) Test Scenario Identification.
2. Specification	2. Specification Review	1. System Test Case Generation.
3. Architecture	3. Architecture Review	1. Integration Test Case Generation.
4. Detailed Design	4. Detailed Design Review	1. Unit Test Case Generation.
5. Code	5. Code Walkthrough	1. Execute Unit Tests
		1. Execute Integration Tests.
		1. Regression Round 1.
		1. Execute System Tests.
		1. Regression Round 2.
		1. Performance Tests
		1. Regression Round 3
		1. Performance/Beta Tests

OBJECT ORIENTED TESTING

What is Object-Oriented?

This is an interesting question, answered by many scientists. I will just give a brief of the same. "Objects" are re-usable components. General definition of "Object-Oriented Programming" is that which combines data structures with functions to create re-usable objects.

What are the various Object Oriented Life Cycle Models?

What is mainly required in OOLife Cycle is that there should be iterations between phases. This is very important. One such model which explains the importance of these iterations is the **Fountain Model.**

This **Fountain Model** was proposed by Henderson-Sellers and Edwards in 1990. Various phases in the Fountain Model are as follows:

Requirements Phase

Object-Oriented Analysis Phase
Object-Design Phase
Implementation Phase
Implementation and Integration Phase
Operations Mode
Maintenance

Various phases in Fountain Model overlap:
Requirements and Object Oriented Analysis Phase
Object-Oriented Design, Implementation and Integration Phase
Operations Mode and Maintenance Phases.

Also, each phase will have its own iterations. By adopting the Object-Oriented Application Development, it is scientifically proved that the Maintenance of the software has a tremendous drop. The software is easy to manage and adding new functionality or removing the old is well within controll. This can be achieved without disturbing the overall functionality or other objects. This can help reduce time in software maintenance.

My aim is not to provide informaiton on Object Oriented Application development, but to provide information and techniques as to how to go about the testing of Object Oriented Systems.

Testing of Object Oriented Testing is the same as usual testing when you follow the conventional Black Box testing (of course there will be differences depending on your Test Strategy).

But, otherwise, while testing Object Oriented Systems, we tend to adopt different Test Strategies. This is because, the development cycle is different from the usual cycle(s). Why? This is an interesting question. Why is testing of Object Oriented Systems different? First, let us cover the basics of OOPS.

The Object Oriented Methodology

Let us look at the Object Modelling Technique(OMT) methodology:
Analysis: Starting from the statement of the problem, the analyst buildsa model of the real-world situation showing its important properties.

Object Diagrams
Object Diagrams describe the static structure of a system at a particular time. Whereas a class model describes all possible situations, an object model describes a particular situation. Object diagrams contain the following elements: *Objects* and *Links.*

Use Case Diagrams
Use Case Diagrams describe the functionality of a system and users of the system. These diagrams contain the following elements: *Actors* and *Use Cases.*

Sequence Diagrams
Sequence Diagrams describe interactions among classes. These interactions are modeled as exchange of messages. These diagrams focus on classes and the messages they exchange to accomplish some desired behavior. Sequence diagrams are a type of interaction diagrams. Sequence diagrams contain the following elements: *Class Roles, Lifelines, Activations* and *Messages.*

Collaboration Diagrams
Collaboration Diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes through their associations. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements: *Class Roles, Association Roles, Message Roles.*

Statechart Diagrams
Statechart (or state) diagrams describe the states and responses of a class. Statechart diagrams describe the behavior of a class in response to external stimuli. These diagrams contain the following elements: *States, Transitions.*

Activity Diagrams
Activity diagrams describe the activities of a class. These diagrams are similar to statechart diagrams and use similar conventions, but activity diagrams describe the behavior of a class in response to internal processing rather than external events as in statechart diagram.

Component Diagrams
Component diagrams describe the organization of and dependencies among software implementation components. These diagrams contain components, which represent distributable physical units; including source code, object code, and executable code.

Deployment Diagrams
Deployment diagrams describe the configuration of processing resource elements and the mapping of software implementation components onto them. These diagrams contain components and nodes, which represent processing or computational resources, including computers, printers, etc.

Alternate Flow 4: Actor clicks on <Clear>

Action	Response
1. Actor enters some information in the User ID, Password or Connect field and then clicks on <Clear>	1. Clear the contents in the fields and position the cursor in the User ID field.

Alternate Flow 3: Actor clicks on <Cancel>

Action	Response
1. Actor clicks on <Cancel> button.	1. Close the login screen.

Business Rules

1. When the login screen is initially displayed, the <Login> and <Clear> buttons should be disabled.
2. <Cancel> button should always be enabled.
3. When the actor enters any information in User ID, Password or selects any other database in the list in the Connect To field, then enable <Clear> button.
4. When the actor enters information in User ID and Password fields, then enable <Login> button.
5. The tabbing order should be User ID, Password, Connect To, Login, Password, Clear and Cancel buttons.

The Business Rules which we have addressed towards the end of the use case are for the whole functionality of the use case. If there are any business rules for individual fields we can address them here.

Let us look at another way of writing the above use case. In this format, I would be addressing the Business Rules in a third column in the use case itself.

