

MANUAL TESTING

JAGAN MOHAN J

Jagan

Software Engineering

Table of Contents

Chapter 1	Introduction	
1.1 Scope		1
1.1.1 The emergence of software engineering		1
1.1.2 The term software engineering		3
1.2 Quality attributes of software products		3
1.2.1 Software quality attributes		3
1.2.2 The importance of quality criteria		8
1.2.3 The effects of quality criteria on each other		9
1.2.4 Quality assurance measures		10
1.3 The phases of a software project		10
1.3.1 The classical sequential software life-cycle model		10
1.3.2 The waterfall model		13
1.3.3 The prototyping-oriented life-cycle model		14
1.3.4 The spiral model		15
1.3.5 The object-oriented life-cycle model		16
1.3.6 The object-and prototyping-oriented life-cycle model		18
References and selected reading		19

Chapter 2	Software Specification	
2.1 Structure and contents of the requirements definition		21
2.2 Quality criteria for requirements definition		24
2.3 Fundamental problems in defining requirements		24
2.4 Algebraic specification		25
2.5 Model-based specification		27
References and selected reading		32

Chapter 3	Software Design	
------------------	------------------------	--

3.1 Design techniques	34
3.1.1 Top-down design	35
3.1.2 Bottom-up design	35
3.1.3 Systems design	36
3.1.4 Design decomposition	36
3.2 User interface design	37
3.3 Function-oriented design	42
3.4 Object-oriented design	44
3.4.1 The Abbott Method	44
3.4.2 Design of class hierarchies	45
3.4.3 Generalization	47
References and selected reading	48

Chapter 4 Implementation	
4.1 Programming environments	50
4.2 Programming style	55
4.3 Portability and reuse	58
4.3.1 Software portability	58
4.3.2 Machine Architecture dependencies	60
4.3.3 Operating system dependencies	61
4.3.4 Software reuse	64
4.4 Computer-aided software engineering	67
4.4.1 CASE workbenches	67
4.4.2 Text editing systems	69
4.4.3 Language processing systems	70
4.5 Incremental implementation	73
<u>References and selected reading</u>	74

Chapter 5 Software Verification	
5.1 Test methods	75
5.1.1 Verification of algorithms	78
5.1.2 Static program analysis	79

5.1.3 Dynamic testing	80
5.1.4 Black-box and white-box testing	80
5.1.5 Top-down and bottom-up testing	82
5.2 Mathematical program verification	83
5.3 Debugging	86
References and selected reading	87

Chapter 6	Documentation
6.1 User documentation	90
6.2 System documentation	93
6.3 Document quality	94
6.4 Document maintenance	97
6.5 Document portability	98
References and selected reading	98

Chapter 7	Project Management
7.1 The goal and tasks of project management	101
7.2 Difficulties in project management	104
7.3 Cost estimation	105
7.4 Project organization	107
7.4.1 Hierarchical organizational model	107
7.4.2 The chief programmer team	108
7.5 Software maintenance	109
7.5.1 Maintenance costs	111
7.5.2 System restructuring	116
7.5.3 Program evolution dynamics	117

Objectives

The objectives of this chapter are to define what is meant by software engineering, to discuss which attributes are characteristic of software products and which measures need to be undertaken to assure quality. The notion of software process models is introduced and discussed.

Contents

1.1 Scope

1.1.1 The emergence of software engineering

1.1.2 The term software engineering

1.2 Quality attributes of software products

1.2.1 Software quality attributes

1.2.2 The importance of quality criteria

1.2.3 The effects of quality criteria on each other

1.2.4 Quality assurance measures

1.3 The phases of a software project

1.3.1 The classical sequential software life-cycle model

1.3.2 The waterfall model

1.3.3 The prototyping-oriented life-cycle model

1.3.4 The spiral model

1.3.5 The object-oriented life-cycle model

1.3.6 The object and prototyping-oriented life-cycle model

References and selected reading

1.1 Scope

1.1.1 The emergence of software engineering

Many persons work on producing software systems for many users.

The task description and the requirements frequently change even during the program design phase, and continue to change even after the software system has long since been in use.

The major problems encountered in development of large software systems were:

- Correctness
- Efficiency
- Mastery of complexity
- Interface specification
- Reliability
- Flexibility
- Documentation
- Maintainability
- Project organization.

Inadequate theoretical foundation and too few methodological aids were known in both the technical and the organizational realm.

Programmers' qualifications did not suffice to adequately solve the problems.

Software crisis (1965):

The growing economic importance of software production and the enormous expansion of the data processing branch lead to the demand for improved *programming techniques* becoming an increasing focus of research interests in the field of computer science.

Software engineering conferences sponsored by NATO:

- Garmisch, Germany, in 1968 [Naur 1969]
- Rome in 1969 [Buxton 1969].

Contents of the Conferences:

- Programs were defined as *industrial products*.
- The challenge was issued: to move away from the *art of programming* and toward an *engineering approach* to software development.
- Application of scientific approaches throughout software production as an integrated process.
- Research highlights:
 - Specification,
 - Methodical program development,

- Structuring of software systems,
- Reusability of software components,
- Project organization,
- Quality assurance,
- Documentation,
- Requirements for software development tools,
- Automatic generation of software.

1.1.2 The term software engineering

It will be clear that the term *software engineering* involves many difference issues - all are crucial to the success of large-scale software development. The topics covered in this course are intended to address all these different issues.

Boehm [Boehm 1979]:

Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Dennis [Dennis 1975]:

Software engineering is the application of principles, skills, and art to the design and construction of programs and systems of programs.

Parnas [Parnas 1974]:

Software engineering is programming under at last one of the following two conditions:

- (1) *More than one person is involved in the construction and/or use of the programs*
- (2) *More than one version of the program will be produced*

Fairley [Fairley 1985]:

Software engineering is the technological and managerial discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Sommerville [Sommerville 1989]:

Software Engineering is concerned with building software systems which are large than would normally be tackled by a single individual, uses engineering principles in the development of these systems and is made up of both technical and non-technical aspects.

Pomberger and Blaschek [Pomberger 1996]:

Software engineering is the practical application of scientific knowledge for the economical production and use of high-quality software.

1.2 Quality attributes of software products

1.2.1 Software quality attributes

Software quality is a broad and important field of software engineering.

Addressed by standardization bodies:

- ISO
- ANSI
- IEEE
- ...

1 Software quality attributes (see Figure 1.1)

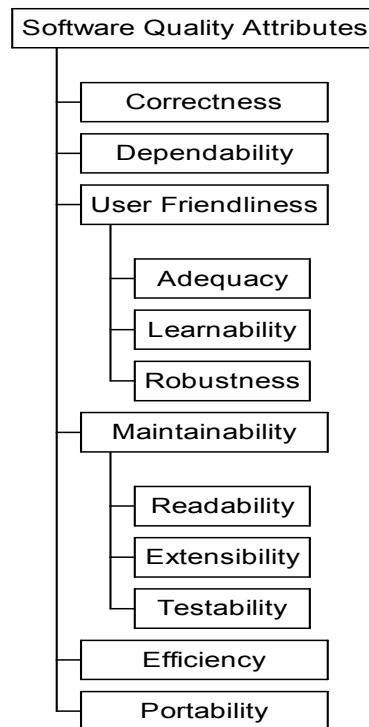


Figure 1.1 Software quality attributes

Correctness

The software system's fulfillment of the specifications underlying its development. The correctness of a software system refers to:

- Agreement of *program code* with *specifications*
- Independence of the actual application of the software system.

The correctness of a program becomes especially critical when it is embedded in a complex software system.

Example 1.1 Let p be the probability that an individual program is correct; then the probability P of correctness of the software system consisting of n programs is computed as $P = p^n$.

If n is very large, then p must be nearly 1 in order to allow P to deviate significantly from 0 [Dijkstra 1972b].

Reliability

Reliability of a software system derives from

- Correctness, and
- Availability.

The behavior over time for the fulfillment of a given specification depends on the reliability of the software system.

Reliability of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).

A software system can be seen as reliable if this test produces a low *error rate* (i.e., the probability that an error will occur in a specified time interval.)

The error rate depends on the frequency of inputs and on the probability that an individual input will lead to an error.

User friendliness:

- Adequacy
- Learnability
- Robustness

Adequacy

Factors for the requirement of **Adequacy**:

1. The input required of the user should be limited to only what is necessary. The software system should expect information only if it is necessary for the functions that the user wishes to carry out. The software system should enable flexible data input on the part of the user and should carry out plausibility checks on the input. In dialog-driven software systems, we vest particular importance in the uniformity, clarity and simplicity of the dialogs.
2. The performance offered by the software system should be adapted to the wishes of the user with the consideration given to extensibility; i.e., the functions should be limited to these in the specification.
3. The results produced by the software system:

The results that a software system delivers should be output in a clear and well-structured form and be easy to interpret. The software system should afford the user flexibility with respect to the scope, the degree of detail, and the form of presentation of the results. Error messages must be provided in a form that is comprehensible for the user.

Learnability

Learnability of a software system depends on:

- The design of user interfaces
- The clarity and the simplicity of the user instructions (tutorial or user manual).

The user interface should present information as close to reality as possible and permit efficient utilization of the software's failures.

The user manual should be structured clearly and simply and be free of all dead weight. It should explain to the user what the software system should do, how the individual functions are activated, what relationships exist between functions, and which exceptions might arise and how they can be corrected. In addition, the user manual should serve as a reference that supports the user in quickly and comfortably finding the correct answers to questions.

Robustness

Robustness reduces the impact of operational mistakes, erroneous input data, and hardware errors.

A software system is robust if the consequences of an error in its operation, in the input, or in the hardware, in relation to a given application, are inversely proportional to the probability of the occurrence of this error in the given application.

- Frequent errors (e.g. erroneous commands, typing errors) must be handled with particular care
- Less frequent errors (e.g. power failure) can be handled more laxly, but still must not lead to irreversible consequences.

Maintainability

Maintainability = suitability for debugging (localization and correction of errors) and for modification and extension of functionality.

The maintainability of a software system depends on its:

- Readability
- Extensibility
- Testability

Readability

Readability of a software system depends on its:

- Form of representation
- Programming style
- Consistency
- Readability of the implementation programming languages
- Structuredness of the system
- Quality of the documentation

- Tools available for inspection

Extensibility

Extensibility allows required modifications at the appropriate locations to be made without undesirable side effects.

Extensibility of a software system depends on its:

- Structuredness (modularity) of the software system
- Possibilities that the implementation language provides for this purpose
- Readability (to find the appropriate location) of the code
- Availability of comprehensible program documentation

Testability

Testability: suitability for allowing the programmer to follow program execution (run-time behavior under given conditions) and for debugging.

The testability of a software system depends on its:

- Modularity
- Structuredness

Modular, well-structured programs prove more suitable for systematic, stepwise testing than monolithic, unstructured programs.

Testing tools and the possibility of formulating consistency conditions (assertions) in the source code reduce the testing effort and provide important prerequisites for the extensive, systematic testing of all system components.

Efficiency

Efficiency: ability of a software system to fulfill its purpose with the best possible utilization of all necessary resources (time, storage, transmission channels, and peripherals).

Portability

Portability: the ease with which a software system can be adapted to run on computers other than the one for which it was designed.

The portability of a software system depends on:

- Degree of hardware independence
- Implementation language
- Extent of exploitation of specialized system functions
- Hardware properties
- Structuredness: System-dependent elements are collected in easily interchangeable program components.

A software system can be said to be portable if the effort required for porting it proves significantly less than the effort necessary for a new implementation.

1.2.2 The importance of quality criteria

The quality requirements encompass all levels of software production.

Poor quality in intermediate products always proves detrimental to the quality of the final product.

- Quality attributes that affect the end product
- Quality attributes that affect intermediate products

Quality of *end products* [Bons 1982]:

- Quality attributes that affect their *application*: These influence the suitability of the product for its intended application (correctness, reliability and user friendliness).
- Quality attributes related to their *maintenance*: These affect the suitability of the product for functional modification and extensibility (readability, extensibility and testability).
- Quality attributes that influence their *portability*: These affect the suitability of the product for porting to another environment (portability and testability).

Quality attributes of *intermediate products*:

- Quality attributes that affect the transformation: These affect the suitability of an intermediate product for immediate transformation to a subsequent (high-quality) product (correctness, readability and testability).
- Quality attributes that affect the quality of the end product: These directly influence the quality of the end product (correctness, reliability, adequacy, readability, extensibility, testability, efficiency and portability).

1.2.3 The effects of quality criteria on each other

Attribute	Effect on										
		Correctness	Dependability	Adequacy	Learnability	Robustness	Readability	Modifiability/extensibility	Testability	Efficiency	Portability
Correctness		+	0	0	+	0	0	0	0	0	0
Dependability	0		0	0	+	0	0	0	0	-	0
Adequacy	0	0		+	0	0	0	0	0	+	-
Learnability	0	0	0		0	0	0	0	0	-	0
Robustness	0	+	+	0		0	0	+	-	0	
Readability	+	+	0	0	+		+	+	-	+	
Modifiability/extensibility	+	+	0	0	+	0		+	-	+	
Testability	+	+	0	0	+	0	+		-	+	
Efficiency	-	-	+	-	-	-	-	-		-	
Portability	0	0	-	0	0	0	+	0	-		

Table 1.1 Mutual effects between quality criteria (“+”: positive effect, “-”: negative effect, “0”: no effect)

1.2.4 Quality assurance measures

The most important measures are:

1. Constructive measures:
 - Consistent application of methods in all phases of the development process
 - Use of adequate development tools
 - Software development on the basis of high-quality semifinished products
 - Consistent maintenance of development documentation
2. Analytical measures:

- Static program analysis
- Dynamic program analysis
- Systematic choice of adequate test cases
- Consistent logging of analysis results

3. Organizational measures:

- Continuous education of product developers
- Institutionalization of quality assurance

1.3 The phases of a software project

Software projects are divided into individual phases. These phases collectively and their chronological sequence are termed the *software life cycle*.

Software life cycle: a time span in which a software product is developed and used, extending to its retirement.

The cyclical nature of the model expresses the fact that the phases can be carried out repeatedly in the development of a software product.

1.3.1 The classical sequential software life-cycle model

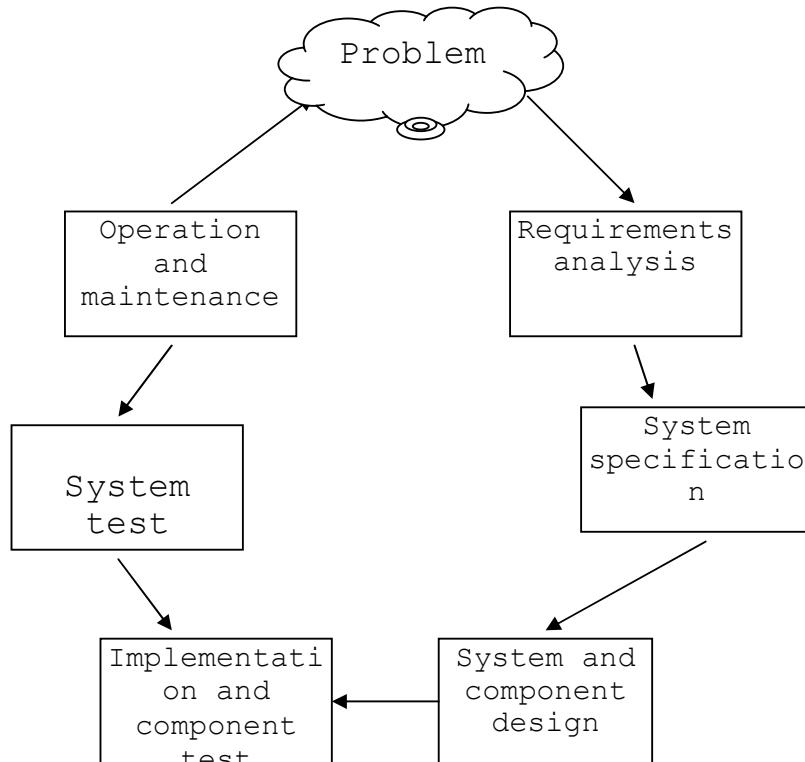


Figure 1.2 The classical sequential software life-cycle model

Requirements analysis and planning phase

Goal:

- Determining and documenting:
 - ❖ Which steps need to be carried out,
 - ❖ The nature of their mutual effects,
 - ❖ Which parts are to be automated, and
 - ❖ Which resources are available for the realization of the project.

Important activities:

- Completing the requirements analysis,
- Delimiting the problem domain,
- Roughly sketching the components of the target system,
- Making an initial estimate of the scope and the economic feasibility of the planned project, and
- Creating a rough project schedule.

Products:

- User requirements,
- Project contract, and
- Rough project schedule.

System specification phase

Goal:

- a contract between the client and the software producer (precisely specifies what the target software system must do and the premises for its realization.)

Important activities:

- Specifying the system,
- Compiling the requirements definition,
- Establishing an exact project schedule,
- Validating the system specification, and
- Justifying the economic feasibility of the project.

Products:

- Requirements definition, and
- Exact project schedule.

System and components design

- Determining which system components will cover which requirements in the system specification, and
- How these system components will work together.

Important activities:

- Designing system architecture,
- Designing the underlying logical data model,
- Designing the algorithmic structure of the system components, and
- Validating the system architecture and the algorithms to realize the individual system components.

Products:

- Description of the logical data model,
- Description of the system architecture,
- Description of the algorithmic structure of the system components, and
- Documentation of the design decisions.

Implementation and component test

- Transforming the products of the design phase into a form that is executable on a computer.

Important activities:

- Refining the algorithms for the individual components,
- Transferring the algorithms into a programming language (coding),
- Translating the logical data model into a physical one,
- Compiling and checking the syntactical correctness of the algorithm, and
- Testing, and syntactically and semantically correcting erroneous system components.

Products:

- Program code of the system components,
- Logs of the component tests, and
- Physical data model.

System test

- Testing the mutual effects of system components under conditions close to reality,
- Detecting as many errors as possible in the software system, and
- Assuring that the system implementation fulfills the system specification.

Operation and maintenance

Task of software maintenance:

- Correcting errors that are detected during actual operation, and
- Carrying out system modifications and extensions.

This is normally the longest phase of the software life cycle.

Two important additional aspects:

- Documentation, and
- Quality assurance.

During the development phases the *documentation* should enable communication among the persons involved in the development; upon completion of the development phases it supports the utilization and maintenance of the software product.

Quality assurance encompasses analytical, design and organizational measures for quality planning and for fulfilling quality criteria such as correctness, reliability, user friendliness, maintainability, efficiency and portability.

1.3.2 The waterfall model

Developed in the 1970s [Royce 1970].

The Waterfall Model represent an experience-based refinement of the classical sequential software life-cycle model.

Two extensions:

1. It introduces iteration between the phases along with the restriction of providing iterations, if possible, only between successive phases in order to reduce the expense of revision that results from iterations over multiple phases.
2. It provides for validation of the phase outputs in the software life cycle.

This approach modifies the strictly sequential approach prescribed by the classical life-cycle model and advances an incremental development strategy. Incorporating a stepwise development strategy for the system specifications and the system architecture as well as phase-wise validation helps to better manage the effects of poor decisions and to make the software development process more controllable.

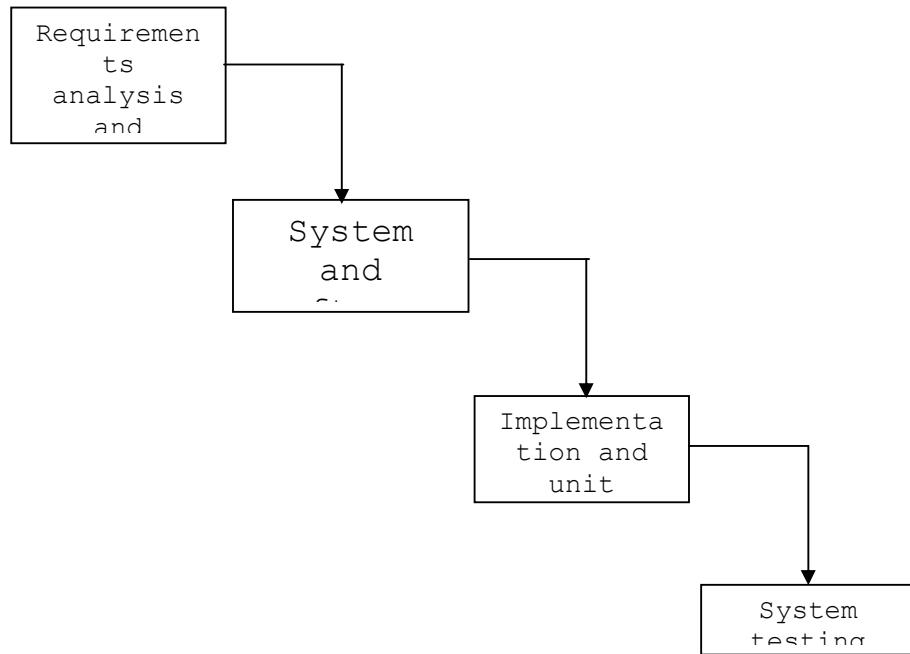


Figure 1.3 The Waterfall model

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.