Main Flow: Login

Action	Response	Business Rule
1. Actor invokes the application.	1. Display login page with User ID, Password, Connect fields and Login, Clear and Cancel buttons.	1. When the login screen is initially displayed, the <Login> and <Clear> buttons should be disabled. 2. <Cancel> button should always be enabled. 3. When the actor enters any information in User ID, Password or selects any other database in the list in the Connect To field, then enable <Clear> button. 4. When the actor enters information in User ID and Password fields, then enable <Login> button. 5. The tabbing order should be User ID, Password, Connect To, Login, Password, Clear and Cancel buttons.
2. Actor enters User ID,	2. Authenticate and display the	

Password and clicks on <Login> button. If actor enters wrong User ID, see alternative flow 1. If actor enters wrong Password, see alternative flow 2. If actor chooses to connect to different database, see alternative flow 3. If actor clicks on <Clear>, see alternative flow 4. If actor clicks on <Cancel>, see alternative flow 5.	home page.	

In this format, the use case might look a bit jazzy, but it is easier to read. The business rules (validation rules) for each step are addressed in the same row itself. In my opinion when you are writing functional use cases then we can use the first format and when we are writing the user interface use cases, then we can use the second format.

Understanding a Use Case

Understanding a use case is nothing big if you know how to read English and have a little bit of reasoning skills.

Let us look at understanding the above written use case itself.

The use case depicts the behavior of the system. When you are reading the above use case you read each step horizontally.

Look at step 1 written in the first type of use case:

Main Flow: Login

Action	**Response**
1. Actor invokes the application.	1. Display login page with User ID, Password, Connect fields along with Login, Clear and Cancel buttons.

Here **Action**, is something which is performed by the user; and **Response** is the application/system response to the action performed by the user.

Thus, you can understand that when the actor performs an action of invoking the application, the login screen is displayed with the mentioned fields and information.

In the first type of writing use case, the **Business Rules** have been addressed below after all the flows.

In the second type of writing use case, the same **Business Rules** have been addressed in a third column. Business Rules are nothing but special conditions which have to be satisfied by the response of the system.

Testing Use Case's

Testing Use Case's calls for a through understanding the concept of use case, how to read it and how do you derive test cases from it.

I will explain briefly a methodology I follow when deriving test cases and scenarios from Use Cases.

- For each actor involved in the use case, identify the possible sequence of interactions between the system and the actor, and select those that are likely to produce different system behaviors.
- For each input data coming from an actor to the use case, or output generated from the use case to an actor, identify the equivalence classes – sets of values which are likely to produce equivalent behavior.
- Identify Test Cases based on Range Value Analysis and Error Guessing.
- Each test case represents one combination of values from each of the below: objects, actor interactions, input / output data.
- Based on the above analysis, produce a use case test table (scenario) for each use case.
- Select suitable combinations of the table entries to generate test case specifications.
- For each test table, identify the test cases (success or exception) tested.
- Ensure all extensions are tested at least once.
- Maintain a **Use Case Prioritization Table** for the use cases for better coverage as follows:

Use Case No	Use Case	Risk	Frequency	Criticality	Priority

The **Risk** column in the table describes the risk involved in the Use Case.
The **Frequency** column in the table describes how frequently the Use Case occurs in the system.
The **Criticality** column in the table describes the importance of the Use Case in the system.
The **Priority** column in the table describes the priority for testing by taking the priority of the use case from the developer.

- Some use cases might have to be tested more thoroughly based on the frequency of use, criticality and the risk factors.
- Test the most used parts of the program over a wider range of inputs than lesser user portions to ensure better coverage.
- Test more heavily those parts of the system that pose the highest risk to the project to ensure that the most harmful faults are identified as soon as possible.
- The most risk factors such as change in functionality, performance shortfall or change in technology should be bared in mind.
- Test the use cases more thoroughly, which have impact on the operation of the system.
- The pre-conditions have to be taken into consideration before assuming the testing of the use case. Make test cases for the failure of the pre-conditions and test for the functionality of the use case.
- The post-conditions speak about the reference to other use cases from the use case you are testing. Make test cases for checking if the functionality from the current use case to the use case to which the functionality should be flowing is working properly.
- The business rules should be incorporated and tested at the place where appropriately where they would be acting in the use case.
- Maintain a **Test Coverage Matrix** for each use case. The following format can be used:

UC No.	UC Name	Flow	TC No's	No. of TC's	Tested	Status

In the above table:

- The **UC No.** column describes the Use Case Number.
- The **UC Name** column describes the Use Case Name.
- The **Flow** column describes the flow applicable: Typical Flow, Alternate Flow 1, Alternate Flow 2, etc.
- The **TC No's** column describes the start and end test case numbers for the flow.
- The **No. of TC's** column describes the total number of test cases written.
- The **Tested** column describes if the flow is tested or not.
- The **Status** column describes the status of the set of test cases, if they have passed or failed.

Testers Dictionary

Alpha Test: Alpha testing happens at the development site just before the roll out of the application to the customer. Alpha tests are conducted replicating the live environment where the application would be installed and running

Behavioral Tests: Behavioral Tests are often used to find bugs in the high-level operations, at the levels of features, operational profiles, and customer scenarios.