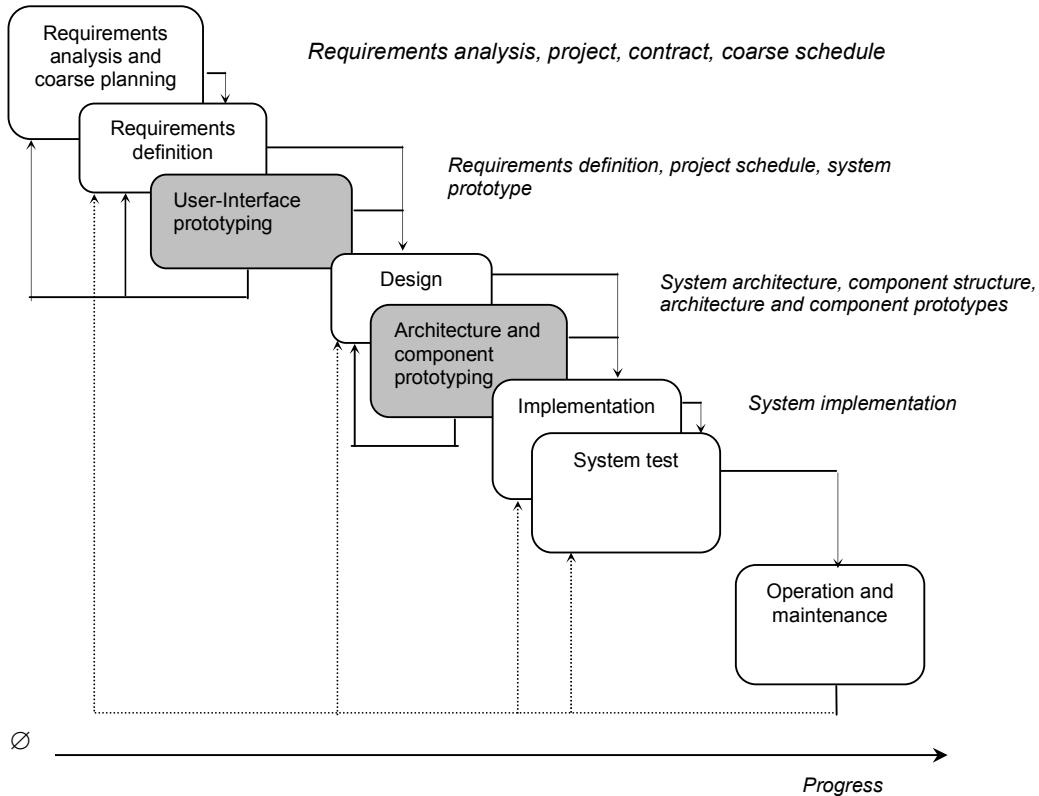


Figure 1.4 Prototyping-oriented software life cycle

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:

- Goals for and requirements on the (sub)product
- Alternatives to realization of the (sub)product
- Constraints and restrictions

The next step evaluates the proposed solution variant with respect to the project goals and applicable constraints, emphasizing the detection of risks and uncertainties. If such are found, measures and strategies are considered to reduce these risks and their effects.

Important aspects of the spiral model:

- Each cycle has its own validation step that includes all persons participating in the project and the affected future users or organizational unit,
- Validation encompasses all products emanating from the cycle, including the planning for the next cycle and the required resources.

1.3.5 The object-oriented life-cycle model (Figure 1.5)

The usual division of a software project into phases remains intact with the use of object-oriented techniques.

The requirements analysis stage strives to achieve an understanding of the client's application domain.

The tasks that a software solution must address emerge in the course of requirements analysis.

The requirements analysis phase remains completely independent of an implementation technique that might be applied later.

In the system specification phase the requirements definition describes *what* the software product must do, but not *how* this goal is to be achieved.

One point of divergence from conventional phase models arises because implementation with object-oriented programming is marked by the assembly of already existing components.

The advantages of object-oriented life-cycle model:

- Design no longer be carried out independently of the later implementation because during the design phase we must consider which components are available for the solution of the problem. Design and implementation become more closely associated, and even the choice of a different programming language can lead to completely different program structures.
- The duration of the implementation phase is reduced. In particular, (sub)products become available much earlier to allow testing of the correctness of the design. Incorrect decisions can be recognized and corrected earlier. This makes for closer feedback coupling of the design and implementation phases.
- The class library containing the reusable components must be continuously maintained. Saving at the implementation end is partially lost as they are

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.

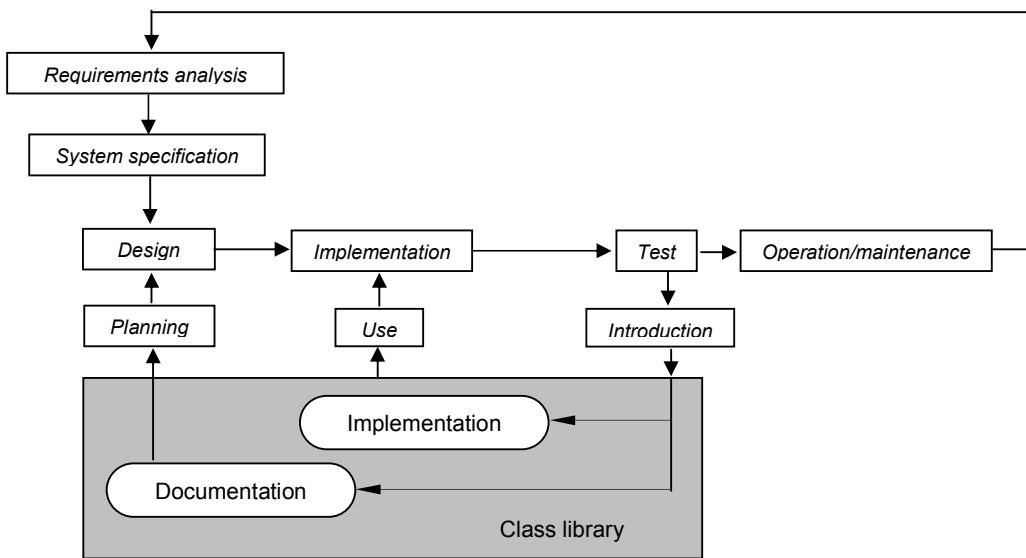


Figure 1.5 Object-oriented life-cycle model

1.3.6 The object and prototyping-oriented life-cycle model (Figure 1.6)

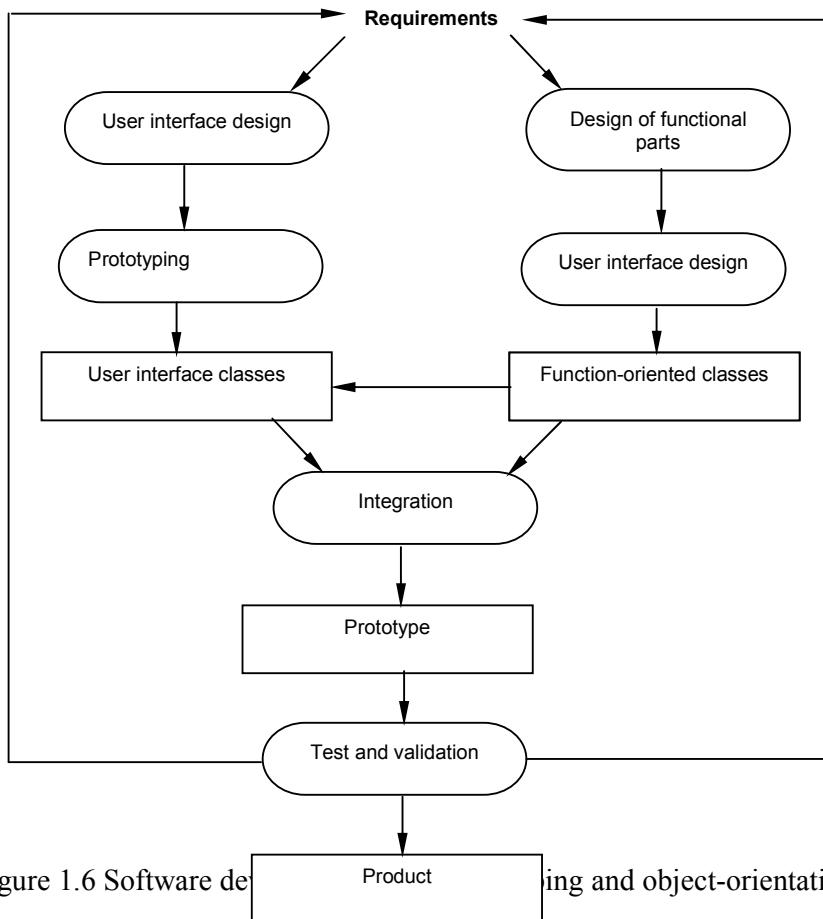
The specification phase steadily creates new prototypes. Each time we are confronted with the problem of having to modify or enhance existing prototypes. If the prototypes were already implemented with object-oriented technology, then modifications and extensions are particularly easy to carry out. This allows an abbreviation of the specification phase, which is particularly important when proposed solutions are repeatedly discussed with the client. With such an approach it is not important whether the prototype serves solely for specification purposes or whether it is to be incrementally developed to the final product. If no prototyping tools are available, object-oriented programming can serve as a substitute tool for modeling user interfaces. This particularly applies if an extensive class library is available for user interface elements.

For incremental prototyping (i.e. if the product prototype is to be used as the basis for the implementation of the product), object-oriented programming also proves to be a suitable medium. Desired functionality can be added stepwise to the prototypes without having to change the prototype itself. This results in a clear distinction between the user interface modeled in the specification phase and the actual functionality of the program. This is particularly important for the following reasons:

- This assures that the user interface is not changed during the implementation of the program functionality. The user interface developed in collaboration with the client remains as it was defined in the specification phase.
- In the implementation of the functionality, each time a subtask is completed, a more functional prototype results, which can be tested (preferably together with the client) and compared with the specifications. During test runs situations sometimes arise that require rethinking the user interface. In such cases the

software life cycle retreats one step and a new user interface prototype is constructed.

Since the user interface and the functional parts of the program are largely decoupled, two cycles result that share a common core. The integration of the functional classes and the user interface classes creates a prototype that can be tested and validated. This places new requirements on the user interface and/or the functionality, so that the cycle begins.



Chapter 2

Software Specification

Objectives

This chapter concerns both formal and informal specification techniques and their use in the software process. The role of specification in the software process and the major problems concerned with producing complete and consistent specifications are discussed in the first part of the chapter. Two techniques for specification, algebraic and model-based specification are described in the rest part of the chapter. Algebraic specification is a specification technique where the actions on an object are specified in terms of their relationships. This is particularly appropriate when used in conjunction with an object-oriented approach to design as it follows the object classes to be formally specified. Model-based specification is a complementary technique where the system is

modeled using mathematical entities, such as sets, whose properties are well understood.

Contents

- 2.1 Structure and contents of the requirements definition
 - 2.2 Quality criteria for requirements definition
 - 2.3 Fundamental problems in defining requirements
 - 2.4 Algebraic specification
 - 2.5 Model-based specification
- References and selected reading
-

2.1 Structure and contents of the requirements definition

The sectional organization of the requirements definition (ANSI/IEEE guide to *Software Requirement Specification* [ANSI 1984]) are:

1. Initial situation and goals
2. System application and system environment
3. User interfaces
4. Functional requirements
5. Nonfunctional requirements
6. Exception handling
7. Documentation requirements
8. Acceptance criteria
9. Glossary and index

1. Initial situation and goals

Contents:

A general description of

- ❖ The initial situation with reference to the requirements analysis,
- ❖ The project goals, and
- ❖ A delimitation of these goals with respect to the system environment.

2. System application and system environment

Contents:

- ❖ 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.

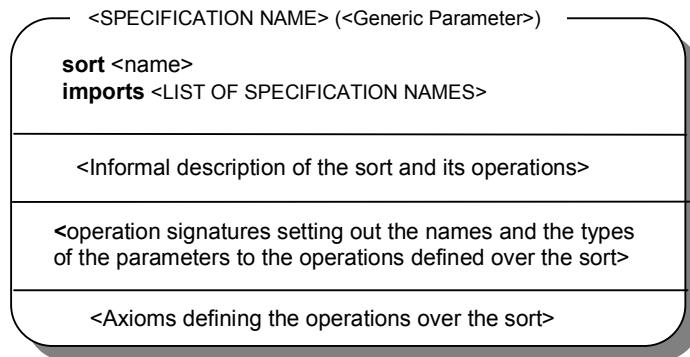


Figure 2.1 The format of an algebraic specification.

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:

- ❖ Operations in terms of their relationships with each other.

Two classes of operations:

- *Constructor operations*: Operation that create or modify entities of the sort which is defined in the specification.
- *Inspection operations*: Operations that evaluate attributes of the sort which is defined in the specification.

Example (Figure 2.2)

Sort: Coord.

Operations:

- Creating a coordinate,
- Testing coordinates for equality, and
- Accessing the X and Y components.

Imports:

Two specifications:

- BOOLEAN
- INTEGER.

Note:

- In the specification of the Eq operation the operator '=' is overloaded.

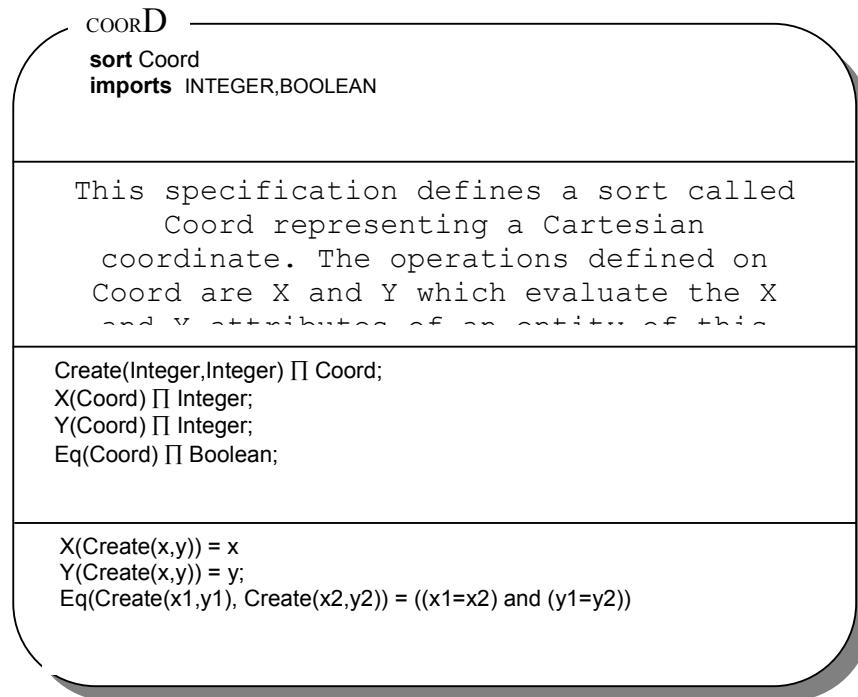


Figure 2.2 The specification of Coord.

2.5 Model-based specification

Specification languages:

- Z ([Abrial 1980], [Hayes 1987])
- VDM ([Jones 1980], [Jones 1986])
- RAISE ([RAISE 1992])

Note:

- *Z is based on typed set theory because sets are mathematical entities whose semantics are formally defined.*

Advantages:

- ❖ Model-based specification is a technique that relies on formulating a model of the system using well understood mathematical entities such as sets and functions
- ❖ System specifications are specified by defining how they affect the overall system model
- ❖ By contrast with the algebraic approach, the state of the system is exposed and a richer variety of mathematical operations is available
- ❖ State changes are straightforward to define

- ❖ All of the specification for each operation is grouped together
- ❖ The model is more concise than corresponding algebraic specifications.

A specification in Z is presented as a collection of schemas where a schema introduces some specification entities and sets out relationships between these entities.

Schema form: (Figure 2.3)

- *Schema name* (the top line)
- *Schema Signature* sets out the names and types of the entities introduced in the schema.
- *Schema predicate* (the bottom part of the specification) sets out the relationships between the entities in the signature by defining a predicate over the signature entities which must always hold.

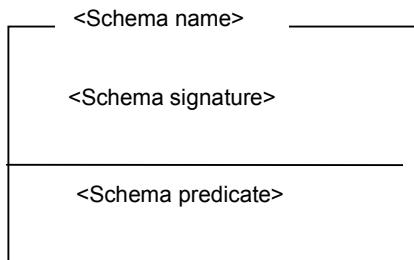


Figure 2.3 Schema form

Example 2.1

A specification of a container which can be filled with “thing” (Figure 2.4).

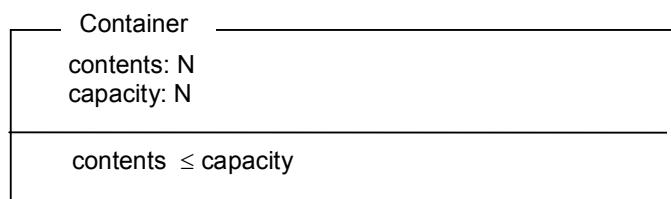


Figure 2.4 The specification of a container

- *Schema name*: Container
- *Schema signature*:
 - contents: a natural number
 - capacity: a natural number
- *Schema predicate*: contents ≤ capacity

(contents cannot exceed the capacity of the container.)

Example 2.2

A specification of an indicator (Figure 2.5).

- *Schema name:* Indicator
- *Schema signature:*
 - light: {off, on}
 - reading: a natural number
 - danger: a natural number
- *Schema predicate:* light = on \Leftrightarrow reading \leq danger

Notes:

- *Light is modeled by the values off and on,*
- *Reading is modeled as a natural number*
- *Danger is modeled as a natural number.*
- *The light should be switched on if only if the reading drops to some dangerous value*

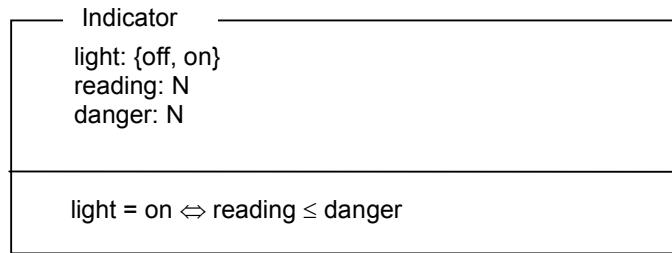


Figure 2.5 The specification of an indicator

Example 2.3

Given the specification of an indicator and a container, they can be combined to define a hopper which is a type of container (Figure 2.6).

- *Schema name:* Hopper
- *Schema signature:*
 - Container
 - Indicator

- Schema predicate:

reading = contents
 capacity = 5000
 danger = 50

Notes:

- *Hopper which has a capacity of 5000 “things”*
- *Light comes on when contents drops to 1% full*
- *We need not specify what is held in the hopper.*

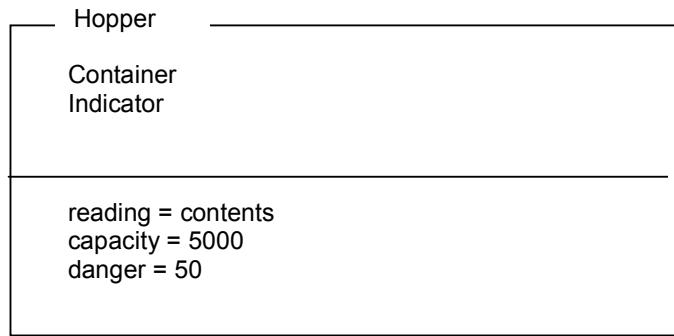


Figure 2.6 The specification of a hopper

The effect of combining specifications is to make a new specification which inherits the signatures and the predicates of the included specifications. Thus, hopper inherits the signatures of Container and Indicator and their predicates. These are combined with any new signatures and predicates which are introduced in the specification. In Figure 2.7, three new predicates are introduced.

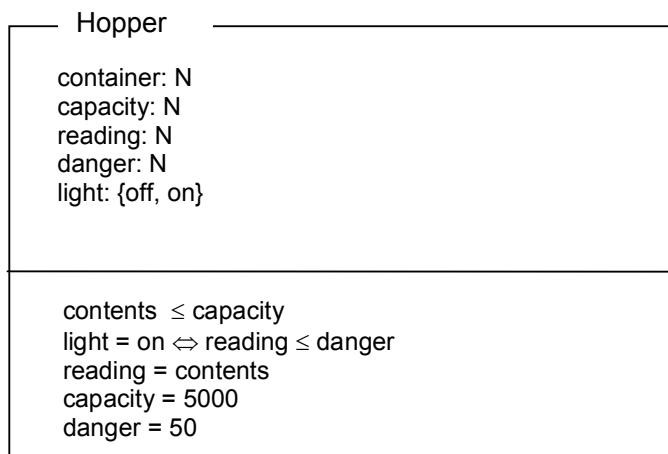


Figure 2.7 The expanded specification of a hopper

Example 2.4

Operation FillHopper (Figure 2.8)

- ❖ The fill operation adds a specified number of entities to the hopper.

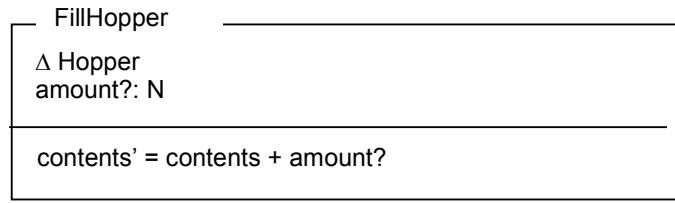


Figure 2.8 The specification of the hopper filling operation

New notions:

- Delta schemas (Figure 2.9), and
- Inputs.

? symbol:

- ? is a part of the name
- Names whose final character is a ? are taken to indicate inputs.

Predicate: $\text{contents}' = \text{contents} + \text{amount}$?

(the contents after completion of the operation (referred as contents') should equal the sum of the contents before the operation and the amount added to the hopper).

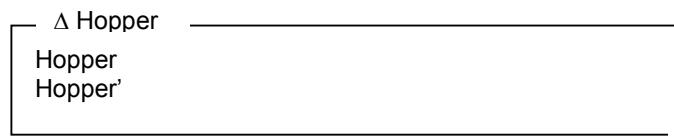


Figure 2.9 A delta schema

New notions:

- Ξ schemas (Figure 2.10)

Some operations do not result in a change of value but still find it useful to reference the values before and after operation.

Figure 2.10 (a Ξ schema) shows a schema which includes a delta schema and a predicate which states explicitly that the values are unchanged.

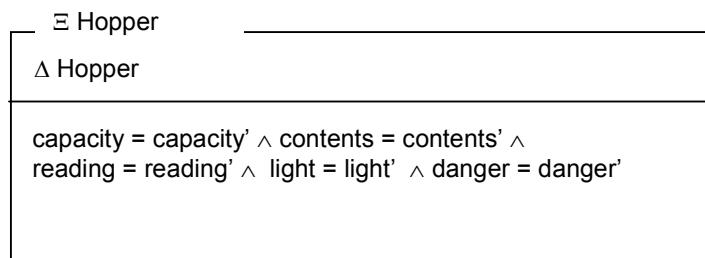


Figure 2.10 A Ξ schema

One of the most commonly used techniques in model-based specifications is to use functions or mappings in writing specifications. In programming languages, a *function* is a abstraction over an expression. When provided with an input, it computes an output value based on the value of the input. A *partial function* is a function where not all possible inputs have a defined output.