Beta Tests: Beta testing happens at the actual place of installation in the live environment. Here the users play the role of testers.

Black Box Tests: Black Box tests aim at testing the functionality of the application basing on the Requirements and Design documents.

Defect: Any deviation in the working of the application that is not mentioned in any documents in SDLC can be termed as a defect.

Defect Density: Defect Density is the number of defects raised to the size of the program.

Defect Report: A report, which lists the defects, noticed in the application.

Grey Box Tests: Grey Box tests are a combination of Black Box and White Box tests.

Installation Tests: Installation tests aim at testing the installation of an application. Testing of application for installing on a variety of hardware and software requirements is termed as installation.

Integration Tests: Testing two or more programs, which together accomplish a particular task. Also, Integration Tests aim at testing the binding and communication between programs.

Load Tests: Load testing aims at testing the maximum load the application can take basing on the requirements. Load can be classified into number of users of the system, load on the database etc.

Performance Tests: Performance tests are coupled with stress testing and usually require both hardware and software instrumentation.

Quality Control
Relates to a specific product or service.
Verifies whether specific attributes are in, or are not in, a specific product or service.
Identifies defects for the primary purpose of correction defects. Is the responsibility of team/workers.
Is concerned with a specific product.

Quality Assurance
Helps establish process.
Sets up measurements programs to evaluate processes.
Identifies weakness in processes and improves them.
Is management responsibility, frequently performed by staff function.
Is concerned with all of the products that will ever be produced by a process.
Is sometimes called quality control over Quality Control because it evaluates whether quality is working.

6.4 Defect Reporting

When defects are found, the testers will complete a defect report on the defect tracking system. The defect tracking Systems is accessible by testers, developers & all members of the project team. When a defect has been fixed or more information is needed, the developer will change the status of the defect to indicate the current state. Once a defect is verified as FIXED by the testers, the testers will close the defect report.

Article X. 7. Functions To Be Tested

The following is a list of functions that will be tested:

- Add/update employee information
- Search / Lookup employee information
- Escape to return to Main Menu
- Security features
- Scaling to 700 employee records
- Error messages
- Report Printing
- Creation of payroll file
- Transfer of payroll file to the mainframe
- Screen mappings (GUI flow). Includes default settings
- FICA Calculation
- State Tax Calculation
- Federal Tax Calculation
- Gross pay Calculation
- Net pay Calculation
- Sick Leave Balance Calculation
- Annual Leave Balance Calculation

A Requirements Validation Matrix will "map" the test cases back to the requirements. See Deliverables.

Article XI. 8. Resources and Responsibilities

The Test Lead and Project Manager will determine when system test will start and end. The Test lead will also be responsible for coordinating schedules, equipment, & tools for the testers as well as writing/updating the Test Plan, Weekly Test Status reports and Final Test Summary report. The testers will be responsible for writing the test cases and executing the tests. With the help of the Test Lead, the Payroll Department Manager and Payroll clerks will be responsible for the Beta and User Acceptance tests.

8.1. Resources

The test team will consist of:

- A Project Manager
- A Test Lead
- 5 Testers
- The Payroll Department Manager
- 5 Payroll Clerks

Fault Tolerance

The ability of a system or component to continue normal operation despite the presence of hardware or software faults.

Flaw hypothesis methodology

A systems analysis and penetration technique in which specifications and documentation for the system are analyzed and then flaws in the system are hypothesized. The list of hypothesized flaws is then prioritized on the basis of the estimated probability that a flaw exists and, assuming a flaw does exist, on the ease of exploiting it, and on the extent of control or compromise it would provide. The prioritized list is used to direct a penetration attack against the system.

Formal

Expressed in a restricted syntax language with defined semantics based on well-established mathematical concepts.

Formal specification

(I) A specification of hardware or software functionality in a computer-readable language; usually a precise mathematical description of the behavior of the system with the aim of providing a correctness proof

Format

The organization of information according to preset specifications (usually for computer processing) [syn: formatting, data format, data formatting]

Glossary

A glossary is an alphabetical list of words or expressions and the special or technical meanings that they have in a particular book, subject, or activity

Hacker

A person who enjoys exploring the details of computers and how to stretch their capabilities. A malicious or inquisitive meddler who tries to discover information by poking around. A person who enjoys learning the details of programming systems and how to stretch their capabilities, as opposed to most users who prefer to learn on the minimum necessary.

Implementation under test, IUT

The particular portion of equipment which is to be studied for testing. The implementation may include one or more protocols.

Implementation vulnerability

A vulnerability resulting from an error made in the software or hardware implementation of a satisfactory design.

Input

A variable (whether stored within a component or outside it) that is read by the component.

Instrument

1. A tool or device that is used to do a particular task. 2. A device that is used for making measurements of something.

In software and system testing, to install or insert devices or instructions into hardware or software to monitor the operation of a system or component.

Instrumentation

Instrumentation is a group or collection of instruments, usually ones that are part of the same machine.