The *domain* of a function is the set of inputs over which the function has a defined result.

The *range* of a function is the set of results which the function can produce.

If ordering is important, *sequences* can be used as a specification mechanism. Sequences may be modeled as functions where the domain is the natural numbers greater than zero and the range is the set of entities which may be held in the sequence.

Objectives

This chapter gives a general discussion of software design. The introduction considers the importance of design and its role in the software process. Some methods and techniques are presented for mastering design problems, that is, for decomposing systems into subsystems and subsystems into components, and for handling questions regarding the internal structure of system components. First we give an overview of various design techniques; then we explore some of those design techniques and their methodological foundation in detail and demonstrate their application with an example. In addition, we discuss certain design aspects such as the design of user interfaces and modularization criteria.

Contents

3.1 Design techniques

- 3.1.1 Top-down design
- 3.1.2 Bottom-up design
- 3.1.3 Systems design
- 3.1.4 Design decomposition

3.2 User interface design

3.3 Function-oriented design

3.4 Object-oriented design

- 3.4.1 The Abbott Method
- 3.4.2 Design of class hierarchies
- 3.4.3 Generalization

References and selected reading

3.1 Design techniques

- Specification design sits at the technical kernel of the software engineering process
- Specification design is applied regardless of the software process model that is used.
- Software design is the first of three technical activities – *design, code generation, and testing* – that are required to build and verify the software.

One of the most important principles for mastering the complexity of software systems is the *principle of abstraction*.

Two approaches:

- Top-down design, and
- Bottom-up design.

3.1.1 Top-down design

([Dijkstra 1969], [Wirth 1971])

- The design activity must begin with the analysis of the requirements definition and should not consider implementation details at first.
- A project is decomposed into subprojects, and this procedure is repeated until the subtasks have become so simple that an algorithm can be formulated as a solution.
- Top-down design is a successive concretization of abstractly described ideas for a solution.
- Abstraction proves to be the best medium to render complex systems comprehensible; the designer is involved only with those aspects of the system that are necessary for understanding before the next step or during the search for a solution.

3.1.2 Bottom-up design

The fundamental idea:

- To perceive the hardware and layer above it as an abstract machine.

Technique:

- Bottom-up design begins with the givens of a concrete machine and successively develops one abstract machine after the other by adding needed attributes, until a machine has been achieved that provides the functionality that the user requires.

An abstract machine is a set of basic operations that permits the modeling of a system.

3.1.3 Systems design

- In large-scale embedded system, the design process includes an element of systems design in which functions are partitioned into software and hardware functions.
- The advantage of implementing functionality in hardware is that a hardware component can deliver much better performance than the equivalent software unit. System bottlenecks can be identified and replaced by hardware components, thus freeing the software engineer from expensive software optimization.
- Providing performance in hardware means that the software design can be structured for adaptability and that performance considerations can take second place.

Parallelism

- 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.

- A software system is viewed as a collection of objects that communicate with one another. Each object has data structures that are invisible from outside and operations that can be executed on these structures.
- The decomposition principle orients itself to the unity of data and operations and is based on Parnas' principle of information hiding [Parnas 1972] and the principle of inheritance derived from Dahl and Nygaard [Dahl 1966].

3.2 User interface design

- The design of the user interfaces is a subtask of the design phase.

Modes

- A program mode is a situation in which the user can only execute a limited number of operations.
- Windowing technology can provide valuable services for modes.
- With the help of windows, the user can process various subtasks simultaneously in different windows representing different modes.

Elementary operations

- The key to avoiding modes is the decomposition of the overall user dialog into elementary operations.

Example 3.1

In modern text editors, we can move a block of lines of a text to a new position in the text by four elementary operations:

- *Selection of the block of lines*
- *Menu command “Cut”*
- *Move the cursor to the new position*
- *Menu command “Paste”*

This *cut-copy-paste principle* permits the user a change of mind or corrections after each elementary operation.

The user first determines what is to be manipulated and only then what is to occur whereas in command languages the user first establishes the operation and then specifies the parameters.

Menus

- The requirement to decomposition the user dialog into elementary operations also means that the input of commands should take place via a minimum of individual actions. Menu command provide a mean to achieve this goal.

Classification of menus

- ***Pop-up menus*** prove more efficient because they can appear at any position therefore they require less mouse movement.
- ***Pull-down menus*** permit better structuring of an extensive set of commands and are easier to use with a single-button mouse.

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:

- *The command has no effect:* This variant is easiest to implement, but can cause confusion and frustration for the user when the command fails to function in certain situations.
- *An error message is displayed.* This possibility is preferred over the first because the user can be advised why the execution of the command is impossible. If the same situation arises repeatedly, however, the recurring error message tends to irritate the user.
- *The command* is disabled. The best choice is to prevent the selection of a senseless command from the start. To achieve this goal , the command should not be removed from the menu, which would confuse an occasional user and cause a search in vain for the command. It is better to mark non-executable commands as disabled in some suitable form.

Dialog windows

- *Static text* is used for titles, label and explanations. Dialog windows should be structured to minimize static text. In particular, avoid long explanations by means of adequate labeling of the elements available to the user.
- As a far as possible, use *editable text* only for inputting character strings. For the input of numbers, other user interface elements do a better job of eliminating errors.
- *Command buttons* trigger immediate action. Typically an input dialog window contains an OK button to confirm an input and a Cancel button to abort an action.
- *Checkboxes* permit the selection of Boolean values.
- *Radio buttons* allow the selection of one alternative from a set of mutually exclusive possibilities.
- *Pop-up menus* provide an alternative to radio buttons. A mouse click on a framed area presents a list of available possibilities from which the user can select. Compared to radio buttons, pop-up menus require less space and can be arbitrarily extended. That they normally do not display all available options proves to be a disadvantage.
- *Sliders* enable the input of continuous, variable values. Use sliders wherever precision plays no role, e.g., for adjusting brightness or volume.
- *Selection lists* serve a function similar to pop-up menus. Use them when the number of alternatives exceeds the capacity of a menu.
- *Direction buttons* allow the setting of values in discrete steps. Their function resembles setting a digital watch. Direction buttons make it easy to restrict the numeric range.

- *Icons* (or *pictograms*) frequently provide a space-saving alternative to verbal description.
Note: *The design of icons represents an art; use only self-explanatory pictures, preferably ones that the user recognizes from other areas of daily life.*
- *Frames* and *separators* provide an optical structuring medium. They augment special ordering for structuring a dialog window in cohesive functional blocks. Whenever possible, avoid extensive dialog windows with numerous individual elements. They become time-consuming to use because the user must control every adjustment, even to set only a single field.
- Modern graphical user interfaces with windows and menus have come to dominate not only because of their user friendliness, but also for aesthetic reasons. No one disputes that working with an attractive screen is more fun and so is perceived as less of a burden than character-oriented communication with the computer. Designers of newer systems thus pay particular attention to achieving an appropriate look. The graphic capabilities of monitors are exploited, e.g., to give windows and buttons a plasticity like Three-dimensional objects. Use such design possibilities only if they are standard in the target environment. It would be wrong to sacrifice the uniformity of a user interface for purely aesthetic reasons.

Color

Guidelines for the design of color user interfaces ([Apple 1987])

- About one person in twenty is colorblind in some way. To avoid excluding such persons from using a program, color must not serve as the sole source of information. For example, highlighting an error in a text with red effectively flags it for most people, but is scarcely recognizable, if at all, to a red-green colorblind user. Another form of highlighting (e.g., inversion of the text) should be preferred over color.
- Longitudinal studies have shown that black writing on a white background proves most readable for daily work. For a black background, amber proves the most comfortable to the eyes. Writing in red and blue proves least readable because these colors are at opposite ends of the visible spectrum for humans. Because of the extreme wave length of these colors, their focal point is not exactly on the retina of the eye, but in front of or behind it. Although the eye automatically adapts to reading red or blue text, the resulting strain causes early fatigue. Experts recommend avoiding red and blue to display text.
- Red serves as a danger signal. However, because the user's eye is drawn to red, it would be wrong to employ red to mark dangerous actions. Color "support" in such a case would have the reverse effect.
- Light blue on a white background is hard to read, especially for small objects or thin lines. Thus important things should never be displayed in light blue. Still, this effect can be exploited, for example, to set grid lines in the background as unimportant.
- In general, color should only be used where the task expressly requires it, such as in graphic editors. In all other cases color should be used at best as a

supplementary means for highlighting , but never for mere decoration of a user interface.

Sound

Guidelines for the design of sound user interfaces

- Sound is an excellent way to attract attention. Hence it should only be used when the user's attention is really required, but never as simple background for other actions. Appropriate applications of sound include error situations (e.g. paper exhausted while printing) and the occurrence of temporally uncertain events such as the end of an extended procedure (e.g. data transfer).
- Sound should only be used to underscore a message displayed on the screen, but never as the sole indicator of an exception situation. Otherwise an important message can easily be missed because the user is hearing-impaired, works in a loud environment, or has left the office.
- It might make sense to signal different events with different sound. However, observe that only a user with musical training or talent can discern signals with similar pitch.
- Avoid loud and shrill sounds, imagine a large office where every few minutes another computer trumpets an alarm. For the same reason, avoid sounds with more than three sequential tones; such signals become irritating when heard too often.
- Even in particularly pressing emergencies, never under any circumstances, employ a long, continuous sound or, even worse, an enduring staccato. Such sounds can create panic and provoke erroneous action from the user. It is better to use a short, uninterrupted sound and to repeat it at intervals of one minute until the user reacts.

Consistency

- Menu commands with similar functions should have the same nomenclature and the same location even in different programs,
- Shortcut keys for menu commands should always be consistent. Rather than introduce a nonstandard key combination, leave the menu command without a shortcut,
- Buttons with similar functions should have similar labels and the same relative position in different dialog windows,
- Dialog windows for the input of similar functions should have a consistent look.

A typical user works with various programs on the same computer. The acceptance of a new program depends significantly on whether the user encounters similar interface functionality as in other familiar programs. Every deviation as well as every omission can prove an irritation factor. For example, if the user is accustomed to interrupting any action by pressing a given key combination, then irritation results when this key has no function or a different function in a new program.

The greater the internal consistency of a user interface and its consistency with the user interfaces of other programs, the less effort is required for learning to use the new program. In many cases this can replace extensive user manuals. On the other hand, the obligation to maintain consistency, like any form of standardization, can also restrict possibilities for user interfaces.

Prototyping

- The dynamics of a program cannot be described statically. To avoid unpleasant surprises in the implementation phase, it is important to also define the behavior of the program as early as possible.
- A prototype serves as the best medium for specifying a user interface because it allows playing through the typical steps in working with the planned program. In such an experimentation phase, most questions regarding operation can be discussed and solved together with the client.
- Many software products are available today that support the interactive description of the user interface. Most of these tools generate the source code of a program that implements the user interface definition. The time required to implement a prototype thus shrinks from weeks to hours.

3.3 Function-oriented design

- The best-known design methods are function oriented, which means that they focus on the algorithm to solve the problem.

Example 3.5

Imagine the algorithm as a mathematical function that computes results on the basis of given parameters. At the start of the design stage the algorithm is still a black box whose contents are unknown. The most difficult task to be solved is its solution algorithm. Thus it makes sense to proceed as with the modularization, i.e., to decompose the task into self-contained subtasks, whereby the algorithms for the solution of subtasks again should be viewed as black boxes. The resulting overall solution thus becomes a network of cooperating subalgorithms.

Two simple rules of the stepwise refinement principle of problem solving ([Wirth 1971]):

- Decompose the task into subtasks,
- Consider each subtask independently and decompose it again into subtasks.

Continue the ongoing decomposition of large tasks into smaller subtasks until the subtasks become so simple that they can readily be expressed with statements in the selected programming language.

Solving a task using above approach is characterized as follows:

- For each (sub)task, begin with the identification of its significant components. However, merely specifying the subtasks will not suffice; it must be clear whether and how the solutions to these subtasks can be assimilated into an overall solution. This means that the initial design of an algorithm certainly can

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 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], [Wasserman 1990], [Wilson 1990], [Wirfs-Brock 1989], [Wirfs-Brock 1990].

Function-oriented design and object-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.

- The fundamental problem in object-oriented design consists of finding suitable classes for the solution of the task.
- The next step is identifying the relationships of these classes and their objects to each other and defining the operations that should be possible with the individual objects.

3.4.1 The Abbott Method ([Abbot 1983])

- In object-oriented design we try quickly to identify the objects involved in task. These could be data (e.g. texts), forms (e.g. order forms), devices (e.g. robots) or organizational aids (e.g. archives).
- Verbal specifications can serve as the basis of the design.

Approach:

- Looking for various kinds of words (nouns, verbs) in the specifications and deriving the design from them.
- Forming abstract data structures.
- Defining the operations required on abstract data structures.

The approach can also be used for object-oriented design.

Working schema:

- *Filtering out nouns*

The relevance of a noun depends on its usage:

- *Generic terms* (e.g. person, automobile, traffic light) play a central role. Every such term is a candidate for a class.
- *Proper nouns* serve as identifiers for specific objects (e.g. Paris, War and Peace, Salvador Dali). They can indirectly contribute to forming classes since every proper noun has a corresponding generic term (e.g. city, book, painter).
- *Quantity and size units* (e.g., gallon) and material descriptor (e.g. wood) as well *collective terms* (e.g., government) usually prove unsuitable for forming classes.
- *Abstract terms* (e.g. work, envy, beauty) likewise prove unsuitable.
- *Gerunds* indicate actions. They are treated like the verbs from which they derive.

- *Seeking commonalities*

In real tasks descriptions many related nouns usually occur together, such as “department manager” and “supervisor”, “author” and “write”, “auto” and “vehicle”, “street” and “one-way street”. In such cases the analyst must first determine whether the terms are really distinct, or whether they indicate the same class. The answer to the question “Same or different?” generally emanates from the task description. For example, for a traffic simulation it would be of no importance

whether only cars or arbitrary vehicles utilize a street. For a program to manage license plates and registrations, this difference would certainly play a role. In any case, the criterion for distinguishability is the question of whether the elements of the class in question have differing attributes.

If we find that two related terms designate different things, then we need to clarify what relation they have to one another. Often we find a subset relation that finds expression in sentences like: "Every auto is a vehicle".

- *Finding the relevant verbs*

Verbs denote action with objects. In contrast to the function-oriented approach, here these actions are not viewed in isolation, but always in connection with objects of certain classes. For example, the verb "toggle" belongs to a traffic light, and "accelerate" is associated with a vehicle. Assignment to a certain class proves easy for intransitive verbs because they are bound only to the subject of the sentence (e.g., wait). Transitive verbs and sentence structures that connect multiple nouns present more problems. For example, the statement: "The auto is to be displayed on the screen" does not indicate whether the action "display" should be assigned to the design. A usual solution to the problem is to assign the action to both classes in question, and then during implementation to base one of the two operations on the other.

3.4.2 Design of the class hierarchies

A particular characteristic of object-oriented programming is that the classes (data types) are hierarchically structured. This approach has the organizational advantage that similar things are recognized and combined under a common superordinate term. This produces a tree structure in which the general (abstract) terms are near the root and the leaves represent specialized (concrete) terms.

The class MotorVehicle determines the general attributes of the classes derived from it. For example, independent of the type of vehicle, every vehicle has a brand name, a model, a series number, a power rating for the motor, and many other attributes. Every auto (since it is also a vehicle) also has these attributes. In addition, it has further attributes such as the number of passengers, the number of doors, etc. This scheme can be continued to any extent. Each specialization stage adds further attributes to the general ones.

Another characteristic of a class hierarchy is the extension of functionality with increasing specialization. A convertible, for example, permits all actions that are possible with a car. In addition, its roof can be folded down.

The most important relation in a class library is the *is-a* relation. Since the class Car is a subset of the class Auto, every car (object of class Car) is an auto. This relation represents the first criterion for designing class hierarchies. Often, however, there are multiple possibilities for assignment. For example, the class StationWagon could also be a subset of commercial vehicle for the transport of goods as reflected in the class Truck. Such ambiguities can impede the search for an adequate common superordinate term.

Such a class hierarchy has two consequences for implementation:

- *Inheritance*: Attributes and operations defined for a general class automatically apply to the more specialized classes derived from it. For example, if Auto has a function that computes the top speed from the wind resistance value and the engine data, then this function is also valid for all objects of derived classes (e.g., Car, StationWagon).
- *Polymorphism*: A program fragment that functions for objects of a class K can also work with objects of a class derived from K even though such objects have different attributes (structure, behavior) from K objects. This is possible because derived classes only *add* attributes, but can never *remove* any. This results in a flexibility that can be achieved only with arduous effort with conventional programming.

Often specifications employ terms that are similar in many ways yet differ significantly in certain details. Such an example arises in a program to display rectangles and circles. These geometric shapes have attributes that could be handled collectively (e.g. line thickness, fill pattern, movement on the screen), yet they cannot be combined fruitfully in an is-a relation. In such cases a common superordinate class, or superclass, must be created, e.g., GeomFigure. The attributes common to a circle and a rectangle (e.g., line thickness, color, fill pattern) are then transferred to the common superclass, and only the specialized attributes and operations (center and radius, width and height, drawing operations) are implemented in the derived classes. The common super-class represents an abstraction of the specialized classes. In this context we speak of *abstract classes*. The availability of a collection of prefabricated classes (a class library) can save a great deal of implementation and even design work). A little luck might even produce an exact match, or at least a more general class from which the desired class can be derived by inheritance and specialization. In this case two activities in opposing directions characterize the design procedure: top-down analysis and bottom-up synthesis.

The Abbott Method is a *top-down analytical method* because we begin with the design of the application-oriented classes starting with the task description and progress to the implementation-specific classes. This method gives no consideration to any classes that might already be available. If new classes are derived exclusively on the basis of classes in the class library, this amounts to a *bottom-up constructive approach*. The truth lies somewhere between. For specification of higher complexity, a designer can scarcely find problem-oriented classes in the class library. Thus it makes sense in any case to establish the initial classes analytically.

3.4.3 Generalization

➤ Reusability plays a central role in object-oriented programming.

Type of reusability:

- Reuse within a program
- Reuse between programs.

Within a program reuse is realized with inheritance. High reuse results almost as a side effect of the design of the class hierarchy by the introduction of abstraction levels. A class library as the basis of implementation is the most important medium for achieving high reuse between programs. Usually such a class library is provided by the compiler manufacturer. If similar tasks are to be solved frequently, similar subtasks crystallize

repeatedly that can be solved in the same way in different programs. This suggests striving for as much generality as possible in every class in order to assure high reusability in later projects. Above all the following measures prove useful toward this goal.

- *Far-sighted extensions*: In the development of a new class, more should be implemented than just the attributes absolutely necessary for the solution of the immediate task. Developers should also plan ahead and give consideration to the contexts in which the class might also be used.
- *Factoring*: When it is predictable that a class will later have alternatives with similar behavior, abstract classes should be planned from which the future classes can be derived. Especially when there is a large abstraction gap between existing (base) class and the new class (e.g., between graphic elements and editable text), intermediate classes should be introduced (e.g., non-editable text).
- *Simple functions*: Since existing functions that are not precisely suitable must be reimplemented (overwritten) in the derivation of new classes, complex operations should be avoided. It is better to decompose complex operations to smaller ones during their design: the simpler operations can then be overwritten individually as needed.

The highest possible generality can be achieved most easily if the design of reusable classes is handled by independent design teams that are not directly involved in the current project. Still, only actual reuse can determine whether this goal has been achieved.

Objectives

This chapter gives a general discussion of software implementation. First we give an overview of the attributes of programming languages which impact on quality; then we discuss certain elements of good programming style such as structuredness, expressiveness, outward form and efficiency. Finally we discuss the main properties of software such as portability and reusability.

Contents

4.1 Programming environments

4.2 Programming style

4.3 Portability and reuse

 4.3.1 Software portability

 4.3.2 Machine architecture dependencies

 4.3.3 Operating system dependencies

 4.3.4 Software reuse

4.4 Computer-aided software engineering

 4.4.1 CASE workbenches

 4.4.2 Text editing systems

 4.4.3 Language processing systems

4.5 Incremental implementation

References and selected reading

- Implementation of a software system = the transformation (coding) of the design results into programs that are executable on certain target machine.
- A good implementation reflects the design decisions.
- The implementation should ensure the following:
 - The decomposition structures, data structures and identifiers selected and established during the design procedure are easily recognized in the implementation.
 - The abstraction levels of the design (the classes, modules, algorithms, data structures and data types) must also be feasible in the implementation.

- The interfaces between the components of a software system should be explicitly described in the implementation.
 - It must be possible to check the consistency of objects and operations immediately with the compiler (before the actual testing phase).
- The degree to which the above characteristics are fulfilled depends on the choice of the implementation language and on the programming style.

4.1 Programming environments

- ❖ The question of which is the “right” programming language has always been a favorite topic of discussion in programming circles.
- ❖ The choice of a programming language for the implementation of a project frequently plays an important role.
- ❖ In the ideal case, the design should be carried out without any knowledge of the later implementation language so that it can be implemented in any language.

Quality criteria for programming languages:

- Modularity
- Documentation value
- Data structures
- Control flow
- Efficiency
- Integrity
- Portability
- Dialog support
- Specialized language elements

➤ *Modularity of a programming language*

- ❖ Degree to which it supports modularization of programs.
- ❖ The decomposition of a large program into multiple modules is a prerequisite for carrying out large software projects.
- ❖ Without modularization, division of labor in the implementation becomes impossible. Monolithic programs become unmanageable: they are difficult to maintain and document and they impede implementation with long compile times.
- ❖ Languages such as standard Pascal (which does not support modules, but compare with Turbo Pascal and Modula-2) prove unsuitable for large projects.
- ❖ If a language supports the decomposition of a program into smaller units, there must also be assurance that the components work together. If a procedure is invoked from another module, there must be a check of whether the procedure

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.

➤ Structuring control flow in the programming language

- ❖ Languages like BASIC and FORTRAN include variations of a GOTO statement, which programmers can employ to create unlimited and incomprehensible control flow structures. In Pascal and C the use of the GOTO statement is encumbered because the target must be declared explicitly. In Eiffel, Modula-2 and Smalltalk there is no GOTO statement at all, which forces better structuring of the control flow.
- ❖ In technical applications additional possibilities for control flow can play an important role. These include the handling of exceptions and interrupts as well as parallel processes and their synchronization mechanisms. Many programming languages (e.g., Ada and Eiffel) support several of these concepts and thus permit simpler program structures in certain cases.

Notes: *Every deviation from sequential flow is difficult to understand and thus has a negative effect on readability.*

➤ Efficiency of a programming language

- ❖ The *efficiency* of a programming language is often overrated as a criterion. For example, the programming language C bears the reputation that it supports the writing of very efficient programs, while object-oriented languages are accused of inefficiency. However, there are few cases where a language is in principle particularly efficient or especially inefficient.
- ❖ Optimizing compilers often generate excellent code that an experienced Assembler programmer could hardly improve upon. For time-critical operations, it pays to use a faster machine or a better compiler rather than a “more efficient” programming language.

➤ Integrity of a programming language

- ❖ The integrity of a programming language emanates primarily from its readability and its mechanisms for type checking (even across module boundaries).
- ❖ Independent of the application domain, therefore, a language with static typing should be preferred. Static typing means that for each expression the compiler can determine which type it will have at run time. Additional integrity risks include type conversions (type casts) and pointer arithmetic operations, which are routine for programming in C, for example.
- ❖ Run-time checks are also important for integrity, especially during the development phase. These normally belong in the domain of the compiler.
- ❖ Mechanisms for formulating assertions (e.g. in Eiffel) and features for exception handling (e.g. in Eiffel and Ada) also contribute to the integrity of a programming language.

➤ Portability

- ❖ *Portability* can be a significant criterion if a software product is destined for various hardware platforms. In such a situation it makes sense to select a standardized language such as Ada or C. However, this alone does not suffice to ensure portability. For any external modules belonging to the language also need

to be standardized. This is a problem in the language Modula-2 because various compiler producers offer different module libraries.

- ❖ Beyond standardization, another criterion is the availability of compilers for the language on different computers. For example, developers of software for mainframes will find a FORTRAN compiler on practically every machine.

➤ ***Dialog support***

- ❖ For interactive programs the programming language must also provide *dialog support*. For example, FORTRAN and COBOL offer only line-oriented input and output;
- ❖ Highly interactive programs (that react to every key pressed) can thus be developed only with the help of specialized libraries.
- ❖ Some languages like BASIC and Logo are particularly designed to provide dialog support for the user (and with the programmer), making these languages better suited for such applications.
- ❖ Object-oriented programming languages also prove well suited to the development of interactive programs, especially with the availability of a corresponding class library or an application framework.
- ❖ For specialized tasks, *specialized language elements* can be decisive for the selection of a programming language. For technical applications, for example, the availability of complex number arithmetic (e.g. in COBOL) can be important. For mathematical problems, matrix operations (e.g. in APL) can simplify the task, and translation and character string operations are elegantly solved in Snobol. The lack of such specialized language elements can be compensated for with library modules in modular languages.
- ❖ Object-oriented languages prove particularly suited to extending the language scope.

Additional characteristics of programming languages:

- Quality of the compiler
- Availability of libraries
- Availability of development tools
- Company policy
- External requirements

➤ ***Quality of the compiler***

The *quality of the compiler* is decisive for the actual implementation phase. A good compiler should not only generate efficient code, but also provide support for debugging (e.g. with clear error messages and run-time checks). Particularly in the area of microcomputers, many compilers have been integrated in development systems. Here the user-friendliness of such systems must also be considered.

➤ ***Availability of libraries***

With modular programming languages, the *availability of libraries* for various application domains represents a significant selection criterion. For example, for practically all FORTRAN compilers, libraries are available with numerous mathematical functions, and Smalltalk class libraries contain a multitude of general classes for constructing interactive programs. The availability of libraries can also be used in C or Modular-2 if the compiler supports linking routines from different languages. On the other hand, there are libraries that are available only in compiled form and usable only in connection with a certain compiler.

➤ ***Availability of development tools***

Today's trend is toward the use of *tools* to support software development. Many tools intended for the implementation phase are bound to a particular programming language or even a certain compiler. Examples of such tools include structure-oriented editors, debugger and program generators. For example, if tools like Lex and YACC (tools for the application of attribute grammars on UNIX machines) are to be employed, then an implementation in C becomes unavoidable; choosing another language would only be justified if it promises gains greater than the cost of forfeiting the tool.

➤ ***Company policy***

Often a particular *company policy* influences the choice of a programming language. Frequently the language decision is made not by the implementors, but by managers that want to use only a single language company-wide for reasons of uniformity. Such a decision was made by the U.S. Department of Defense, which mandates the use of Ada for all programming in the military sector in the U.S (and thus also in most other NATO countries). Such global decisions have also been made in the area of telecommunications, where many programmers at distributed locations work over decades on the same product.

Even in-house situations, such as the education of the employees or a module library built up over years, can force the choice of a certain language. A company might resist switching to a more modern programming language to avoid training costs, the purchase of new tools, and the re-implementation of existing software.

➤ ***External requirements***

Sometimes *external requirements* force the use of a given programming language. Contracts for the European Union increasingly prescribe Ada, and the field of automation tends to require programs in FORTRAN or C. Such requirements arise when the client's interests extend beyond the finished product in compiled form and the client plans to do maintenance work and further development in an in-house software department. Then the education level of the client's programming team determines the implementation language.

4.2 Programming style

- After they have been implemented and tested, software systems can very seldom be used over a longer time without modifications. In fact, usually the opposite is true: as the requirements are updated or extended after completion of the product and during its operation, undetected errors or shortcomings arise. The implementation must constantly be modified or extended, necessitating repeated reading and

understanding of the source code. In the ideal case the function of a program component should be comprehensible without knowledge of the design documents, only from its source code. The source code is the only document that always reflects the current state of the implementation.

- The readability of a program depends on the programming language used and on the programming style of the implementor. Writing readable programs is a creative process. The programming style of the implementor influences the readability of a program much more than the programming language used. A stylistically well-written FORTRAN or COBOL program can be more readable than a poorly written Modula-2 or Smalltalk program.

Most important elements of good programming style:

- Structuredness
- Expressiveness
- Outward form
- Efficiency

This refers to both the *design* and the *implementation*.

Although efficiency is an important quality attribute, we do not deal with questions of efficiency here. Only when we understood a problem and its solution correctly does it make sense to examine efficiency.

4.2.1 Structuredness

- Decomposing a software system with the goal of mastering its complexity through abstraction and striving for comprehensibility (*Structuring in the large*.)
- Selecting appropriate program components in the algorithmic formulation of subsolutions (*Structuring in the small*.)

Structuring in the large

- ❖ Classes and methods for object-oriented decomposition
- ❖ Modules and procedures assigned to the modules.
- ❖ During implementation the components defined in the design must be realized in such a way that they are executable on a computer. The medium for this translation process is the programming language.
- ❖ For the implementation of a software system, it is important that the decomposition defined in the design be expressible in the programming language; i.e. that all components can be represented in the chosen language.

Structuring in the small

- ❖ The understanding and testing of algorithms require that the algorithms be easy to read.
- ❖ Many complexity problems ensue from the freedom taken in the use of GOTO statements, i.e., from the design of unlimited control flow structures.
- ❖ The fundamental ideal behind structured programming is to use only control flow structures with one input and one output in the formulation of algorithms.

This leads to a correspondence between the static written form of an algorithm and its dynamic behavior. What the source code lists sequentially tends to be executed chronologically. This makes algorithms comprehensible and easier to verify, modify or extend.

- ❖ Every algorithm can be represented by a combination of the control elements sequence, branch and loop (all of which have one input and one output) ([Böhm 1996]).
- ❖ D-diagrams (named after Dijkstra): Diagrams for algorithm consisting of only elements sequence, branch and loop.

Note: *If we describe algorithms by a combination of these elements, no GOTO statement is necessary. However, programming without GOTO statements alone cannot guarantee structuredness. Choosing inappropriate program components produces poorly structured programs even if they contain no GOTO statements.*

4.2.2 Expressive power

- The implementation of software systems encompasses the naming of objects and the description of actions that manipulate these objects.
- The choice of names becomes particularly important in writing an algorithm.

Recommendation:

- Choose expressive names, even at the price of long identifiers. The writing effort pays off each time the program must be read, particular when it is to be corrected and extended long after its implementation. For local identifiers (where their declaration and use adjoin) shorted names suffice.
 - If you use abbreviations, then use only ones that the reader of the program can understand without any explanation. Use abbreviations only in such a way as to be consistent with the context.
 - Within a software system assign names in only one language (e.g. do not mix English and Vietnamese).
 - Use upper and lower case to distinguish different kinds of identifiers (e.g., upper case first letter for data types, classes and modules; lower case for variables) and to make long names more readable
(e.g. CheckInputValue).
 - Use nouns for values, verbs for activities, and adjectives for conditions to make the meaning of identifiers clear (e.g., width, ReadKey and valid, respectively).
 - Establish your own rules and follow them consistently.
- Good programming style also finds expression in the use of comments: they contribute to the readability of a program and are thus important program components. Correct commenting of programs is not easy and requires experience, creativity and the ability to express the message concisely and precisely.

The rules for writing comments:

- Every system component (every module and every class) should begin with a detailed comment that gives the reader information about several questions regarding the system component:
 - What does the component do?
 - How (in what context) is the component used?
 - Which specialized methods, etc. are use?
 - Who is the author?
 - When was the component written?
 - Which modifications has have been make?

Example:

```
/* FUZZY SET CLASS: FSET
FSET.HPP    2.0,   5 Sept. 1996
Lists operations on fuzzy sets
Written by Nguyen Xuan Huy
*/
```

- Every procedure and method should be provided with a comment that describes its task (and possibly how it works). This applies particularly for the interface specifications.
- Explain the meaning of variables with a comment.
- Program components that are responsible for distinct subtasks should be labeled with comments.
- Statements that are difficult to understand (e.g. in tricky procedures or program components that exploit features of a specific computer) should be described in comments so that the reader can easily understand them.
- A software system should contain comments that are as few and as concise as possible, but as many adequately detailed comments as necessary.
- Ensure that program modifications not only affect declarations and statements but also are reflected in updated comments. Incorrect comments are worse than none at all.

Note: *These rules have deliberately been kept general because there are no rules that apply uniformly to all software systems and every application domain. Commenting software systems is an art, just like the design and implementation of software systems.*

4.2.3 Outward form

- *Beyond name selection and commenting, the readability of a software systems also depends on its outward form.*

Recommended rules for the outward form of programs:

- For every program component, the declarations (of data types, constants, variables, etc.) should be distinctly separated from 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], [Nissen 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.

These problems can be minimized, however, by making use of abstract data types and by ensuring that real-world entities are always modelled using such types rather than inbuilt program types.

- The general approach which should be adopted in building a portable system is to isolate those parts of the system which depend on the machine architecture and the operating system in a portability interface. All operations which make use of non-portable characteristics should go through this interface.
- The portability interface should be a set of abstract data types or objects which encapsulate the non-portable features and which hide any representation characteristics from the client software. When the system is moved to some other hardware or operating system, only the portability interface need be rewritten to reimplement the software.
- The rate of change of computer hardware technology is so fast that computers become obsolete long before the programs that execute on these machines. It is therefore important that programs should be written so that they may be implemented under more than one computer operating system configuration. This is doubly important if a programming system is widely marketed as a product. The more machines on which a system is implemented, the greater the potential market for it.

4.3.2 Machine architecture dependencies

The principal machine architecture dependencies arise in programs because the programming language must rely on the conventions of information representation adopted by the host machine. Different machines have different word lengths, different character sets and different techniques for representing integer and real numbers.

The length of a computer word directly affects the range of integers available on that machine, the precision of real numbers and the number of characters which may be packed into a single word. It is extremely difficult to program in such a way that implicit dependencies on machine word lengths are avoided.

For example, say a program is intended to count instances of some occurrence where the maximum number of instances that might arise is 500 000. Assume instance counts are to be compared. If this program is implemented on a machine with a 32-bit word size, instance counts can be represented as integers and comparisons made directly. However, if the program is subsequently moved to a 16-bit machine, it will fail because the maximum possible positive integer which can be held in a 16-bit word is 32 767.

Such a situation poses a difficult problem for the programmer. If instance counts are not represented as integers but as real numbers or as character strings this introduces an unnecessary overhead in counting and comparisons on the 32-bit machine. The cost of this overhead must be traded off against the cost of the reprogramming required if the system is subsequently implemented on a 16-bit machine.

If portability considerations are paramount in such a situation and if a programming language which permits the use of user defined types is available, an alternative solution to this problem is possible. An abstract data type, say *CountType*, whose range

encompasses the possible values of the instance counter should be defined. In Pascal on a 32-bit machine this might be represented as follows:

```
type CountType = 0..500000;
```

If the system is subsequently ported to a 16-bit machine, then *CountType* may be redefined:

```
type CountType = array [1..6] of char;
```

Instead of using an integer to hold the counter value, it may be held as an array of digits. Associated with this type must be the operations permitted on instance counters. These can be programmed in Pascal as functions.

The procedure letter must be rewritten when the program is transferred to a machine with an incompatible character set.

4.3.3 Operating system dependencies

- As well as machine architecture dependencies, the other major portability problem which arises in high-level language is dependencies on operating system facilities. Different machines support different operating systems. Although common facilities are provided, this is rarely in a compatible way.
- Some *de facto* operating system standards have emerged (MS-DOS for personal computers, UNIX for workstations, for example), but these are not supported by some major manufacturers. At the time of writing, their future as general standards is uncertain. Until standards are agreed, the problem of dependencies on operating system facilities will remain.
- Most operating systems provide some kind of library mechanism and, often, a set of routines which can be incorporated into user programs.

Standardized and installation libraries:

- (1) Standardized libraries of routines associated with a particular application or operating system. An example of such routines are the NAG library routines for numerical applications. These routines have a standard interface and exactly the same routines are available to all installations which subscribe to the library.
- (2) Installation libraries which consist of routines submitted by users at a particular site. These routines rarely have a standard interface and they are not written in such a way that they may easily be ported from one installation to another.

- 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. There are a number of different file structuring mechanisms enforced by different systems. Systems such as UNIX support only character files whereas other systems consider files to be made up of logical records with many logical records packed into each physical block.
7. The random access primitives supported by different systems vary from one system to another. Some systems may not support random access, others allow random access to individual characters, and yet others only permit random access at the block level.

There is little that programmers can do to make file access over different systems compatible. They are stuck with a set of file system primitives and those parts of the system must be modified if the program is moved to another installation. To reduce the amount of work required, file access primitives should be isolated, whenever possible, in user defined procedures. For example, in UNIX, the mechanism to create a file involves calling a system function called create passing the file name and access permissions as parameters:

```
create ("myfile",0755)
```

creates a file called myfile with universal read and execute access and owner write access In order to isolate this call, a synonymous user function which calls create can be included:

```
access_permissions := "rwxr_x_x"
create_file ("myfile", access_permissions)
```

In UNIX, create_file would simply consist of a single call to the system routine create. On other systems, create_file could be rewritten to reflect the conventions of the system. The parameters to create_file could be translated into the appropriate form for that system.

- It might be imagined that the input/output facilities in a programming language would conceal the details of the operating system input/output routines. Input/output should therefore cause few problems when porting a system from one installation to another.

This is true to some extent. In some programming languages (FORTRAN and COBOL) input/output facilities are defined and each implementation of the language provides these facilities. In other languages, such as Pascal, the input/output facilities are poorly defined or inadequate. As a result, the implementors of a compiler 'extend' the I/O facilities to reflect the facilities provided by the operating system and this leads to portability problems because of incompatible extensions. In particular, the provision of interactive terminal support sometimes causes problems with Pascal as it was developed before interactive terminals were widely used.

- Different systems consider interactive terminals in different ways. In UNIX, a terminal is considered as a special file and file access primitives are used to access it. In other systems, terminals are considered to be devices distinct from files and special terminal access primitives are provided.
 - ❖ ***Advantages in considering a terminal as a file:*** Input and output to and from a program can come from either a terminal or a file on backing store.
 - ❖ ***The disadvantage in considering a terminal as a file:*** The characteristics of a terminal are not exactly those of a file. In fact, a terminal is really like two distinct files, an input file and an output file. If this is not taken into account, portability problems are likely to arise.
- Many systems are made up of a number of separate programs with job control language statements (in UNIX, these are called shell commands) used to coordinate the activities of these programs. Although it might be imagined that these would be relatively small programs, systems like UNIX encourage the use of job control languages as a programming language. Moderately large systems are sometimes written using this notation.
- There is no standardization of job control languages across different systems with the consequence that all job control programs must be rewritten when transferring a system from one installation to another. The difficulties involved in this are exacerbated by the vastly different facilities provided in different languages and may be further compounded by WIMP interfaces such as are used in the Apple Macintosh. Only a move towards a standard operating system will help resolve these problems.

4.3.4 Software reuse

([Horowitz 1984], [Prieto-Diaz 1987]).

- Costs should be reduced as the number of components that must be specified, designed and implemented in a software system is reduced.
- It is possible to reuse specifications and designs.
- Reusing the component might require modification to that component and this, conceivably, could cost as much as component development.
- Advantages of systematic software reuse:
 1. *System reliability is increased.* It can be argued that only actual operational use adequately tests components and reused components, which have been exercised in working systems, should be more reliable than components developed anew.
 2. *Overall risk is reduced.* If a component exists, there is less uncertainty in the costs of using that component than in the costs of developing it. This is an important factor for project management as it reduces the overall uncertainty in the project cost estimation.
 3. *Effective use can be made of specialists.* Instead of application specialists joining a project for a short time and often doing the same work with different projects,

as proposed in [Brooks 1975], these specialists can develop reusable components which encapsulate their knowledge.

4. *Organizational standards can be embodied in reusable components.* For example, say a number of applications present users with menus. Reusable components providing these menus mean that all applications present the same menu formats to users.
5. *Software development time can be reduced.* It is often the case that bringing a system to market as early as possible is more important than overall development costs. Reusing components speeds up system production because both development and validation time should be reduced.

- While standard subroutine libraries have been very successful in a number of application domains in promoting reuse, systematic software reuse as a general software development technique is not commonly practiced.

Technical and managerial impediments to generalization of systematic software reuse:

1. We do not really know what are the critical component attributes which make a component reusable. We can guess that attributes such as environmental independence are essential but assessing the reusability of a component in different application domains is difficult.
 2. Developing generalized components is more expensive than developing a component for a specific purpose. Thus, developing reusable components increases project costs and manager's main preoccupation is keeping project costs down. To develop reusable components requires an organizational policy decision to increase short-term costs for possible, unquantifiable, long-term gain, and senior management is often reluctant to make such decisions.
 3. Some software engineers are reluctant to accept the advantages of software reuse and prefer to write components afresh as they believe that they can improve on the reusable component. In most cases, this is probably true, but only at the expense of greater risk and higher costs.
 4. We do not have a means of classifying, cataloguing and retrieving software components. The critical attribute of a retrieval system is that component retrieval costs should be less than component development costs.
- The importance of systematic software reuse is now widely recognized and there is a great deal of research effort being expended on solving some of the problems of reuse.
- In one application domain, however, reuse is now commonplace. This is in data processing systems where fourth-generation languages (Martin, 1985) are widely used. Broadly, many data processing applications involve abstracting information from a database, performing some relatively simply information processing and then producing reports using that information. This stereotypical structure was recognized and components to carry out these operations devised.

On top of these was produced some control language (there is a close analogy here with the UNIX shell) which allowed programs to be specified and a complete application to be generated.

This form of reuse is very effective but it has not spread beyond the data processing systems domain. The reason for this seems to be that similar stereotypical situations are less obvious in other domains and there seems to be less scope for an approach to reuse based on applications generators.

The initial notion of reuse centred around the reuse of subroutines. The problem with reusing subroutine components is that it is not practicable to embed environmental information in the component so that only functions whose values are not dependent on environmental factors may be reused.

Subroutines are not an adequate abstraction for many useful components such as a set of routines to manipulate queues and it is likely that software reuse will only become widespread when languages such as Ada with improved abstraction facilities become widely used. Indeed, it is probable that the most effectively reusable component is the abstract data type which provides operations on a particular class of object.

Given this assumption, what are useful guidelines to adopt when developing abstract data types for reuse? It can be assumed that abstract data types are independent entities which behave in the same way irrespective of the environment in which they are used (except for timing considerations, perhaps). Thus, we have to be concerned about providing a complete and consistent set of operations for each abstract type.

Unfortunately, there is a large number of possible operations on each abstract type and there is no way in which the component developer can anticipate every possible use. Some broad guidelines as to what facilities should be provided are:

1. An operation should be included to create and initialize instances of the abstract type. Creation is sometimes accomplished simply by language declarations but the programmer should not rely on default initializations provided by the implementation language.
2. For each attribute of the abstract type, access and constructor functions should be provided. Access functions return attribute values; constructor functions allow attribute values to be changed.
3. Operations to print instances of the type and to read and write type instances to and from filestore should be provided.
4. Assignment and equality operations should be provided. This may involve defining equality in some arbitrary way which should be specified in the abstract type documentation.
5. For every possible exception condition which might occur in type operations, a test function should be provided which allows that condition to be checked for before initiating the operation. For example, if an operation on lists might fail if the list is empty, a function should be provided which allows the user to check if the list has any members.

6. If the abstract type is a composite type (that is, is made up of a collection of other objects), operations to add objects to and to delete objects from the collection should be provided. If the collection is ordered, multiple add and delete functions should be provided. For example, for a list type, operations should be available to add and delete an element to and from the front and the end of the list.
7. If the abstract type is a composite type, functions to provide information about attributes of the composition (such as size) should be provided.
8. If the abstract type is a composite type, an iterator should be provided which allows each element of the type to be visited. Iterators allow each component of a composite to be visited and evaluated without destroying the structure of the entity.
9. Wherever possible, abstract types should be parameterized using generic types. However, this is only possible when parameterization is supported by the implementation language.