Devices or instructions installed or inserted into hardware or software to monitor the operation of a system or component.

The insertion of additional code into the program in order to collect information about program behaviour during program execution.

(NBS) The insertion of additional code into a program in order to collect information about program behavior during program execution. Useful for dynamic analysis techniques such as assertion checking, coverage analysis, tuning.

Integrity

Assuring information will not be accidentally or maliciously altered or destroyed.

Sound, unimpaired or perfect condition.

Interface

(1) A shared boundary across which information is passed. (2) A Hardware or software component that connects two or more other components for the purpose of passing information from one to the other. (3) To connect two or more components for the purpose of passing information from one to the other. (4) To serve as a connecting or connected component as in (2).

(1) (ISO) A shared boundary between two functional units, defined by functional characteristics, common physical interconnection characteristics, signal characteristics, and other characteristics, as appropriate. The concept involves the specification of the connection of two devices having different functions. (2) A point of communication between two or more processes, persons, or other physical entities. (3) A peripheral device which permits two or more devices to communicate.

Interface testing

Testing conducted to evaluate whether systems or components pass data and control correctly to each other.

Integration testing where the interfaces between system components are tested.

Language

Any means of conveying or communicating ideas; specifically, human speech; the expression of ideas by the voice; sounds, expressive of thought, articulated by the organs of the throat and mouth.

Least privilege

Feature of a system in which operations are granted the fewest permissions possible in order to perform their tasks.

The principle that requires that each subject be granted the most restrictive set of privileges needed for the performance of authorized tasks. The application of this principle limits the damage that can result from accident, error, or unauthorized use.

Liability

Liability for something such as debt or crime is the legal responsibility for it; a technical term in law.

Malicious code, malicious logic, malware

(I) Hardware, software, or firmware that is intentionally included or inserted in a system for a harmful purpose. (See: logic bomb, Trojan horse, virus, worm.)

Hardware, software, or firmware that is intentionally included in a system for an unauthorized purpose; e.g., a Trojan horse.

Mutation analysis

checklist of historically common programming errors, and analyzing its compliance with coding standards.

Code Walkthrough: A formal testing technique where source code is traced by a group with a small set of test cases, while the state of program variables is manually monitored, to analyze the programmer's logic and assumptions.

Compatibility Testing: Testing whether software is compatible with other elements of a system with which it should operate, e.g. browsers, Operating Systems, or hardware.

Component: A minimal software item for which a separate specification is available.

Component Testing: See Unit Testing.

Concurrency Testing: Multi-user testing geared towards determining the effects of accessing the same application code, module or database records. Identifies and measures the level of locking, deadlocking and use of single-threaded code and locking semaphores.

Conformance Testing: The process of testing that an implementation conforms to the specification on which it is based. Usually applied to testing conformance to a formal standard.

Context Driven Testing: The context-driven testing is flavor of Agile Testing that advocates continuous and creative evaluation of testing opportunities in light of the potential information revealed and the value of that information to the organization right now. It can be defined as testing driven by an understanding of the environment, culture, and intended use of software. For example, the testing approach for life-critical medical equipment software would be completely different than that for a low-cost computer game.

Conversion Testing: Testing of programs or procedures used to convert data from existing systems for use in replacement systems.

Cyclomatic Complexity: A measure of the logical complexity of an algorithm, used in white-box testing.

Data Flow Diagram: A modeling notation that represents a functional decomposition of a system.

Data Driven Testing: Testing in which the action of a test case is parameterized by externally defined data values, maintained as a file or spreadsheet. A common technique in Automated Testing.

Dependency Testing: Examines an application's requirements for pre-existing software, initial states and configuration in order to maintain proper functionality.

Depth Testing: A test that exercises a feature of a product in full detail.

Dynamic Testing: Testing software through executing it. See also Static Testing.

Emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Endurance Testing: Checks for memory leaks or other problems that may occur with prolonged execution.

Network Sensitivity Tests: Network sensitivity tests are tests that set up scenarios of varying types of network activity (traffic, error rates...), and then measure the impact of that traffic on various applications that are bandwidth dependant. Very 'chatty' applications can appear to be more prone to response time degradation under certain conditions than other applications that actually use more bandwidth. For example, some applications may degrade to unacceptable levels of response time when a certain pattern of network traffic uses 50% of available bandwidth, while other applications are virtually un-changed in response time even with 85% of available bandwidth consumed elsewhere.

This is a particularly important test for deployment of a time critical application over a WAN.

Negative Testing: Testing aimed at showing software does not work. Also known as "test to fail".

N+1 Testing: A variation of Regression Testing. Testing conducted with multiple cycles in which errors found in test cycle N are resolved and the solution is retested in test cycle N+1. The cycles are typically repeated until the solution reaches a steady state and there are no errors. See also Regression Testing.

Path Testing: Testing in which all paths in the program source code are tested at least once.

Performance Testing: Testing conducted to evaluate the compliance of a system or component with specified performance requirements. Often this is performed using an automated test tool to simulate large number of users. Also know as "Load Testing".