4.4 Computer-aided software engineering

4.4.1 CASE workbenches

- CASE workbench systems are designed to support the analysis and design stages of the software process.
- These systems are oriented towards the support of graphical notations such as used in the various design methods. They are either intended for the support of a specific method, such as Structured Design, or support a range of diagram types which encompasses those used in the most common methods.

Typical components of a CASE workbench:

1. A *diagram editing system* that is used to create data-flow diagrams, structure charts, entity-relationship diagrams, etc. The editor is not just a simple drafting tool but is aware of the types of entities in the diagram. It captures information about these entities and saves this information in a central repository (sometimes called an encyclopaedia). The design editing system discussed in previous chapters is an example of such a tool.
2. *Design analysis and checking facilities* that process the design and report on errors and anomalies. As far as possible these are integrated with the editing system so that the user may be informed of errors during diagram creation.
3. *Query language facilities* that allow the user to browse the stored information and examine completed designs.
4. *Data dictionary facilities* that maintain information about named entities used in a system design.
5. *Report generation facilities* that take information from the central store and automatically generate system documentation.

6. *Form generation tools* that allow screen and document formats to be specified.
7. *Import/export facilities* that allow the interchange of information from the central repository with other development tools.
8. Some systems support *skeleton code generators* which generate code or code segments automatically from the design captured in the central store.

CASE workbench systems, like structured methods, have been mostly used in the development of data processing systems, but there is no reason why they cannot be used in the development of other classes of system. Chikofsky and Rubenstein ([Chikofsky 1988]) suggest that productivity improvements of up to 40% may be achieved with the use of such systems. They also suggest that, as well as these improvements, the quality of the developed systems is higher, with fewer errors and inconsistencies, and the developed systems are more appropriate to the user's needs.

Deficiencies in current CASE workbench products tools (Martin 1988):

1. The workbenches are not integrated with other document preparation tools such as word processors and desktop-publishing systems. Import/export facilities are usually confined to ASCII text.
2. There is a lack of standardization which makes information interchange across different workbenches difficult or impossible.
3. They lack facilities which allow a method to be tailored to a particular application or class of application. For example, it is not usually possible for users to override a built-in rule and replace it with their own.
4. The quality of the hard-copy documentation which is produced is often low. Martin observes that simply producing copies of screens is not good enough and that the requirements for paper documentation are distinct from those for screen documentation.
5. The diagramming facilities are slow to use so that even a moderately complex diagram can take several hours to input and arrange. He suggests that there is a need for automated diagramming and diagram arrangement given a textual input.

A more serious omission, from the point of view of large-scale software engineering, is the lack of support for configuration management provided by these systems. Large software systems are in use for many years and, in that time, are subject to many changes and exist in many versions.

Configuration management is probably the most critical management activity in the software engineering process. Configuration management tools allow individuals versions of a system to be retrieved, support system building from components, and maintain relationships between components, and their documentation.

The user of a CASE workbench is provided with support for aspects of the design process, but these are not automatically linked to other products such as source code,

test data suites and user documentation. It is not usually possible to create multiple versions of a design and to track these, nor is there a straightforward way to interface the workbench systems with other configuration management tools.

The lack of data standards makes it difficult or impossible to transfer information in the central repository to other workbench systems. This means that users may be faced with the prospect of maintaining obsolete CASE workbenches and their supporting computers for many years in order to maintain systems developed using these workbenches. This problem has not yet manifested itself because of the relative newness of these systems but is likely to arise as new, second-generation CASE products come onto the market.

4.4.2 Text editing systems

- The function of a text editor is to enable the user to create and modify files kept online in the system, and most environments used for software development offer a number of different editors.
- Because this is such a common activity, the power of the editor contributes significantly to the productivity of the software engineer.

Although some editors, such as UNIX's *vi* editor, have facilities which are designed to support program preparation, most editors are general-purpose text preparation systems. This naturally means that it is possible for the user to prepare syntactically incorrect programs with these editors but it has the advantage that the same editing system may be used to produce any type of document. There is no need for the user to learn more than one editing system.

To support program preparation and editing, some work has been done in developing language-oriented editors (sometimes called structure editors) designed to prepare and modify programs in a specific programming language. An example of such a system is the Cornell Program Synthesizer, described by Teitelbaum and Reps ([Teitelbaum 1981]), which is intended to help beginners prepare programs written in a subset of PL/1 called PL/C. Such a system must include a PL/C syntax analyser as well as editing features. Rather than manipulating unstructured text, the system actually manipulates a tree structure representing the program. In fact, the Cornell system is more than just a syntax-directed editor. It is a complete language-oriented environment with integrated editing, translation and program execution facilities.

Such systems are valuable for beginners wrestling with the idiosyncrasies of current programming languages. However, they do have limitations. Passing information about context-sensitive language constructs is difficult, particularly in large programs where the whole program is not visible on the screen. More fundamentally, perhaps, these systems do not recognize the fact that many experienced programmers work by laying out an (incorrect) program skeleton, then filling in that skeleton correcting inaccuracies and inconsistencies. Structure editors force a mode of working on the user which does not always conform to professional practice.

Some of the limitations of existing editing systems (particularly for program editing) are a result of display inadequacies where simple 80 by 25 character terminals are used.

Such hardware forces documents or programs to be viewed sequentially. A sequential representation is hardly ever the most appropriate for programs where the reader wishes to look at and modify program parts which are logically rather than textually related. Furthermore, it may be appropriate to view and edit programs as diagrams rather than text. As bit-mapped workstations become the normal tool of the software engineer, new editing systems such as that described by Shneiderman *et al.* ([Shneiderman 1986]) will become commonly used.

Such systems use multiple windows to present different parts of the program and allow interaction in terms of program concepts. For example, one window might show the code of a Pascal procedure, with the program declarations displayed beside this in another window. Changing one part of the program (for example, modifying a procedure parameter list) results in related parts being immediately identified and presented for modification.

Such systems are impractical in development environments such as UNIX where the files are untyped. However, in an integrated development environment, stored entities are normally typed so that the editing system can examine what is being edited and present appropriate facilities to the user. Thus, as a user moves through a document containing text, programs, tables and diagrams, the editor identifies the working context and gives the user editing facilities geared to the type of object being edited. There is no need to prepare diagrams separately and paste them into documents or to move explicitly between program and text editors.

4.4.3 Language processing systems

- Language processing systems are used to convert programs to machine code.
- The provision of a helpful compilation system reduces the costs of program development by making program errors easier to find and by producing program listings which include information about program structure as seen by the compiler.
- The error diagnostic facilities of a compiler are partially dependent on the language being compiled. A Pascal compiler, for example, can detect many more errors than a FORTRAN compiler. Not only are the rules that govern the validity of a program more strict for Pascal than for FORTRAN, but the Pascal programmer must also supply more information to the compiler about the entities to be manipulated by the program. This information allows the compiler to detect forbidden operations on these entities.

As well as providing information to the programmer, a compilation system must also generate efficient machine code. This latter task involves a good deal of program analysis and can be very time consuming. This has the consequence that it is generally uneconomic to carry out this operation for anything apart from completely developed programs.

- A software engineering environment, might contain two compatible compilers for each language - a development compiler and an optimizing compiler.

Development compilers should be written to compile code as quickly as possible and to provide the maximum amount of diagnostic information to the programmer.

Optimizing compilers, on the other hand, should be tailored to generate efficient

machine code without considering compilation speed and diagnostic facilities. Programs are developed using the development system and, when complete, the optimizing system is used to produce the final version of the program for production use.

- Within the confines of the language being processed, development compilers should provide as much information as possible about the program being compiled:
 1. The compiler listing of the program should associate a line number with each program line.
 2. When a program syntax or semantic error is discovered, the compiler should indicate where it found the error and what the error appears to be. It may also be appropriate to indicate the possible cause of the error.
 3. The compiler should include directives which allow the programmer some control over the program listing generated by the compiler. These directives should allow the suppression of parts of the listing, control over the pagination of the listing, and the enhancement of program keywords by **bold** printing or underlining.
 4. When a program in a block-structured language is compiled, the compiler should indicate the lexical level at the beginning and the end of each block. This allows misplaced **begin/end** brackets to be easily identified.
 5. The compiler should separate source text provided by the user from information provided by the compiler. This can be accomplished by delimiting the input source using special characters such as '|', and prefacing compiler messages by some string of special characters such as '%%'.
 6. The compiler should identify where each procedure in a program starts and finishes. When a program listing is searched for a particular procedure, the location of that procedure is often not immediately obvious because the name of the procedure is not distinguished from the remainder of the source text. When compiling a procedure heading, the procedure name should be abstracted and, as well as listing the procedure heading normally, the procedure name should be reprinted so that it stands out from the rest of the program text.

These facilities were supported in the XPL compiler ([McKeeman 1970]), which was part of a compiler construction system. It is unfortunate that many of the display facilities provided by XPL have not been taken up by later compilers.

- If an environment supports both development and optimizing compilers, there is no need for the optimizing compiler to provide comprehensive diagnostic facilities. Rather, given that code optimization is a time-consuming business, the compiler may allow the user to specify the degree of optimization to be carried out by the compiler or whether time or space considerations are most important. Ada has a specific construct called a pragma which, amongst other things, provides some facilities for the user to control compiler optimization.

4.4.4 Separate compilation

- Separate compilation of program units: Units may be compiled separately and subsequently integrated to form a complete program.
- The integration process is carried out by another software tool known as a linker or link editor. Without the facility of separate compilation, a programming language should not be considered as a viable language for software engineering.
- A large program may consist of several hundred thousand lines of source code and it may take hours or even days to compile the complete program. If every program unit needed to be recompiled each time any one of the units was changed, this would impose significant overhead and increase the costs of program development, debugging and maintenance. If separate compilation is available, compiling the whole system is unnecessary. Only modified units need be recompiled and relinked.

4.4.5 Compiler support tools

- The compilation process involves an analysis of the support text and, given this analysis, it is possible to provide additional information for the programmer and to lay out the program code, automatically, in a standard way. Commonly, these facilities are embedded in a compilation system. Other analysis tools, called static program analysers, are intended to detect anomalies in the source code.

Example of compiler support tool:

Program crossreferencer.

Such a tool provides a collated listing of the names used in the program, the types of the named objects, the line in the program where each name is declared, and the line numbers where a reference is made to that object. More sophisticated cross-referencers can also provide, for each procedure in the program, a list of the procedure parameters and their types, the procedure local variables and the global variables referenced in the procedure.

This latter facility is particularly useful to the programmer who must modify the value of some global variable. By examining the cross-reference listing, either manually or with an automatic tool, those procedures which reference that variable can be identified. They can be checked to ensure that global variable modification will not adversely affect their actions.

In addition to cross-reference systems, other source code analysis tools include layout programs (prettyprinters) which set out programs in some standard way. Prettyprinters are tools which incorporate standard layout conventions, and it is sometimes suggested that the availability of such tools makes disciplined layout on the part of the programmer redundant. Furthermore, if all listings are produced using a prettyprinter, the maintenance programmer will not be presented with the problem of getting used to different layout conventions.

The problem with most prettyprinting systems is that they incorporate a set of conventions which have been invented by the tool designer and which are rarely explicitly specified. This means that, if an organization has existing standards, it is not possible to tailor prettyprinters to these standards. This may preclude the use of a prettyprinter and the programmers may revert to manual code layout.

All tools for static program analysis are language-oriented and must include a language syntax analyser. This has led to suggestions that the process of analysis normally

carried out by the compiler should be factored out. Program analysis would be distinct from compiling and the analyser output would be processable by translation tools, editors, analysis tools, etc.

4.5 Incremental implementation

- The basic idea of incremental implementation is to blur the distinction between the design and implementation phases rather than to uphold the strict separation of phases that the classical sequential software life-cycle model requires.
- The recommendation of this approach is founded on the experience-based assumption that design and implementation decisions strongly affect one another, and thus a rigorous separation of design and implementation does not accomplish its goal. In many cases only the implementation can determine whether the decomposition structure of the design proves adequate for the problem.
- Incremental implementation means that after each design step that is relevant to the architecture, the current system architecture is verified on the basis of real cases. This means that the interplay of the system components specified in the design (in the form of interface specifications) is verified. To make this possible, the system components (i.e., their input/output behaviour) is simulated or realised as a prototype. Naturally the developer must devote thought to the implementation of individual components. The knowledge gained in the process, as far as possible, is translated directly into the implementation of the components, or documented for the subsequent implementation process. If there is any doubt concerning the feasibility of a component, then the design process is interrupted and these components are implemented. Only when their implementation and their embedding in the previous system architecture have been checked, is the design process continued, or the architecture is adapted according to the knowledge gained in the implementation of the component.

The efficiency of this approach depends on the extent to which it is possible to integrate the system components, which can be written in different formalisms and completed to varying degrees, to an overall system in order to carry out experiments close to reality. Some system components, e.g., the user interface or the data model, might be present in the form of prototypes; other components, which might stem from an existing component library or already exist as a complete implementation, are present in the form of executable code; still other system components might only be available as interface specifications. For the validation of the current system design, whenever the user interface is employed, the corresponding prototypes need to be activated. If a system component is available in compiled form, it needs to be executed directly. A system component for which only the interface specifications are available must be simulated. This is only possible with a software development environment that supports the integration and execution process for such *hybrid systems*. Describing such a development environment and its application scenarios would exceed the scope of this book. The interested reader can find both in [Bischofberger 1992].

Objectives

This chapter describes various test methods and discusses how testing and debugging should be carried out and documented. The testing process and the stages of testing and the advantages and disadvantages of top-down testing are discussed. A formal method of program verification is described.

Contents

- 5.1 Test methods
 - 5.1.1 Verification of algorithms
 - 5.1.2 Static program analysis
 - 5.1.3 Dynamic testing
 - 5.1.4 Black-box and white-box testing
 - 5.1.5 Top-down and bottom-up testing
 - 5.2 Mathematical program verification
 - 5.3 Debugging
 - References and selected reading
-

5.1 Test methods

In carrying out testing we will wish to address some or all of the following questions:

- How is functional validity tested?
- What *classes* of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?
- Which activities are necessary for the systematic testing of a software system?
- What is to be tested?

- ❖ *System specifications,*
- ❖ *Individual modules,*
- ❖ *Connections between modules,*
- ❖ *Integration of the modules in the overall system*
- ❖ *Acceptance of the software product.*

Testing the system specifications

- The system specifications provide the foundation for the planning and execution of program tests.
- The comprehensibility and unambiguity of the system specifications is the prerequisite for testing a software system.
- The goal of the specification test is to examine the completeness, clarity, consistency and feasibility of the system specifications.

***Techniques of cause and effect* ([Hughes 1976]).**

- *Causes* are conditions or combinations of conditions that are prescribed or that have arisen
- *Effects* are results or activities.
- The specification test is intended to show whether causes (data, functions) are specified for which no effects (functions, results) are defined or vice versa.
- Prototypes of user interface and system components play an important role in testing the system specifications; they permit experimental inspection of the system specifications.
- The specifications test must always be carried out in cooperation with the user.

Testing modules

- Modules encapsulate data structures and operations that work with these data structures.
- The *goal* of the module test is to *identify all deviations* of the implementation from the module specifications.
- Module testing requires first testing the individual functions and then their interplay.
- Modules are not independently programs; their execution requires the existence of other modules (due to the networking of modules through import and inheritance relationships).
- The test can only be executed if the environment of the module already exists or can be simulated.
- Part of work in module testing consists of creating a suitable test environment that permits of the module, examination of the results of its invocation, and simulation of modules yet to be developed (see Figure 5.1).

- Keep the test as simple as possible, for increasing complexity also raises the probability the environment itself contains errors.

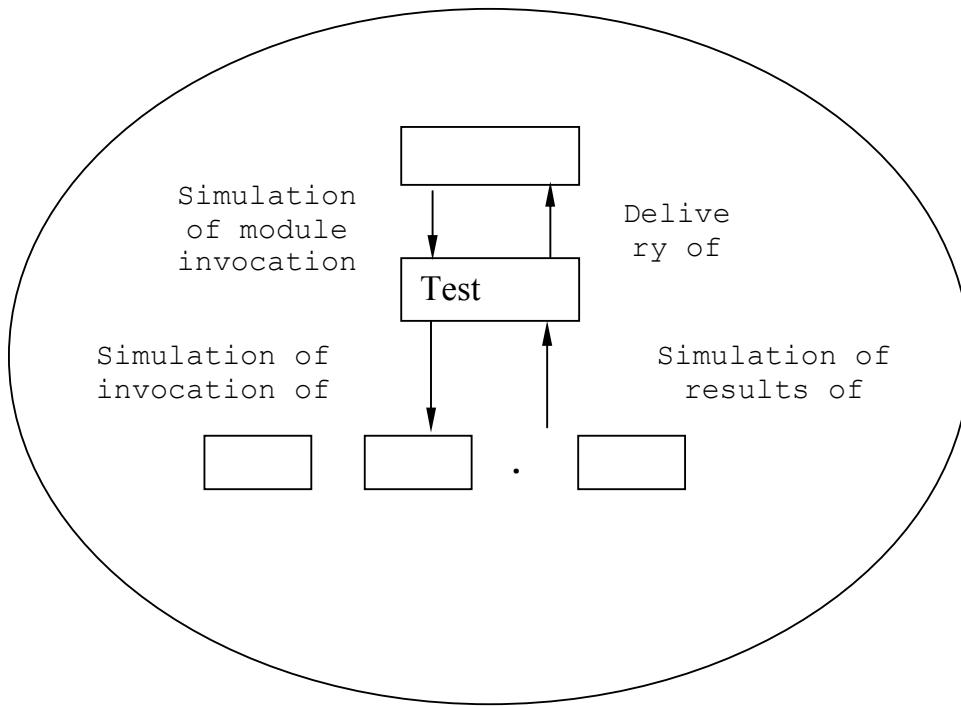


Figure 5.1 Test object and test environment

Testing module connections (testing of subsystems)

- A *subsystem* comprises a problem-specific collection of individual (tested) modules.
- The *goal* is to test the *connections between the individual modules* of a system.
- The test encompasses checking the correctness of module communication hardly differs in its approach from module testing.
- Subsystems require the creation of test environments. After testing subsystems, several subsystems are combined to a (hierarchically superordinate) subsystem, and the connections between subsystems are tested (*integration test*).

Test of the overall system

- Test the integration of all subsystems.
- The *goal* is to *detect all deviations* of the system behavior from that in the requirements definition.
- The goal is not only to test the completeness of the user requirements and the correctness of the results, but also to test whether the software system is reliable and robust with respect to erroneous input data.
- The system's adherence to nonfunctional requirements, e.g., the required efficiency, must also be tested.

- The scope and number of tests as well as the selection of appropriate test data vary from case to case; the present state of the art offers no universal recipe for testing software systems.
- The effort vested in testing of the overall system must be in some proportion to the amount of damage that erroneous operation of the software systems can inflict.

Acceptance test

- The development of a software product ends with the *inspection (acceptance test)* by the user.
- At inspection the software system is tested with *real data under real conditions of use*.
- The *goal* of the inspection is to *uncover all errors* that arose from such sources as misunderstandings in consultations between users and software developers, poor estimates of application-specific data quantities, and unrealistic assumptions about the real environment of the software system.

5.1.1 Verification of algorithms

- Two important questions:
 - Whether the chosen approach will actually lead to the required solution?
 - Whether the already developed components are correct, i.e., whether they fulfill the specifications?
- Ideally we would like to be able to prove at any time that for every imaginable input the algorithms included in the design and their interplay will deliver the expected results according to the specification.
- Since today's large software systems permit the complete description of neither the input set nor the anticipated result set (due to combinatorial explosion), such systems can no longer be completely tested. Thus efforts must be made to achieve as much clarity as possible already in the design phase about the correctness of the solution.
- Proof of correctness cannot be handled without formal and mathematical tools ([McCarthy 1962], [Naur 1966], [Hoare 1969], [Manna 1969], [Knuth 1973], [Dijkstra 1976], see overview in [Elspas 1972].)
- The use of a verification procedure forces the software engineer to reproduce all design decisions and thus also helps in finding logical errors. Verification is an extremely useful technique for early detection of design errors and also complements the design documentation. Verification itself can be fallible, however, it cannot replace testing of a software product.
- Verification can be used successfully to prove the correctness of short and simple algorithms. For the production of larger software systems, the difficulties rise so sharply that it should be clear that verification fails as a practical test aid.

5.1.2 Static program analysis

- *Static program analysis* seeks to detect errors without direct execution of the test object.

- The activities involved in static testing concern *syntactic, structural and semantic analysis of the test object* (compare [Ramamoorthy 1974] and [Schmitz 1982]).
- The goal is to localize, as early as possible, error-prone parts of the test object.
- The most important activities of static program analysis are:
 - Code inspection
 - Complexity analysis
 - Structural analysis
 - Data-flow analysis

Code inspection

- *Code inspection* is a useful technique for localizing design and implementation errors. Many errors prove easy to find if the author would only read the program carefully enough.
- The idea behind code inspection is to have the author of a program discuss it step by step with other software engineers.
- Structure of the four person team ([Fagan 1976]):
 1. An experienced software engineer who is not involved in the project to serve as moderator
 2. The designer of the test object
 3. The programmer responsible for the implementation
 4. The person responsible for testing

The moderator notes every detected error and the inspection continues.

- The *task of the inspection team* is to *detect, not to correct*, errors. Only on completion of the inspection do the designer and the implementor begin their correction work.

Complexity analysis

- The *goal of complexity analysis* is to *establish metrics* for the complexity of a program ([McCabe 1976], [Halstead 1972], [Blaschek 1985])
- These metrics include complexity measures for modules, nesting depths for loops, lengths of procedures and modules, import and use frequencies for modules, and the complexity of interfaces for procedures and methods.
- The results of complexity analysis permit statements about the quality of a software product (within certain limits) and localization of error-prone positions in the software system.
- Complexity is difficult to evaluate objectively; a program that a programmer perceives as simple might be viewed as complex by someone else.

structural analysis

- The goal of *structural analysis* is to uncover structural anomalies of a test object.

Data-flow analysis

- Data-flow analysis is intended to help discover data-flow anomalies.
- Data-flow analysis provides information about whether a data object has a value before its use and whether a data object is used after an assignment.
- Data-flow analysis applies to both the body of a test object and the interfaces between test objects.

5.1.3 Dynamic testing

- For *dynamic testing* the test objects are executed or simulated.
- Dynamic testing is an imperative process in the software life cycle. Every procedure, every module and class, every subsystem and the overall system must be tested dynamically, even if static tests and program verifications have been carried out.
- The activities for dynamic testing include:
 - Preparation of the test object for error localization
 - Availability of a test environment
 - Selection of appropriate test cases and data
 - Test execution and evaluation

5.1.4 Black-box and white-box testing

Every test object can be tested in two principal ways to determine whether it fulfills its specifications:

1. Black-box test (functional test or exterior test): Test of the input/output behavior without considering the inner structure of the test object (interface test)
2. White-box test (structure test or interior test): Test of the input/output behavior considering the inner structures (interface and structure test)

Black-box tests

- *Black-box testing* focuses on the functional requirements of the software.
- Black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
 1. Incorrect or missing functions,
 2. Interface errors,
 3. Errors in data structures or external database access,
 4. Performance errors, and

5. Initialization and termination errors.

- *Black-box tests* serve to detect deviations of a test object from its specification. The selection of test cases is based on the specification of the test object without knowledge of its inner structure. It is also important to design test cases that demonstrate the behavior of the test object on erroneous input data.
- Criteria for test cases [Myers 1979]
 1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing
 2. Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.
- The black-box test provides no information about whether all functions of the test object are actually necessary or whether data objects are manipulated that have no effect on the input/output behavior of the test object. Such indicators of design and implementation errors emanate from the white-box test.

White-box tests

- *White-box testing* is one of the most important test methods. For a limited number of program paths, which usually suffices in practice, the test permits correct manipulation of the data structures and examination of the input/output behavior of test objects.
- In white-box testing the test activities involve not only checking the input/output behavior, but also examination of the inner structure of the test object.
- The goal is to determine, for every possible path through the test object, the behavior of the test object in relation to the input data.
- Test cases are selected on the basis of knowledge of the control flow structure of the test object.
- The selection of test cases must consider the following:
 - ❖ Every module and function of the test object must be invoked at least once
 - ❖ Every branch must be taken at least once
 - ❖ As many paths as possible must be followed
- It is important to assure that every branch is actually taken. It does not suffice to merely assure that every statement is executed because errors in binary branches might not be found because not all branches were tested.
- It is important to consider that, even for well-structured programs, in practice it remains impossible to test all possible paths, i.e., all possible statement sequences of a program ([Pressman 1987], see Figure 5.2)

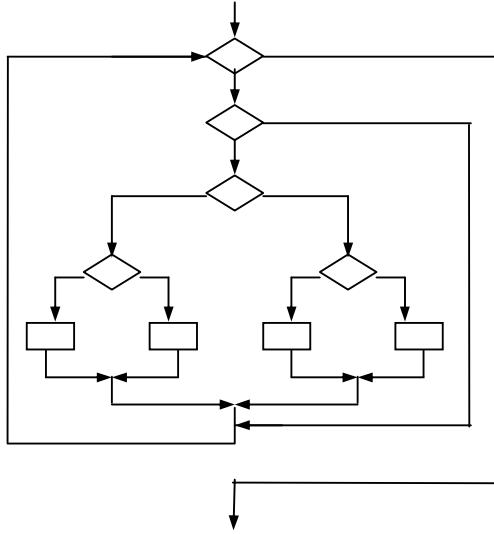


Figure 5.2 Control flowchart: for 10 iterations
there are almost one million paths

5.1.5 Top-down and bottom-up testing

Top-down testing

Method

- The control module is implemented and tested first.
- Imported modules are represented by surrogate modules.
- Surrogates have the same interfaces as the imported modules and simulate their input/output behavior.
- After the test of the control module, all other modules of the software systems are tested in the same way; i.e., their operations are represented by surrogate procedures until the development has progressed enough to allow implementation and testing of the operations.
- The test advances stepwise with the implementation. Implementation and phases merge, and the integration test of subsystems becomes superfluous.

The advantages

- Design errors are detected as early as possible, saving development time and costs because corrections in the module design can be made before their implementation.

- The characteristics of a software system are evident from the start, which enables a simple test of the development state and the acceptance by the user.
- The software system can be tested thoroughly from the start with test cases without providing (expensive) test environments.

The drawbacks

- Strict top-down testing proves extremely difficult because designing usable surrogate objects can prove very complicated, especially for complex operations.
- Errors in lower hierarchy levels are hard to localize.

Bottom-up test

Method

- *Bottom-up testing* inverts the top-down approach.
- First those operations are tested that require no other program components; then their integration to a module is tested.
- After the module test the integration of multiple (tested) modules to a subsystem is tested, until finally the integration of the subsystems, i.e., the overall system, can be tested.

The advantages

- The advantages of bottom-up testing prove to be the drawbacks of top-down testing (and vice versa).
- The bottom-up test method is solid and proven. The objects to be tested are known in full detail. It is often simpler to define relevant test cases and test data.
- The bottom-up approach is psychologically more satisfying because the tester can be certain that the foundations for the test objects have been tested in full detail.

The drawbacks

- The characteristics of the finished product are only known after the completion of all implementation and testing, which means that design errors in the upper levels are detected very late.
- Testing individual levels also inflicts high costs for providing a suitable test environment.

5.2 Mathematical program verification

- If programming language semantics are formally defined, it is possible to consider a program as a mathematical object.
- Using mathematical techniques, it is possible to demonstrate the correspondence between a program and a formal specification of that program.
- Program is proved to be correct with respect to its specification.
- Formal verification may reduce testing costs, it cannot replace testing as a means of system validation.

- Techniques for proving program correctness and axiomatic approaches:
[McCarthy 1962], [Hoare 1969], [Dijkstra 1976], [Manna 1969].

The basis of the axiomatic approach

- Assume that there are a number of points in a program where the software engineer can provide assertions concerning program variables and their relationships. At each of these points, the assertions should be invariably true. Say the points in the program are $P(1), P(2), \dots, P(n)$. The associated assertions are $a(1), a(2), \dots, a(n)$. Assertion $a(1)$ must be an assertion about the input of the program and $a(n)$ an assertion about the program output.
- To prove that the program statements between points $P(i)$ and $P(i+l)$ are correct, it must be demonstrated that the application of the program statements separating these points causes assertion $a(i)$ to be transformed to assertion $a(i+l)$.
- Given that the initial assertion is true before program execution and the final assertion is true after execution, verification is carried out for adjacent program statements.

3 Example 5.1

Axioms for proving program correctness

$$(A1) \quad \text{if } \{P\} A \{R\} \text{ and } \{R\} B \{Q\} \\ \text{then } \{P\} A; B \{Q\}$$

$$(A2) \quad \text{if } P \Rightarrow Q[x/e] \text{ then } \{P\} x:=e \{Q\}$$

where $Q[x/e]$ is received from Q by replacing all the occurrences of variable x with e

$$(A3.1) \quad \text{if } \{P \wedge E\} A \{Q\} \text{ and } \{P \wedge !E\} B \{Q\} \\ \text{then } \{P\} \text{ if } E \text{ then } A \text{ else } B \{Q\}$$

where $!E$ is not E

$$(A3.2) \quad \text{if } \{P \wedge E\} A \{Q\} \text{ and } P \wedge !E \Rightarrow Q \\ \text{then } \{P\} \text{ if } E \text{ then } A \{Q\}$$

$$(A4) \quad \text{if } \{P \wedge E\} A \{P\} \\ \text{then } \{P\} \text{ while } E \text{ do } A; \{P \wedge !E\}$$

Example 5.2 The greatest common divisor of two natural numbers

$gcd(a,b)$ (*The Euclidian Algorithm*)

```

x:=a;
y:=b;
{ Invariant: gcd(x,y) = gcd(a,b) }
while y <> 0 do
begin
```

```

{ (gcd(x,y) = gcd(a,b)) ∧ y <> 0 }
{ gcd(x,y) = gcd(y,x mod y) = gcd(a,b) }
r := x mod y;
{ gcd(x,y) = gcd(y,r) = gcd(a,b) }

x := y;
{ gcd(x,y) = gcd(x,r) = gcd(a,b) }

y := r;
{ gcd(x,y) = gcd(a,b) }

end;

{ (gcd(x,y) = gcd(a,b)) ∧ y = 0 ⇒ (gcd(x,0) = gcd(a,b) = x) }

{ x = gcd(a,b) }

```

5.3 Debugging

- Software testing is a process that can be systematically planned and specified. Test design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.
- *Debugging* occurs as a consequence of successful testing. When a test case uncovers an error, debugging is the process that results in the removal of the error.
- Debugging is not testing, but it always occurs as a consequence of testing.
- The debugging process begins with the execution of a test case (Figure 5.3).
- The debugging process attempts to match symptom with cause, thereby leading to error correction.
- Two outcomes of the debugging:
 1. The cause will be found, corrected, and removed,
 2. The cause will not be found.

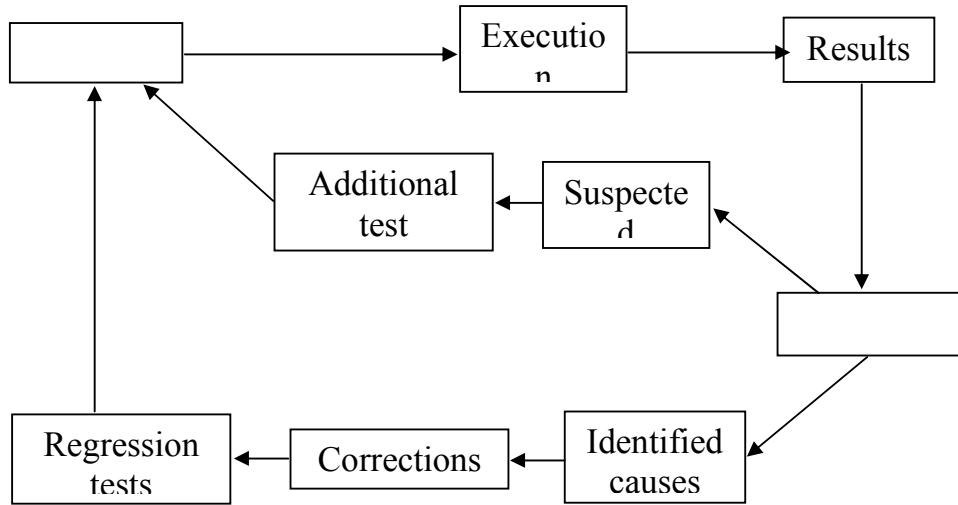


Figure 5.3 Debugging

Characteristics of bugs ([Cheung 1990])

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging approaches ([Bradley 1985], [Myers 1979])

- Brute force
- Backtracking

- Cause elimination
- We apply brute force debugging methods when all else fails.
- Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.
- Backtracking is a fairly common debugging

Objectives

This chapter is concerned with documentation which is an integral part of a software systems. The structure of user and system documentation is described and the importance of producing high-quality documentation is emphasized. The final parts of the chapter comment on document maintainability and document portability.

Contents

- 6.1 User documentation
 - 6.2 System documentation
 - 6.3 Document quality
 - 6.4 Document maintenance
 - 6.5 Document portability
- References and selected reading

There are two classes of documentation associated with a computer system. These classes are user documentation which describes how to use the system and system documentation which describes the system design and implementation.

The documentation provided with a system can be useful at any stage in the lifetime of the system.

All kinds of documentation need effective indexing. A good index, which allows the user to find the information he or she needs, is probably the most useful feature that can be provided but, sadly, is often the most neglected part of document production. A comprehensive index can make a badly written document usable but, without an index, even the best written prose is unlikely to convince the reader that the document is effective.

6.1 User documentation

User documentation consists of the documents which describe the functions of the system, without reference to how these functions are implemented.

User documentation should be structured so that it is not necessary to read almost all the documentation before starting to use the system. It should be integrated with

on-line help but it is not sufficient simply to print help frame text as user documents.

The documentation provided for system users is usually the first contact they have with the system.

It should provide an accurate initial impression of the system.

Five constituents of user documentation: (Figure 6.1)

- *A functional description*, which explains what the system can do.
- *An installation document*, which explains how to install the system and tailor it for particular hardware configurations.
- *An introductory manual*, which explains, in simple terms, how to get started with the system.
- *A reference manual*, which describes in detail all of the system facilities available to the user and how these facilities can be used.
- *A system administrator's guide* (if necessary), explaining how to react to situations which arise while the system is in use and how to carry out system housekeeping tasks such as making a system backup.

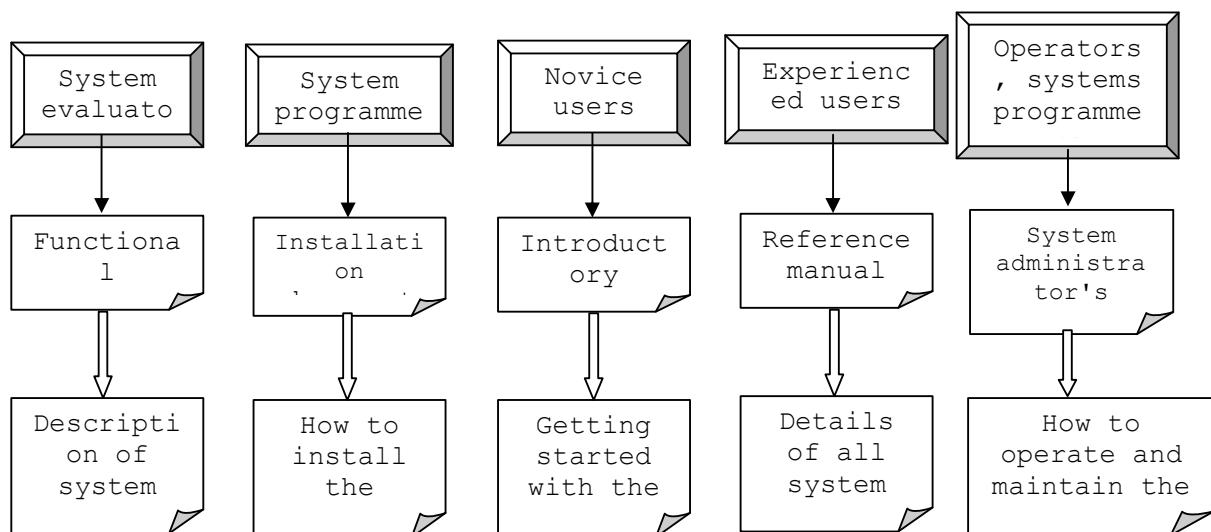


Figure 6.1 System document roles ([Sommerville 1996])

Functional description

- ❖ Outlines the system requirements,
- ❖ Outlines the aims of the system designers,
- ❖ Describes what the system can do,

- ❖ Describes what the system cannot do,
- ❖ Introduces small, self-evident examples wherever possible,
- ❖ Diagrams should be plentiful.

Introductory manual

- ❖ Provides an overview of the system,
- ❖ Enables users to decide if the system is appropriate for their needs,
- ❖ Presents an informal introduction to the system,
- ❖ Describes how to get started on the system and how the user might make use of the common system facilities,
- ❖ Tells the system user how to get out of trouble when things go wrong,
- ❖ It should be liberally illustrated with examples,

Reference manual

- ❖ The system reference manual is the definitive document on system usage,
- ❖ The reference manual should be complete,
- ❖ Wherever possible, formal descriptive techniques should be used to ensure that completeness is achieved,
- ❖ The writer of this manual may assume:
 - the reader is familiar with both the system description and introductory manual,
 - the reader has made some use of the system and understands its concepts and terminology.
- ❖ The system reference manual should also describe
 - the error reports generated by the system,
 - the situations where these errors arise and, if appropriate, refer the user to a description of the facility which was in error.
- ❖ A comprehensive index is particularly important in this document.

Installation document

- ❖ The system installation document should provide full details of how to install the system in a particular environment.

- ❖ It must contain a description of the machine-readable media on which the system is supplied - its format, the character codes used, how the information was written, and the files making up the system.
- ❖ It should describe
 - the minimal hardware configuration required to run the system,
 - the permanent files which must be established,
 - how to start the system, and
 - the configuration dependent files which must be changed in order to tailor the system to a particular host system.
- ❖ Describes the messages generated at the system console and how to react to these messages,
- ❖ Explains the operator's task in maintaining the system.

Other easy-to-use documentation

- Quick reference card listing available system facilities and explaining how to use them.
- On-line help systems.

6.2 System documentation

System documentation encompasses all of the documents describing the implementation of the system from the requirements specification to the final acceptance test plan:

- Documents describing the design,
- Documents describing the implementation,
- Documents describing testing.

System documentation is essential for understanding and maintaining a software system.

The documentation should be

- structured, and
- described with overviews leading the reader into more formal and detailed descriptions of each aspect of the system.

One of the major difficulties with system documentation is the maintenance of consistency across the different documents describing the system. To keep track of changes, it is recommended that documents should be placed under the control of a configuration management system.

Components of the system documentation:

- 1) The requirements definition and specification and an associated rationale.
- 2) An overall system specification showing how the requirements are decomposed into a set of interacting programs. This document is not required when the system is implemented using only a single program.
- 3) For each program in the system, a description of how that program is decomposed into components and a statement of the specification of each component.
- 4) For each unit, a description of its operation. This need not extend to describing program actions as these should be documented using intra-program comments.
- 5) A comprehensive test plan describing how each program unit is tested.
- 6) A test plan showing how integration testing, that is, the testing of all units/programs together is carried out.
- 7) An acceptance test plan, devised in conjunction with the system user. This should describe the tests which must be satisfied before the system is accepted.

6.3 Document quality

Document quality is as important as program quality.

Without information on how to use a system or how to understand it, the utility of that system is degraded.

Producing good documents is neither easy nor cheap and the process is at least as difficult as producing good programs.

Documentation standards should describe exactly what the documentation should include and should describe the notations to be used in the documentation.

Within an organization, it is useful to establish a standard for documents and require that all documents conform to that format.

The standard for documents might include:

- a description of a front-cover format to be adopted by all documents,
- page numbering and page annotation conventions,
- methods of reference to other documents, and
- the numbering of headings and sub-headings.

Writing style

The most fundamental factor affecting documentation quality is the ability of the writer to construct clear and concise technical prose.

Good documentation requires good writing.

Writing documents well is neither easy nor is it a single-stage process. Written work must be written, read, criticized and then rewritten, and this process should continue until a satisfactory document is produced.

It is impossible to present a set of rules which govern exactly how to set about this particular task. Technical writing is a craft rather than a science and only broad guidelines about how to write well may be given.

Guidelines for writing software instruction manuals: ([Sommerville 1996])

- 1) *Use active rather than passive tenses.* It is better to say ‘You should see a flashing cursor at the top left of the screen’ rather than ‘A flashing cursor should appear at the top left of the screen’.
- 2) *Do not use long sentences which present several different facts.* It is better to use a number of shorter sentences. Each sentence can then be assimilated on its own. The reader does not need to maintain several pieces of information at one time in order to understand the complete sentence.
- 3) *Do not refer to information by reference number alone.* Instead, give the reference number and remind the reader what that reference covered. For example, rather than say ‘In section 1.3...’ you should say ‘In section 1.3, which described software evolution’.
- 4) *Itemize facts wherever possible.* It is usually clearer to present facts in a list rather than in a sentence. You might have found this set of guidelines harder to read if they hadn’t been itemized. Use textual highlighting (*Italics* or underlining) for emphasis.
- 5) *If a description is complex, repeat yourself.* It is often a good idea to present two or more differently phrased descriptions of the same thing. If the reader fails to completely understand one description, he or she may benefit from having the same thing said in a different way.
- 6) *Don’t be verbose.* If you can say something in five words do so, rather than use ten words so that the description might seem more profound. There is no merit in quantity of documentation. Quality is more important.
- 7) *Be precise and define the terms you use.* Computing terminology is very fluid and many terms have more than one meaning. Therefore, if specialized terms (such as module or process) are used, make sure that your definition is clear. If you need to define a number of words or if you are writing for readers with little or no knowledge of computing terminology, you should provide a glossary with your document. This should contain definitions of all terms that might not be completely understood by the reader.
- 8) *Keep paragraphs short.* As a general rule, no paragraph should be made up of more than seven sentences. This is because of short-term memory limitations. Our capacity for holding immediate information is limited, so by keeping paragraphs short all of the concepts in the paragraph can be maintained in short-term memory.

- 9) *Make use of headings and sub-headings.* These break up a chapter into parts which may be read separately. Always ensure that a consistent numbering convention is used for these.
- 10) *Use grammatically correct constructs and correct spelling.* To boldly go on splitting infinitives (like this) and to misspell words (like mispell) irritates many readers and reduces the credibility of the writer in their eyes.

Documentation tools

There is a variety of software tools available to help with the production of documents. In general, documentation should be produced and stored on the same computer as is used to develop the system software.

The most important documentation tool is a powerful editing system which allows documents to be generated and modified.

A general-purpose text editor may be used for this task or a word processing system may be preferred.

Guidelines for selecting documentation tools:

- 1) *The documentation is always on hand.* Software developers or users need not search for a manual if they have access to the computer. In the case of user documentation, copies of this should be maintained on the same system as the application so that, again, reference to manuals may be unnecessary. The need for effective indexing of documents is particularly important when the documentation is maintained on-line. There is no point in having the information available if the reader cannot easily and quickly find the information required.
- 2) *Documents are easy to modify and maintain* This has the consequence that the project documentation is more likely to be kept up to date.
- 3) *Documents may be automatically analysed* The text can be analysed in various ways to produce different types of index and to check for spelling and typing errors.
- 4) *Document production may be shared.* Several individuals may work on the production of a document at the same time.
- 5) *Documentation management is simplified.* All associated documents can be collected under a common heading. The state of development of these documents may be ascertained. Configuration management tools may be used to maintain different versions of documents and to control changes.
- 6) *Automatic information retrieval is feasible.* Information retrieval systems may operate on documents retrieving all documents which contain particular keywords, for example.