Performance Tests are tests that determine end to end timing (benchmarking) of various time critical business processes and transactions while the system is under low load, but with a production sized database. This sets 'best possible' performance expectation under a given configuration of infrastructure. It also highlights very early in the testing process if changes need to be made before load testing should be undertaken. For example, a customer search may take 15 seconds in a full sized database if indexes had not been applied correctly, or if an SQL 'hint' was incorporated in a statement that had been optimized with a much smaller database. Such performance testing would highlight such a slow customer search transaction, which could be remediate prior to a full end to end load test.

Positive Testing: Testing aimed at showing software works. Also known as "test to pass".

Protocol Tests: Protocol tests involve the mechanisms used in an application, rather than the applications themselves. For example, a protocol test of a web server may will involve a number of HTTP interactions that would typically occur if a web browser were to interact with a web server - but the test would not be done using a web browser. LoadRunner is usually used to drive load into a system using VUGen at a protocol level, so that a small number of computers (Load Generators) can be used to simulate many thousands of users.

Quality Assurance: All those planned or systematic actions necessary to provide adequate confidence that a product or service is of the type and quality needed and expected by the customer.

Each test can be quite simple, For example, a test ensuring that 500 concurrent (idle) sessions can be maintained by Web Servers and related equipment should be executed prior to a full 500 user end to end performance test, as a configuration file somewhere in the system may limit the number of users to less than 500. It is much easier to identify such a configuration issue in a Targeted Infrastructure Test than in a full end to end test.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Testing:

The process of exercising software to verify that it satisfies specified requirements and to detect errors.

The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item.

The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Test Bed: An execution environment configured for testing. May consist of specific hardware, OS, network topology, configuration of the product under test, other application or system software, etc. The Test Plan for a project should enumerate the test beds(s) to be used.

Test Case:

Test Case is a commonly used term for a specific test. This is usually the smallest unit of testing. A Test Case will consist of information such as requirements testing, test steps, verification steps, prerequisites, outputs, test environment, etc.

A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Driven Development: Testing methodology associated with Agile Programming in which every chunk of code is covered by unit tests, which must all pass all the time, in an effort to eliminate unit-level and regression bugs during development. Practitioners of TDD write a lot of tests, i.e. an equal number of lines of test code to the size of the production code.

Test Driver: A program or test tool used to execute a tests. Also known as a Test Harness.

Test Environment: The hardware and software environment in which tests will be run, and any other software with which the software under test interacts when under test including stubs and test drivers.

Test First Design: Test-first design is one of the mandatory practices of Extreme Programming (XP).It requires that programmers do not write any production code until they have first written a unit test.

Test Harness: A program or test tool used to execute a test. Also known as a Test Driver.

TEST AUTOMATION EFFORT ESTIMATION

Babu Narayanan

Recommendations:

1. Neither group test case steps too close, nor wide for labeling the complexity. Be aware that the pre-script development effort for each test script is considerable as the following activities are time-consuming operations:-
   1) Executing test case manually before scripting for confirming the successful operation.
   2) Test data selection and/or generation for the script.
   3) Script template creation (like header information, comments for steps, identifying the right reusable to be used from the repository and so on.)

   These efforts are highly based on the number of steps in the test case. Note that if test case varies by fewer steps, then this effort does not deviate much but incase it varies by many steps even this effort widely differs.

2. Also another factor in determining the complexity is the functionality repetition. If the test case is Complex by steps but the functionality is same as the other test case then this can be labeled as 'Medium or Simple' (based on the judgment).
3. If the test case steps count are more than upper control limit (~ 25 in this case) value then those additional steps need to be considered as another test case. For example, the TC - 06 containing 30 steps shall be labeled as 1 complex + 1 simple (30-25) test cases.

If the test case is marked as 'Complex' instead of 'Medium', understand that your efforts shoot up and hurts your customer. On other way of miscalculation, it hurts us. There by, this 'complexity grouping' is more of logical workout with data as input.

5. Scripting Effort Estimation

SL.NO	SUB COMPONENT	ESTIMATED EFFORT			REMARKS
		Simple (<8 steps)	Medium (8-16 steps)	Complex (17-25 steps)	
1	**Pre–Script Development**				
a	Test Case execution (Manual)				For 1 iteration (assuming scripter knows navigation)
b	Test data selection				For one data set (valid/invalid/erratic kind)
c	Script Template creation				Can use script template generation utility to avoid this.
d	Identify the reguired reusable				Assuming proper reusable traceability matrix presence.
2	**Script Development**				
a	Application map creation				Assuming the no of objects = number of actions
b	Base scripting				Normally all these go hand-in-hand. Separated for analysis & reasoning.
c	Addendum/Exception handling				
d	Implement framework elements				
3	**Script Execution**				
a	Script execution				For n iterations (~ average iteration count)
b	Verification & Reporting				Assuming there will minimal defect reporting.
	Total Effort per script				

Keyword driven

This total effort would vary if you choose key-word driven methodology but at the same time, the effort of building framework will be high (for initial design and scripting).

- Do not use keyword driven approach for small projects.
- These efforts may differ based on the above discussed (section 2) factors. Suggest you to perform PoC for 2 scripts from each class to confirm.
- The negative test cases normally consume additional script efforts as the pattern changes.

Overall effort calculation may have the following components:-

1. Test Requirement gathering & Analysis
2. Framework design and development
3. Test Case development (incase the available manual test cases not compatible)
4. Script Development
5. Integration Testing and Baseline.
6. Test Management.

All these components shall include review (1/2 cycles).

b) Ease of scheduling

This strategy saves scheduling nightmares, especially for big projects. Team members are aware of their test execution responsibilities in this scenario.

However, this strategy might not be the most efficient in all circumstances. Following are some of the factors which should also be considered together with the ones identified above:

a) Errors might be overlooked

This occurs quite frequently where the test case owner fails to capture various possible scenarios. This includes missing test cases, test cases based on misunderstood requirements, and incorrect test case design(e.g., incorrect test case naming conventions). These errors might not be caught if test case owners are executing the scripts written by them. This becomes even more significant if there are no internal or external test case reviews where these errors might be detected before the start of test execution cycle.

b) Casual approach

Test cases might not be written with same considerations as when others would be executing them. Test cases, generally speaking, should be written with following considerations:

-Write test cases such that even a lay man should be easily able to understand and execute them.

-Replace yourself with the person who might be executing it. Think what you would expect from someone if you were executing their scripts. Make the test scripts unambiguous, self-contained, and self-explanatory.

Following is a conversation between two test team members, who are in the middle of a test case creation cycle:

“Use case XYZ seems pretty complex”, Peter said.

“Yes, and this has been assigned to me. There are a lot of different possible scenarios for this use case”, John replied.

“So, you might end up with a lot of test cases for this use case”, Peter replied.

“Well….since I will be executing this use case, I will make sure to cover most of the scenarios while testing. However, I might not document all of them. I am very well familiar with this use case and don’t think I need to document everything I test. More so, I can’t spend too much time on this because of scheduling constraints. As long as I test it well, I think I am OK”, John said.

1 Introduction

Often testers encounter many Windows or web pages in an application. And each of these Windows or web pages could have many objects within them. And each object would have unique characteristics associated with them. (The typical Objects are Radio buttons, Push buttons, Dropdown lists, Edit fields, Check boxes etc)
While designing test cases one ascertain each and every object state in detail in order to cover the Functional Test cases along with the Error handling.

This paper intends to simplify the whole process of preparing test data with the aid of a Microsoft Excel spread sheet. It also covers the basic concepts of BVA and Equivalence partitioning techniques.

2. Similarly, Test Step**=C2&", "&F2&" and click insert order button"** would mean concatenate "C2 contents i.e., Action1", ", ", with "F2 Contents i.e., Action2" and "and click insert order button".

4 Summary

1. The test data table could be readily used for Data Driven testing and Key word driven testing for a future usage.
2. This paper would be useful while preparing the test data for **exhaustive testing** or **mission critical applications** (for example, Banking or military) where every aspect of the application requires to be tested.
3. The formulas that are required to arrive at the various combinations of data for different objects are as follows :

a. Radio button or Check box could have only two states either ON or OFF.

So, if a window has three Radio buttons or check boxes in it, one could have 8 unique combinations of data.

(Formula would be "no. of states" ^ "no. of variables" i.e., 2^3 = 8)

b. An edit field could have three states such as Blank, Valid data, Invalid data, then one could have 9 unique combinations of data.

(Formula would be "no. states" ^ "no. of variables" i.e., 3^2 = 9)

A dropdown list could have two or more states depending upon the values it has, for instance blank, value1, value2, value3, value4, then one could have 25 unique combinations of data.

(Formula would be "no. states" ^ no. of variables" i.e., 5^2 = 25)

c. In case when a window has 5 dropdown lists (each dropdown list has 3 values) and 2 edit fields (each edit field has 3 states), then one could have 2187 unique combinations of data.

(F1=Formula for dropdown lists alone would be "no. states" ^ "no. of variables" i.e., 3^5 = 243

F2=Formula for edit fields alone would be "no. states" ^ "no. of variables" i.e., 3^2 = 9

However, such diagrams can help testers to ascertain the functionality quickly and effectively. Given below are some of the data models and how they can be cast to suit testing needs.

ER Diagram approach

An Entity-Relationship diagram shows entities and their relationships. The ER diagram relates to business data analysis and data base design.

There are three basic elements in ER models:

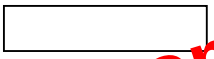
1. **Entities** are the "things" about which we seek information.
2. **Attributes** are the data we collect about the entities.
3. **Relationships** provide the structure needed to draw information from multiple entities.

Before you draft an ERD for a screen, ask yourself these questions:

1. What/Who are the entities?
2. Is there more than one entity? Is there any relationship between these entities?
3. Does the entity have relationships other than the existing entities on the screen?
4. What are the qualities of the entities?
5. Cardinality of the entities (many-to-one, one-to-many)
6. Are 2 screens connected via entities? Can this be indicated via the ER diagram?
7. Is there additional information that cannot be captured through the ER diagram?