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.

Self-containedness is the key to

- program portability, and
- documentation portability.

Documentation portability means that the information provided in the documentation should be as complete as possible. Reference should not be made to any other documents, such as an operating system manual, which are not directly associated with the system being documented.

Those parts of the system which generally cause portability problems are obviously the sections describing non-portable parts of the programming system. These include file organization, file naming conventions, job control, input/output, and so on. When the system is moved from one computer to another, those parts of the documentation must be rewritten.

To make this rewriting easier, descriptions of system dependent functions should be confined to separate sections. These sections should be clearly headed with information about their system dependence and they are replaced when the system is moved to another installation. If possible, other references to system dependent features should be avoided. If this is impossible, an index of such references should be maintained so that they may be located and changed when the program is moved.

As transporting a program and its documentation is really a specialized form of system maintenance, the availability of machine readable documentation and appropriate software tools reduces the work involved in producing documentation for a new system. System dependent parts of the document may be located, rewritten and a new version of the document produced automatically.

Objectives

This chapter is concerned with the goal and tasks of project management, aspects of cost estimation, project organization and software maintenance. A classical hierarchical organizational model is discussed. Then an alternative organizational model, and the chief programmer team are presented. Types of maintenance, maintenance costs and cost estimation are described in this chapter. A relatively new approach to maintenance, namely program restructuring, is discussed.

Contents

- 7.1 The goal and tasks of project management
 - 7.2 Difficulties in project management
 - 7.3 Cost estimation
 - 7.4 Project organization
 - 7.4.1 Hierarchical organizational model
 - 7.4.2 The chief programmer team
 - 7.5 Software maintenance
 - 7.5.1 Maintenance costs
 - 7.5.2 System restructuring
 - 7.5.3 Program evolution dynamics
- References and selected reading
-

[Balzert 1985], [Pressman 1987], [Friihauf 1988] and [Denert 1991]

- The production of complex software products generally involves a group of people with different tasks and varying expertise.
- To attain the goal of *software engineering - the economic production* of software products - the organizational measures for on-schedule and economic production of software products need to be just as well thought out as the technical measures for product development.

- Frequently software products are completed later than the target release date and with significantly higher costs than estimated at the start of the project, or software projects fail completely. The causes often reside at the organizational level.

Important reasons:

- *The lack of project planning, project organization and project standards.*

Due to their complexity, software projects need to be planned and organized especially carefully. Programmers are individualists who see themselves as craftspeople and do not like to put all their cards on the table. This proves to be a disturbing factor in software development involving a division of labor. Furthermore, the lack of a systematically created project plan, no specification or enforcement of an approach, and missing project standards and milestones for project progress checks can have devastating effects on large software projects because shortcomings in the quality of individual system components and schedule delays are detected much too late.

- *Incompetent project leadership.*

The project leadership is usually organized strictly hierarchically, and the manager is often a software engineer with little experience in organization and planning, or a qualified manager with little or no experience in the area of software engineering.

- *Lack of up-to-date documentation.*

Documentation is very frequently neglected in the course a project. This impedes communication among people involved in the project and prevents project progress control.

- *Lack of project progress and quality control.*

More complex software systems consist of subsystems which consist of components that are generally implemented by different people. Quality shortcomings in individual components usually require expensive corrections that lead to cost overruns and missed deadlines. Neglecting project progress and quality control can lead to the interruption of further project progress due to the delay in the completion of individual system components; this in turn can overturn the overall project planning.

- *Lack of cost control.*

Without running control of the costs of a project, it is impossible to assure early detection of errors in cost assessments and of bottlenecks in the course of the project and to take appropriate measures.

7.1 The goal and tasks of project management

- The *goal of project management* is to ensure that a software product that meets the quality specifications of the requirements definition is produced on schedule and at

an economically justifiable cost within the planned resources (e.g. personnel, equipment, tools).

- The primary tasks of project management are *planning, organization and technical and economic control* of project execution.

Planning

- The *goal of planning* is to establish deadlines, costs, personnel requirements, and measures for quality assurance of the software product to be developed. This necessitates organizing software projects so that the individual tasks are largely independent of one another and so that these tasks can be carried out with testable results in teams as small as possible, with economically justifiable investment of time and technical resources.
- Exact project planning, appropriate to the size of the project, is of deciding importance for the successful execution of a software project.
- The scope of the planning and the effort invested in it depend on the following:
 - *Task description*
A project with a short, clear requirements definition and corresponding staff experience in similar projects is significantly easier to plan than a project with a vague task description that can only be clarified with prototyping and that has no corresponding models.
 - *Project size*
The difficulty of planning rises with an increasing staff and other resources involved (e.g., development machines, interfaces).
 - *Experience and qualifications of the project team*
The qualifications of the project staff members, their experience, their degree of knowledge in the application domain, and their ability to cooperate are important planning parameters.
 - *Constraints for the project*
 - *Development methods used*
- *The result of the planning:*
 - Organizational plan and personnel schedule
 - Timetable for each phase of the software project
 - Cost schedule
 - Documentation schedule
 - Testing schedule and planning of measures for quality assurance for all intermediate products and for the final product

Organization

[Baker 1979]

- Software products differ from other technical products in that they are intellectual products whose design and construction cannot be as easily distinguished as those in other technical products.
- Conventional organizational forms do not apply directly to the development of software products. Therefore we lack uniform organizational models for software development.
- Most important *organizational tasks* of project management:
 - Assignment of teams or individuals to phases (or subphases) of the software development process
 - Composition of teams (members, form of organization, leadership)
 - Exact specification of tasks, rights and duties of all persons involved in the project
 - Establishment of project standards, methods to be used (prototyping, object-oriented design) and measures to provide software tools
 - Regulation of communication (setting meetings, documentation guidelines, etc.)
 - Providing resources (computers, office materials, money and rooms)

Technical monitoring

- Even careful and extensive planning is nearly worthless without continuous control of the achievement of the goals prescribed in the planning.
- The goal of *technical monitoring* is to check both *quantitatively and qualitatively the technical attributes* of the intermediate products of the individual phases.
- *Quantitative* monitoring checks
 - Whether the development goals are being met according to the timetable, and
 - Whether an intermediate product fulfills its functionality as specified in the requirements definition and its quality requirement.
- *Technical monitoring* encompasses not only checking whether certain technical goals have been achieved, but also triggering corrective measures if a goal has not been achieved.

Economic monitoring

- *Economic monitoring* has *quantitative and qualitative* aspects.
 - Quantitative monitoring must check for cost overruns.
 - Qualitative monitoring determines whether the completed intermediate products fulfill the contract conditions.

- *Cost control* is only possible if corresponding cost accounting is carried out. Project management must therefore assure that all incurred costs are recorded and documented.
- Monitoring the economy of a software project is normally more difficult than technical monitoring. While developers normally succeed in completing a planned program at some time with adequate technical quality and in putting it into operation, very seldom do the schedule and budget estimates fall anywhere near reality.
- *Schedule and cost estimates* for software products are difficult tasks.

7.2 Difficulties in project management

- *Uniqueness of software systems*

Software systems are developed only once. Data is too scarce to allow for reliable cost estimates. Cost estimates are usually based on the scope of the software system to be developed, as estimated by an experienced software engineer (as a multiple of the number of program statements). Generally this provides an unreliable basis for calculations.

- *Multitude of possible solutions*

There are practically an unlimited number of possibilities for solving a given problem. The development of software products is not subject to the strict constraints that apply for other technical products (e.g., standards and material attributes). For software products, the limits stem from complexity and thus are difficult to determine in advance.

- *Individuality of programmers*

Even with the state of the art in software engineering, software development remains more of a craft than a science. Software engineers are individualists with large differences in capability; this makes it particularly difficult to estimate personnel costs. In addition, individualists usually resist being forced into an organizational corset.

- *Rapidity of technological change*

The speed of technological development in hardware and software encumbers both planning and organization of software projects. Sometimes during the development of a large software system, new, more powerful components (e.g., graphic monitors and window management systems) appear on the market that render the conception of the system obsolete after its design and force redesign. This can completely overturn existing planning. The project might also employ new software tools whose benefits are difficult to assess in advance.

- *The immaterial nature of software products*

The invisibility of software products encumbers their control. It is difficult to assess how much of a software product is actually finished, and the programmer has many

possibilities for disguising the actual state of development. Control tasks are more difficult to carry out than with other technical projects.

7.3 Cost estimation

[Boehm 1981], [Putnam 1978], [Albrecht 1983], [Schnupp 1976]

- The necessity of cost estimation stems from the requirements of scheduling and cost planning. For lack of more precise methods, cost estimation for software development is almost always based on a comparison of the current project with previous ones. Due to the uniqueness of software systems, the number of comparable projects is usually quite small, and empirical data are seldom available. But even if there are exact cost records of comparable projects, these data are based on the technical and organizational conditions under which the comparable project was carried out at that time.
- The technical and organizational conditions are variable parameters, which makes empirical data from comparable projects only an unreliable basis for estimates.

Cost estimation models

- *COCOMO Model,*
- *Putnam Estimation Model*
- *Function Point Model.*

Factors for cost and time estimates

- Experience and qualifications of the estimator
 - Type of software (e.g., function-, data- or object-oriented, time-critical software with high efficiency requirements, control and monitoring software with high quality requirements)
 - Anticipated complexity (e.g., number of components (modules, classes), number of component couplings, number of methods per class, complexity of methods)
 - Expected size of program (number of statements)
 - Experience and qualifications of personnel (e.g., project experience, programming experience, knowledge of methods, experience in use of tools, knowledge of problem domain)
 - Techniques and tools to be used (e.g., planning, design and documentation techniques, test strategies and tools, programming languages)
 - Number of staff members
- Relationship between the best and worst programming experience (referring to the same task, [Schnupp 1976]):

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 one 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%

Approach model: prototyping-oriented software life cycle

Implementation technique: module-oriented

problem analysis and system specification.....	40%
design.....	25%
implementation.....	10%
testing.....	25%

Approach model: object- and prototyping-oriented software life cycle

Implementation technique: object-oriented

problem analysis and system specification.....	45%
design.....	20%
implementation.....	8%
testing.....	27%

(These values can only be used as rough guidelines.)

7.4 Project organization

7.4.1 Hierarchical organizational model

- There are many ways to organize the staff of a project. For a long time the organization of software projects oriented itself to the hierarchical organization common to other industrial branches. Special importance is vested in the decomposition of software development into individual phases. A responsible leader is assigned to each of the phases, which are led and controlled by the project leader and which, depending on the size of the project, are led and controlled either by a single person or by a group leader.
- The project manager normally also has a project management staff with advisory and administrative tasks.
- The larger the project, the greater is the number of hierarchy levels in the organizational schema.
 - The project manager's tasks and responsibilities encompass
 - personnel selection,
 - assignment and management,
 - planning of and division of labor for the project,
 - project progress checks, and
 - appropriate measures in case of cost or schedule overruns.

- *The project management staff* includes personnel who advise the project manager in task-specific questions, provide support in administrative tasks concerning project progress checks, prepare project standards, provide the necessary resources, and carry out training for project team members as needed.
- *The managers at the middle management level* are responsible for planning, execution and control of phase-related activities of the software life cycle.

7.4.2 The chief programmer team

Baker's organizational model ([Baker 1972])

- Important characteristics:
 - The lack of a project manager who is not personally involved in system development
 - The use of very good specialists
 - The restriction of team size
- The chief programmer team consists of:
 - The chief programmer
 - The project assistant
 - The project secretary
 - Specialists (language specialists, programmers, test specialists).
- The chief programmer is actively involved in the planning, specification and design process and, ideally, in the implementation process as well.
- The chief programmer controls project progress, decides all important questions, and assumes overall responsibility.
- The qualifications of the chief programmer need to be accordingly high.
- The project assistant is the closest technical coworker of the chief programmer.
- The project assistant supports the chief programmer in all important activities and serves as the chief programmer's representative in the latter's absence. This team member's qualifications need to be as high as those of the chief programmer.
- The project secretary relieves the chief programmer and all other programmers of administrative tasks.
- The project secretary administrates all programs and documents and assists in project progress checks.
- The main task of the project secretary is the administration of the project library.
- The chief programmer determines the number of specialists needed.
- Specialists select the implementation language, implement individual system components, choose and employ software tools, and carry out tests.
- The advantages of the Baker's organizational model:

- The chief programmer is directly involved in system development and can better exercise the control function.
 - Communication difficulties of pure hierarchical organization are ameliorated. Reporting concerning project progress is institutionalized.
 - Small teams are generally more productive than large teams.
- The drawbacks of the Baker's organizational model:
- It is limited to small teams. Not every project can be handled by a small team.
 - Personnel requirements can hardly be met. Few software engineers can meet the qualifications of a chief programmer or a project assistant.
 - The project secretary has an extremely difficult and responsible job, although it consists primarily of routine tasks, which gives it a subordinate position. This has significant psychological disadvantages. Due to the central position, the project secretary can easily become a bottleneck.
 - The organizational model provides no replacement for the project secretary. The loss of the project secretary would have grave consequences for the remaining course of the project.

7.5 Software maintenance

- *Software maintenance* is a process of modifying a program after it has been delivered and is in use.
- The modifications may involve:
 - Simple changes to correct coding errors,
 - More extensive changes to correct design errors, or
 - Drastic rewrites to correct specification errors or accommodate new requirements ([Turski 1981]).
- Software maintenance is concerned with planning and predicting the process of change.
- It is impossible to produce systems of any size which do not need to be maintained. Over the lifetime of a system, its original requirements will be modified to reflect changing needs, the system's environment will change, and obscure errors, undiscovered during system validation, will emerge.
- Because maintenance is unavoidable, systems should be designed and implemented so that maintenance problems are *minimized*.

Three categories of software maintenance

- Perfective maintenance (about 65% of maintenance).
- Adaptive maintenance (about 18% of maintenance).
- Corrective maintenance (about 17% of maintenance).

- *Perfective maintenance* means those changes demanded by the user or the system programmer which improve the system in some way without changing its functionality.
- *Adaptive maintenance* is maintenance due to changes in the environment of the program.
- *Corrective maintenance* is the correction of undiscovered system errors.
- The techniques involved in maintaining a software system are essentially those used in building the system in the first place.
 - New requirements must be formulated and validated,
 - Components of the system must be redesigned and implemented
 - Part or all of the system must be tested.

The techniques used in these activities are the same as those used during development.

- There are no special technical tricks which should be applied to software maintenance.

Important

- Large organizations devoted about 50% of their total programming effort to maintaining existing systems.
- The program maintainer pays attention to the *principles of information hiding*. For a long-lived system, it is quite possible that a set of changes to a system may themselves have to be maintained. It is a characteristic of any change that the original program structure is corrupted. The greater the corruption, the less understandable the program becomes and the more difficult it is to change. The program modifier should try, as far as possible, to minimize effects on the program structure.
- One of the problems of managing maintenance is that maintenance has a poor image among software engineers. It is seen as a less skilled process than program development and, in many organizations, maintenance is allocated to inexperienced staff. The end result of this negative image is that maintenance costs are probably increased because staff allocated to the task are less skilled and experienced than those involved in system design.

Five steps for improving the motivation of maintenance staff

([Boehm 1983])

- 1) Couple software objectives to organizational goals.
- 2) Couple software maintenance rewards to organizational performance.
- 3) Integrate software maintenance personnel into operational teams.
- 4) Create a discretionary perfective maintenance budget.

- 5) Involve maintenance staff early in the software process during standards preparation reviews and test preparation.

7.5.1 Maintenance costs

- *Maintenance costs* are the *greatest cost* incurred in developing and using a system. In general, these costs were dramatically underestimated when the system was designed and implemented.

It was estimated that one US Air Force System cost \$30 per instruction to develop and \$4000 per instruction to maintain over its lifetime ([Boehm 1975]).

- Maintenance costs certainly vary widely from application to application but, on average, they seem to be between two and four times development costs for large embedded software systems.
- As systems age, relatively more effort must be expended in maintaining those systems.

One reason for this is that these systems may be written in obsolete programming languages which are no longer used for new systems development.

- It is obviously worthwhile to invest time and effort when designing and implementing a system to reduce maintenance and hence overall system costs.

Extra development effort is a negative multiplier on maintenance costs. A percentage increase in development costs, if it leads to a comparable percentage decrease in maintenance costs, results in an overall saving.

Maintenance cost factors

Non-technical factors

- *Application domain*
- *Staff stability*
- *Program age*
- *External environment*
- *Hardware stability*

- 1) *The application being supported.*

If the application of the program is *clearly defined* and *well understood*, the system requirements may be definitive and maintenance due to changing requirements minimized.

If the application is completely new, it is likely that the initial requirements will be modified frequently, as users gain experience with the system.

- 2) *Staff stability.* It is easier for the original writer of a program to understand and change a program rather than some other individual who must understand the program by study of its documentation and code listing.

If the programmer of a system also maintains that system, maintenance costs will be reduced.

In practice, the nature of the programming profession is such that individuals change jobs regularly. It is unusual for one person to develop and maintain a program throughout its useful life.

- 3) *The lifetime of the program.*

The useful life of a program depends on its application.

Programs become obsolete when the application becomes obsolete or their original hardware is replaced and conversion costs exceed rewriting costs.

The older a program, the more it has been maintained and the more degraded its structure.

Maintenance costs tend to rise with program age.

- 4) *The dependence of the program on its external environment.*

If a program is dependent on its external environment it must be modified as that environment changes. For example, changes in a taxation system might require payroll, accounting, and stock control programs to be modified. Taxation changes are relatively common and maintenance costs for these programs are related to the frequency of these changes.

A program used in a mathematical application does not normally depend on humans changing the assumptions on which the program is based.

- 5) *Hardware stability.*

If a program is designed to operate on a particular hardware configuration and that configuration does not change during the program's lifetime, no maintenance costs due to hardware changes will be incurred. However, hardware developments are so rapid that this situation is rare. The program must be modified to use new hardware which replaces obsolete equipment.

Technical factors

- *Module independence*
- *Programming language*
- *Programming style*
- *Program validation*
- *Documentation*

- 1) *Module independence.* It should be possible to modify one program unit of a system without affecting any other unit.

- 2) *Programming language.* Programs written in a high-level programming language are usually easier to understand (and hence maintain) than programs written in a low-level language.
- 3) *Programming style.* The way in which a program is written contributes to its understandability and hence the ease with which it can be modified.
- 4) *Program validation and testing.*

Generally, the more time and effort spent on design validation and program testing, the fewer errors in the program and, consequently, maintenance costs resulting from error correction are lower.

Maintenance costs due to error correction are governed by the type of error to be repaired.

Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve the rewriting of one or more program units.

Errors in the software requirements are usually the most expensive to correct because of the drastic redesign which is usually involved.

- 5) *The quality and quantity of program documentation.*

If a program is supported by clear, complete yet concise documentation, the task of understanding the program can be relatively straightforward.

Program maintenance costs tend to be less for well documented systems than for systems supplied with poor or incomplete documentation.

Maintenance cost estimation

Boehm's maintenance cost estimation

- *Boehm's maintenance cost estimation* ([Boehm 1981]) is calculated in terms of a quantity called the *Annual Change Traffic* (ACT) which is defined as follows:
The fraction of a software product's source instructions which undergo change during a (typical) year either through addition or modification.

$$\text{AME} = 1.0 * \text{ACT} * \text{SDT}$$

where

AME is the *annual maintenance effort* (person-months),

SDT is the *software development time* (person-months).

Example 7.1 Say a software project required 236 person-months of development effort and it was estimated that 15% of the code would be modified in a typical year. The basic maintenance effort estimate is:

$$\text{AME} = 1.0 * 0.15 * 236 = 35.4 \text{ person-months.}$$

- 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:

$$\text{AME} = 35.4 * 1.10 * 0.91 * 0.95 * 0.72 = 24.2 \text{ person-months.}$$

- 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:

$$\text{AME} = 35.4 * 1.10 * 0.91 * 0.95 * 1.40 = 47.1 \text{ person-months.}$$

- 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:

$$AME = 35.24 * 1.10 * 1.07 * 1.13 * 0.72 = 33.89 \text{ person-months.}$$

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.

- The program complexity is not dependent on size but on the decision structure of the program. Measurement of the complexity of a program depends on transforming the program so that it may be represented as a graph and counting the number of nodes, edges and connected components in that graph ([McCabe 1976]).

Other metrics

- [Shepherd 1979]
- [Hamer 1981].
- [Kafura 1987]

7.5.2 System restructuring

- Restructuring involves examining the existing system and rewriting parts of it to improve its overall structure.
- Restructuring may be particularly useful when changes are confined to part of the system. Only this part need be restructured. Other parts need not be changed or revalidated.
- If a program is written in a high-level language, it is possible to restructure that program automatically although the computer time required to do so may be great.
- **Theorem** (The basis for program restructuring [Bohm 1966]) *Any program may be rewritten in terms of simple IF-THEN-ELSE conditionals and WHILE loops and that unconditional GOTO statements were not required.*
- **Method**
 - Step 1. *Construct a program flow graph.*
 - Step 2. *Apply simplification and transformation techniques to the graph to construct while loops and simple conditional statements.*
- It may well be that a combination of automatic and manual system restructuring is the best approach. The control structure could be improved automatically and this makes the system easier to understand. The abstraction and data structures of the program may then be discovered, documented and improved using a manual approach ([Britcher 1986]).
- Decisions on whether to restructure or rewrite a program can only be made on a case-by-case basis. Some of the factors which must be taken into account are ([Sommerville 1996]):
 - 1) Is a significant proportion of the system stable and not subject to frequent change? If so, this suggests restructuring rather than rewriting as it is only really necessary to restructure that part of the program which is to be changed.
 - 2) Does the program rely on obsolete support software such as compilers, etc.? If so, this suggests it should be rewritten in a modern language as the future availability of the support software cannot be guaranteed.

- 3) Are tools available to support the restructuring process? If not, manual restructuring is the only option.
- System restructuring offers an opportunity to control maintenance costs and I believe that it will become increasingly important. The rate of change of hardware development means that many embedded software systems which are still in use must be changed as the hardware on which they execute cannot be supported.