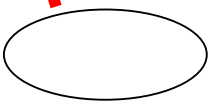
An ER Diagram uses the following notation for its 3 basic elements:

Entity -



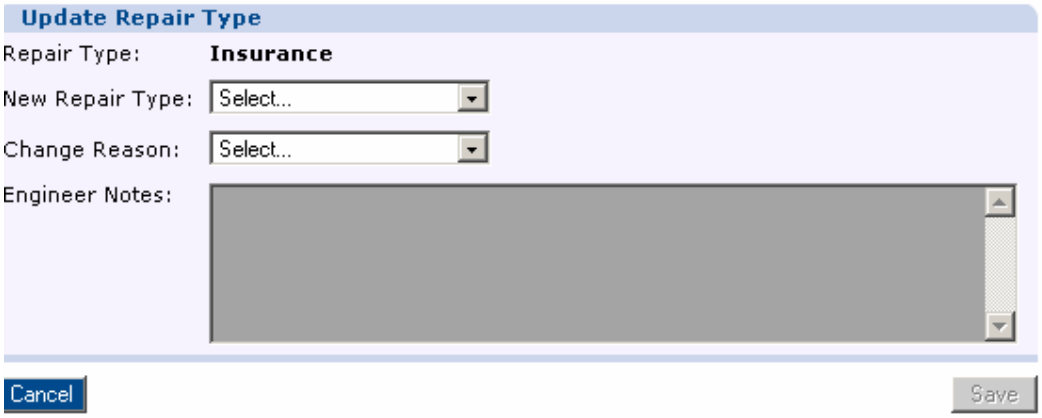
Relationship -

Attribute -



Example ER Diagram -

The screen given was named as "Update Repair Type" as seen below. The screen requires the user to update the repair type while entering a valid reason and adding notes. User can choose to Cancel or Save the task. (I gathered this by fiddling with the screen.)



1. **Define Entities -**

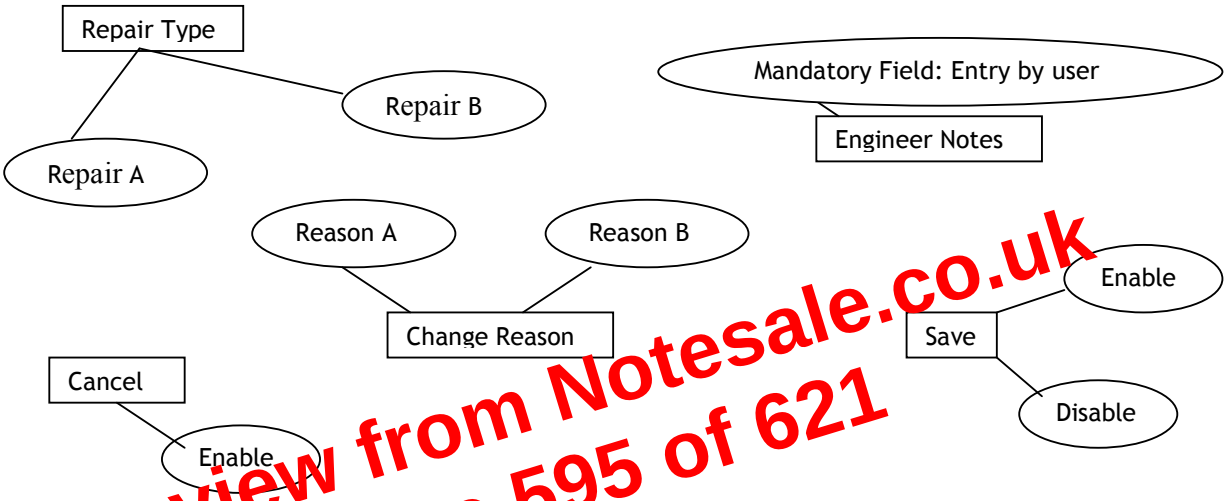
The changing entities here are "New Repair Type", "Change Reason" and "Engineer Notes", with Save and Cancel button being the other entities.

2. **Define Attributes for the Entities -**

Each of those entities has attributes. Based on the screen data availability, I determined the following.

New Repair Type- Repair A, Repair B
Change Reason – Change Reason A, Change Reason B
Engineer Notes- Entry mandatory by user.
Cancel- Enabled
Save- Enabled and Disabled

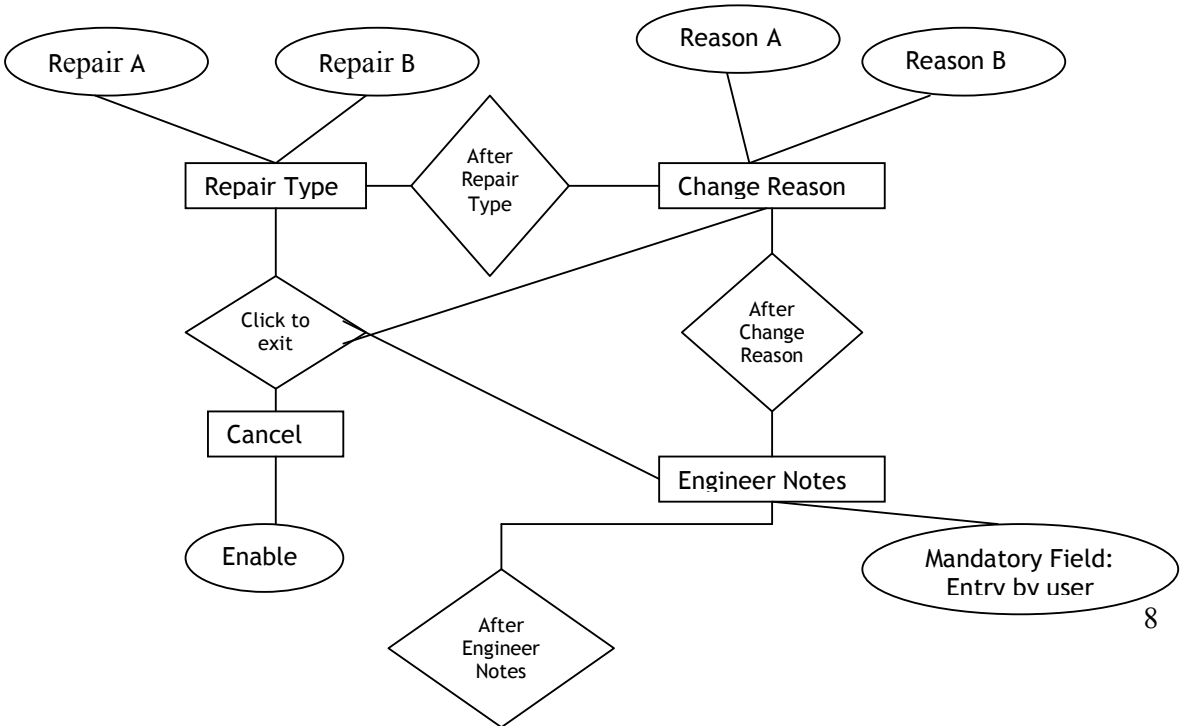
Based on the input I gathered so far, I made the following ER Diagram



3. **Define Relationships -**

Define relationships between each entity. For e.g. in this example, only after I have selected the repair type, can I select the change reason. Engineer Notes can be entered any time. Save button only activates after I have entered all the fields. Cancel button is available for cancellation at any point of time.

My entity diagram could hence look like this -



Note here that, even though the Entity "Save" has 2 attributes, it has not been mentioned in the complete E-R diagram, because it becomes implicit when one studies the diagram.

If the ER diagram gets complex in terms of relationships or attributes, it is advisable to draw 2-3 ER diagrams for that screen. You can base any amount of information in the diagram.
For e.g., if there is a database update, you can indicate database as an entity and indicate the various tables as its attributes and indicate through the relationship when it is likely to get updated.

Activity Model Diagram /UML

The UML enables you to model many different facets of your business, from the actual business and its processes to IT functions such as database design, application architectures, hardware designs, and much more.
You can use the different types of UML diagrams to create various types of models to suit your needs. The model types and their usage are:

Model Type	Model Usage
Business	Business process, workflow, organization
Requirements	Requirements capture and organization
Architecture	High level understanding of the system being built, interaction between different software systems, communicate system design to developers
Application	Architecture of the lower-level designs inside the system itself
Database	Design the structure of the database and how it will interact with the application(s).

The UML contains two different basic diagram types:
Structure diagrams and Behavior diagrams.

An overview of some of these is listed below.

Structure diagrams depict the static structure of the elements in your system. The various structure diagrams are as follows: -

Class diagrams are the most common diagrams used in UML modeling. They represent the static things that exist in your system, their structure, and their interrelationships. They are typically used to depict the logical and physical design of the system.

Component diagrams show the organization and dependencies among a set of components. They show a system as it is implemented and how the pieces inside the system work together.

Object diagrams show the relationships between a set of objects in the system. They show a snapshot of the system at a point in time.

- The organization's standard software process and the projects defined software processes are improved continuously.

These defined standards give the organization a commitment to perform because:

- The organization follows a written policy for implementing software process improvements.
- Senior management sponsors the organization's activities for software process improvement.

The ability of the organization to perform transpires because:

- Adequate resources and funding are provided for software process improvement activities.
- Software managers receive required training in software process improvement.
- The managers and technical staff of the software engineering group and other software-related groups receive required training in software process improvement.
- Senior management receives required training in software process improvement.

The **Process Area Activities** performed include:

- A software process improvement program is established which empowers the members of the organization to improve the processes of the organization.
- The group responsible for the organization's software process activities coordinates the software process improvement activities.
- The organization develops and maintains a plan for software process improvement according to a documented procedure.
- The software process improvement activities are performed in accordance with the software process improvement plan.
- Software process improvement proposals are handled according to a documented procedure.
- Members of the organization actively participate in teams to develop software process improvements for assigned process areas.
- Where appropriate, the software process improvements are installed on a pilot basis to determine their benefits and effectiveness before they are introduced into normal practice.
- When the decision is made to transfer a software process improvement into normal practice, the improvement is implemented according to a documented procedure.
- Records of software process improvement activities are maintained.