7.5.3 Program evolution dynamics

([Lehman 1985])

Lehman's laws

1. **The law of continuing change:** *A program that is used in a real-world environment necessarily must change or become less and less useful in that environment.*
2. **The law of increasing complexity:** *As an evolving program changes, its structure becomes more complex unless active efforts are made to avoid this phenomenon.*
3. **The law of large program evolution:** *Program evolution is a self-regulating process and measurement of system attributes such as size, time between releases, number of reported errors, etc., reveals statistically significant trends and invariances.*
4. **The law of organizational stability:** *Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted to system development.*
5. **The law of conservation of familiarity:** *Over the lifetime of a system, the incremental system change in each release is approximately constant.*

- The first law tells us that system maintenance is an inevitable process. Fault repair is only part of the maintenance activity and that changing system requirements will always mean that a system must be changed if it is to remain useful. Thus, the constant theme of this text is that software engineering should be concerned with producing systems whose structure is such that the costs of change are minimized.
- The second law states that, as a system is changed, its structure is degraded and additional costs, over and above those of simply implementing the change, must be accepted if the structural degradation is to be reversed. The maintenance process should perhaps include explicit restructuring activities which are simply aimed at improving the adaptability of the system. It suggests that program restructuring is an appropriate process to apply.
- The third law suggests that large systems have a dynamic all of their own and that is established at an early stage in the development process. This dynamic determines the gross trends of the system maintenance process and the particular

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.

Effective Software Testing

50 Specific
Ways to

Improve
Your

Testing



For the EPUBCN Group

www.epubcn.com

EPUBCN EX!X!

Elfriede Dustin

Copyright

Preface

Organization

Audience

Acknowledgments

Chapter 1. Requirements Phase

Item 1: Involve Testers from the Beginning

Item 2: Verify the Requirements

Item 3: Design Test Procedures As Soon As Requirements Are Available

Item 4: Ensure That Requirement Changes Are Communicated

Item 5: Beware of Developing and Testing Based on an Existing System

Chapter 2. Test Planning

Item 6: Understand the Task At Hand and the Related Testing Goal

Item 7: Consider the Risks

Item 8: Base Testing Efforts on a Prioritized Feature Schedule

Item 9: Keep Software Issues in Mind

Item 10: Acquire Effective Test Data

Item 11: Plan the Test Environment

Item 12: Estimate Test Preparation and Execution Time

Chapter 3. The Testing Team

Item 13: Define Roles and Responsibilities

Item 14: Require a Mixture of Testing Skills, Subject-Matter Expertise, and Experience

Item 15: Evaluate the Tester's Effectiveness

Chapter 4. The System Architecture

Item 16: Understand the Architecture and Underlying Components

Item 17: Verify That the System Supports Testability

Item 18: Use Logging to Increase System Testability

Item 19: Verify That the System Supports Debug and Release Execution Modes

Chapter 5. Test Design and Documentation

Item 20: Divide and Conquer

Item 21: Mandate the Use of a Test-Procedure Template and Other Test-Design Standards

Item 22: Derive Effective Test Cases from Requirements

Item 23: Treat Test Procedures As "Living" Documents

Item 24: Utilize System Design and Prototypes

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

Item 26: Avoid Including Constraints and Detailed Data Elements within Test Procedures

Item 27: Apply Exploratory Testing

Chapter 6. Unit Testing

Item 28: Structure the Development Approach to Support Effective Unit Testing

Item 29: Develop Unit Tests in Parallel or Before the Implementation

Item 30: Make Unit-Test Execution Part of the Build Process

Chapter 7. Automated Testing Tools

Item 31: Know the Different Types of Testing-Support Tools

Item 32: Consider Building a Tool Instead of Buying One

Item 33: Know the Impact of Automated Tools on the Testing Effort

Item 34: Focus on the Needs of Your Organization

Item 35: Test the Tools on an Application Prototype

Chapter 8. Automated Testing: Selected Best Practices

Item 36: Do Not Rely Solely on Capture/Playback

Item 37: Develop a Test Harness When Necessary

Item 38: Use Proven Test-Script Development Techniques

Item 39: Automate Regression Tests When Feasible

Item 40: Implement Automated Builds and Smoke Tests

Chapter 9. Nonfunctional Testing

Item 41: Do Not Make Nonfunctional Testing an Afterthought

Item 42: Conduct Performance Testing with Production-Sized Databases

Item 43: Tailor Usability Tests to the Intended Audience

Item 44: Consider All Aspects of Security, for Specific Requirements and System-Wide

Item 45: Investigate the System's Implementation To Plan for Concurrency Tests

Item 46: Set Up an Efficient Environment for Compatibility Testing

Chapter 10. Managing Test Execution

Item 47: Clearly Define the Beginning and End of the Test-Execution Cycle

Item 48: Isolate the Test Environment from the Development Environment

Item 49: Implement a Defect-Tracking Life Cycle

Item 50: Track the Execution of the Testing Program

Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales

(317) 581-3793

international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Dustin, Elfriede.

Effective software testing : 50 specific ways to improve your testing / Elfriede Dustin.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-79429-2

1. Computer software--Testing. I. Title.

QA76.76.T48 D873 2002

005.1'4—dc21

2002014338

Copyright © 2003 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.

Rights and Contracts Department

75 Arlington Street, Suite 300

Boston, MA 02116

Fax: (617) 848-7047

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—MA—0605040302

First printing, December 2002

Dedication

To Jackie, Erika, and Cedric

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

pseudocode is given where necessary, the content is not tied to any particular technology or application platform.

It is important to note that there are factors outside the scope of the testing program that bear heavily on the success or failure of a project. Although a complete software-development process with its attendant testing program will ensure a successful engineering effort, any project must also deal with issues relating to the business case, budgets, schedules, and the culture of the organization. In some cases, these issues will be at odds with the needs of an effective engineering environment. The recommendations in this book assume that the organization is capable of adapting, and providing the support to the testing program necessary for its success.

Organization

This book is organized into 50 separate items covering ten important areas. The selected best practices are organized in a sequence that parallels the phases of the system development life cycle.

The reader can approach the material sequentially, item-by-item and part-by-part, or simply refer to specific items when necessary to gain information about and understanding of a particular problem. For the most part, each chapter stands on its own, although there are references to other chapters, and other books, where helpful to provide the reader with additional information.

Chapter 1 describes requirements-phase considerations for the testing effort. It is important in the requirements phase for all stakeholders, including a representative of the testing team, to be involved in and informed of all requirements and changes. In addition, basing test cases on requirements is an essential concept for any large project. The importance of having the testing team represented during this phase cannot be overstated; it is in this phase that a thorough understanding of the system and its requirements can be obtained.

Chapter 2 covers test-planning activities, including ways to gain understanding of the goals of the testing effort, approaches to determining the test strategy, and considerations related to data, environments, and the software itself. Planning must take place as early as possible in the software life cycle, as lead times must be considered for implementing the test program successfully. Early planning allows for testing schedules and budgets to be estimated, approved, and incorporated into

the overall software development plan. Estimates must be continually monitored and compared to actuals, so they can be revised and expectations can be managed as required.

Chapter 3 focuses on the makeup of the testing team. At the core of any successful testing program are its people. A successful testing team has a mixture of technical and domain knowledge, as well as a structured and concise division of roles and responsibilities. Continually evaluating the effectiveness of each test-team member throughout the testing process is important to ensuring success.

Chapter 4 discusses architectural considerations for the system under test. Often overlooked, these factors must be taken into account to ensure that the system itself is testable, and to enable gray-box testing and effective defect diagnosis.

Chapter 5 details the effective design and development of test procedures, including considerations for the creation and documentation of tests, and discusses the most effective testing techniques. As requirements and system design are refined over time and through system-development iterations, so must the test procedures be refined to incorporate the new or modified requirements and system functions.

Chapter 6 examines the role of developer unit testing in the overall testing strategy. Unit testing in the implementation phase can result in significant gains in software quality. If unit testing is done properly, later testing phases will be more successful. There is a difference, however, between casual, ad-hoc unit testing based on knowledge of the problem, and structured, repeatable unit testing based on the requirements of the system.

Chapter 7 explains automated testing tool issues, including the proper types of tools to use on a project, the build-versus-buy decision, and factors to consider in selecting the right tool for the organization. The numerous types of testing tools available for use throughout the phases in the development life cycle are described here. In addition, custom tool development is also covered.

Chapter 8 discusses selected best practices for automated testing. The proper use of capture/playback tools, test harnesses, and regression testing are described.

Chapter 9 provides information on testing nonfunctional aspects of a software application. Ensuring that nonfunctional requirements are met, including performance, security, usability, compatibility, and concurrency testing, adds to the overall quality of the application.

[Chapter 10](#) provides a strategy for managing the execution of tests, including appropriate methods of tracking test-procedure execution and the defect life cycle, and gathering metrics to assess the testing process.

Audience

The target audience of this book includes Quality Assurance professionals, software testers, and test leads and managers. Much of the information presented can also be of value to project managers and software developers looking to improve the quality of a software project.

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 1: Involve Testers from the Beginning

Testers need to be involved from the beginning of a project's life cycle so they can understand exactly what they are testing and can work with other stakeholders to create testable requirements.

Defect prevention is the use of techniques and processes that can help detect and avoid errors before they propagate to later development phases. Defect prevention is most effective during the requirements phase, when the impact of a change required to fix a defect is low: The only modifications will be to requirements documentation and possibly to the testing plan, also being developed during this phase. If testers (along with other stakeholders) are involved from the beginning of the development life cycle, they can help recognize omissions, discrepancies, ambiguities, and other problems that may affect the project requirements' testability, correctness, and other qualities.

A requirement can be considered **testable** if it is possible to design a procedure in which the functionality being tested can be executed, the expected output is known, and the output can be programmatically or visually verified.

Testers need a solid understanding of the product so they can devise better and more complete test plans, designs, procedures, and cases. Early test-team involvement can eliminate confusion about functional behavior later in the project life cycle. In addition, early involvement allows the test team to learn over time which aspects of the application are the most critical to the end user and which are the highest-risk elements. This knowledge enables testers to focus on the most important parts of the application first, avoiding over-testing rarely used areas and under-testing the more important ones.

Some organizations regard testers strictly as consumers of the requirements and other software development work products, requiring them to learn the application and domain as software builds are delivered to the testers, instead of involving them during the earlier phases. This may be acceptable in smaller projects, but in complex environments it is not realistic to expect testers to find all significant defects if their first exposure to the application is after it has already been through requirements, analysis, design, and some software implementation. More than just understanding the "inputs and outputs" of the software, testers need deeper knowledge that can come only from understanding the *thought process* used during the specification of product functionality. Such understanding not only increases

the quality and depth of the test procedures developed, but also allows testers to provide feedback regarding the requirements.

The earlier in the life cycle a defect is discovered, the cheaper it will be to fix it. [Table 1.1](#) outlines the relative cost to correct a defect depending on the life-cycle stage in which it is discovered.^[1]

^[1] B. Littlewood, ed., *Software Reliability: Achievement and Assessment* (Henley-on-Thames, England: Alfred Waller, Ltd., November 1987).

Table 1.1. Prevention is Cheaper Than Cure: Error Removal Cost Multiplies over System Development Life Cycle

Phase	Relative Cost to Correct
Definition	\$1
High-Level Design	\$2
Low-Level Design	\$5
Code	\$10
Unit Test	\$15
Integration Test	\$22
System Test	\$50
Post-Delivery	\$100+

Item 2: Verify the Requirements

In his work on specifying the requirements for buildings, Christopher Alexander^[1] describes setting up a **quality measure** for each requirement: "The idea is for each requirement to have a quality measure that makes it possible to divide all solutions to the requirement into two classes: those for which we agree that they fit the requirement and those for which we agree that they do not fit the requirement." In other words, if a quality measure is specified for a requirement, any solution that meets this measure will be acceptable, and any solution that does not meet the measure will not be acceptable. Quality measures are used to test the new system against the requirements.

^[1] Christopher Alexander, *Notes On the Synthesis of Form* (Cambridge, Mass.: Harvard University Press, 1964).

Attempting to define the quality measure for a requirement helps to rationalize fuzzy requirements. For example, everyone would agree with a statement like "the system must provide good value," but each person may have a different interpretation of "good value." In devising the scale that must be used to measure "good value," it will become necessary to identify what that term means. Sometimes requiring the stakeholders to think about a requirement in this way will lead to defining an agreed-upon quality measure. In other cases, there may be no agreement on a quality measure. One solution would be to replace one vague requirement with several unambiguous requirements, each with its own quality measure.^[2]

^[2] Tom Gilb has developed a notation, called Planguage (for Planning Language), to specify such quality requirements. His forthcoming book *Competitive Engineering* describes Planguage.

It is important that guidelines for requirement development and documentation be defined at the outset of the project. In all but the smallest programs, careful analysis is required to ensure that the system is developed properly. **Use cases** are one way to document functional requirements, and can lead to more thorough system designs and test procedures. (In most of this book, the broad term **requirement** will be used to denote any type of specification, whether a use case or another type of description of functional aspects of the system.)

In addition to functional requirements, it is also important to consider nonfunctional requirements, such as performance and security, early in the process: They can determine the technology choices and areas of risk. Nonfunctional requirements do not endow the system with any specific functions, but rather constrain or further define how the system will perform any given function. Functional requirements should be specified along with their associated nonfunctional requirements. ([Chapter 9](#) discusses nonfunctional requirements.)

Following is a checklist that can be used by testers during the requirements phase to verify the quality of the requirements.^[3] ^[4] Using this checklist is a first step toward trapping requirements-related defects as early as possible, so they don't propagate to subsequent phases, where they would be more difficult and expensive to find and correct. All stakeholders responsible for requirements should verify that requirements possess the following attributes.

^[3] Suzanne Robertson, "An Early Start To Testing: How To Test Requirements," paper presented at EuroSTAR 96, Amsterdam, December 2–6, 1996. Copyright 1996 The Atlantic Systems Guild Ltd. Used by permission of the author.

^[4] Karl Wiegers, *Software Requirements* (Redmond, Wash.: Microsoft Press, Sept. 1999).

- **Correctness** of a requirement is judged based on what the user wants. For example, are the rules and regulations stated correctly? Does the requirement exactly reflect the user's request? It is imperative that the end user, or a suitable representative, be involved during the requirements phase. Correctness can also be judged based on standards. Are the standards being followed?
- **Completeness** ensures that no necessary elements are missing from the requirement. The goal is to avoid omitting requirements simply because no one has asked the right questions or examined all of the pertinent source documents.

Testers should insist that associated nonfunctional requirements, such as performance, security, usability, compatibility, and accessibility,^[5] are described along with each functional requirement. Nonfunctional requirements are usually documented in two steps:

^[5] Elfriede Dustin et al., "Nonfunctional Requirements," in *Quality Web Systems: Performance, Security, and*

Usability (Boston, Mass.: Addison-Wesley, 2002), Sec. 2.5.

1. A system-wide specification is created that defines the nonfunctional requirements that apply to the system. For 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 should contain a section titled "Nonfunctional Requirements" documenting any specific nonfunctional needs of that particular requirement that deviate from the system-wide nonfunctional specification.
- **Consistency** verifies that there are no internal or external contradictions among the elements within the work products, or between work products. By asking the question, "*Does the specification define every essential subject-matter term used within the specification?*" we can determine whether the elements used in the requirement are clear and precise. For example, a requirements specification that uses the term "viewer" in many places, with different meanings depending on context, will cause problems during design or implementation. Without clear and consistent definitions, determining whether a requirement is correct becomes a matter of opinion.
 - **Testability** (or **verifiability**) of the requirement confirms that it is possible to create a test for the requirement, and that an expected result is known and can be programmatically or visually verified. If a requirement cannot be tested or otherwise verified, this fact and its associated risks must be stated, and the requirement must be adjusted if possible so that it can be tested.
 - **Feasibility** of a requirement ensures that it can be implemented given the budget, schedules, technology, and other resources available.
 - **Necessity** verifies that every requirement in the specification is relevant to the system. To test for relevance or necessity, the tester checks the requirement against the stated goals for the system. Does this requirement contribute to those goals? Would excluding this requirement prevent the system from meeting those goals? Are any other requirements dependent on this requirement? Some irrelevant requirements are not really requirements, but proposed solutions.
 - **Prioritization** allows everyone to understand the relative value to stakeholders of the requirement. Pardee^[6] suggests that a scale from 1 to 5 be used to specify the level of reward for good performance and penalty for bad performance on a requirement. If a requirement is absolutely vital to the success of the system, then it has a penalty of 5 and a reward of 5. A

requirement that would be nice to have but is not really vital might have a penalty of 1 and a reward of 3. The overall value or importance stakeholders place on a requirement is the sum of its penalties and rewards—in the first case, 10, and in the second, 4. This knowledge can be used to make prioritization and trade-off decisions when the time comes to design the system. This approach needs to balance the perspective of the user (one kind of stakeholder) against the cost and technical risk associated with a proposed requirement (the perspective of the developer, another kind of stakeholder).^[7]

^[6] William J. Pardee, *To Satisfy and Delight Your Customer: How to Manage for Customer Value* (New York, N.Y.: Dorset House, 1996).

^[7] For more information, see Karl Wiegers, *Software Requirements*, Ch. 13.

- **Unambiguousness** ensures that requirements are stated in a precise and measurable way. The following is an example of an ambiguous requirement: "The system must respond quickly to customer inquiries." "Quickly" is innately ambiguous and subjective, and therefore renders the requirement untestable. A customer might think "quickly" means within 5 seconds, while a developer may think it means within 3 minutes. Conversely, a developer might think it means within 2 seconds and over-engineer a system to meet unnecessary performance goals.
- **Traceability** ensures that each requirement is identified in such a way that it can be associated with all parts of the system where it is used. For any change to requirements, is it possible to identify all parts of the system where this change has an effect?

To this point, each requirement has been considered as a separately identifiable, measurable entity. It is also necessary to consider the connections among requirements—to understand the effect of one requirement on others. There must be a way of dealing with a large number of requirements and the complex connections among them. Suzanne Robertson^[8] suggests that rather than trying to tackle everything simultaneously, it is better to divide requirements into manageable groups. This could be a matter of allocating requirements to subsystems, or to sequential releases based on priority. Once that is done, the connections can be considered in two phases: first the internal connections among the

requirements in each group, then the connections among the groups. If the requirements are grouped in a way that minimizes the connections between groups, the complexity of tracing connections among requirements will be minimized.

^[8] Suzanne Robertson, "An Early Start to Testing," op. cit.

Traceability also allows collection of information about individual requirements and other parts of the system that could be affected if a requirement changes, such as designs, code, tests, help screens, and so on. When informed of requirement changes, testers can make sure that all affected areas are adjusted accordingly.

As soon as a single requirement is available for review, it is possible to start testing that requirement for the aforementioned characteristics. Trapping requirements-related defects as early as they can be identified will prevent incorrect requirements from being incorporated in the design and implementation, where they will be more difficult and expensive to find and correct.^[9]

^[9] T. Capers Jones, *Assessment and Control of Software Risks* (Upper Saddle River, N.J.: Prentice Hall PTR, 1994).

After following these steps, the feature set of the application under development is now outlined and quantified, which allows for better organization, planning, tracking, and testing of each feature.

Item 3: Design Test Procedures As Soon As Requirements Are Available

Just as software engineers produce design documents based on requirements, it is necessary for the testing team to design test procedures based on these requirements as well. In some organizations, the development of test procedures is pushed off until after a build of the software is delivered to the testing team, due either to lack of time or lack of properly specified requirements suitable for test-procedure design. This approach has inherent problems, including the possibility of requirement omissions or errors being discovered late in the cycle; software implementation issues, such as failure to satisfy a requirement; nontestability; and the development of incomplete test procedures.

Moving the test procedure development effort closer to the requirements phase of the process, rather than waiting until the software-development phase, allows for test procedures to provide benefits to the requirement-specification activity. During the course of developing a test procedure, certain oversights, omissions, incorrect flows, and other errors may be discovered in the requirements document, as testers attempt to walk through an interaction with the system at a very specific level, using sets of test data as input. This process obliges the requirement to account for variations in scenarios, as well as to specify a clear path through the interaction in all cases.

If a problem is uncovered in the requirement, that requirement will need to be reworked to account for this discovery. The earlier in the process such corrections are incorporated, the less likely it is that the corrections will affect software design or implementation.

As mentioned in [Item 1](#), early detection equates to lower cost. If a requirement defect is discovered in later phases of the process, all stakeholders must change the requirement, design, and code, which will affect budgets, schedules, and possibly morale. However, if the defect is discovered during the requirements phase, repairing it is simply a matter of changing and reviewing the requirement text.

The process of identifying errors or omissions in a requirement through test-procedure definition is referred to as **verifying the requirement's testability**. If not enough information exists, or the information provided in the specification is too ambiguous to create a complete test procedure with its related test cases for relevant paths, the specification is not considered to be testable, and may not be suitable for software development. Whether a test can be developed for a

requirement is a valuable check and should be considered part of the process of approving a requirement as complete. There are exceptions, where a requirement cannot immediately be verified programmatically or manually by executing a test. Such exceptions need to be explicitly stated. For example, fulfillment of a requirement that "all data files need to be stored for record-keeping for three years" cannot be immediately verified. However, it does need to be approved, adhered to, and tracked.

If a requirement cannot be verified, there is no guarantee that it will be implemented correctly. Being able to develop a test procedure that includes data inputs, steps to verify the requirement, and known expected outputs for each related requirement can assure requirement completeness by confirming that important requirement information is not missing, making the requirement difficult or even impossible to implement correctly and untestable. Developing test procedures for requirements early on allows for early discovery of nonverifiability issues.

Developing test procedures after a software build has been delivered to the testing team also risks incomplete test-procedure development because of intensive time pressure to complete the product's testing cycle. This can manifest in various ways: For example, the test procedure might be missing entirely; or it may not be thoroughly defined, omitting certain paths or data elements that may make a difference in the test outcome. As a result, defects might be missed. Or, the requirement may be incomplete, as described earlier, and not support the definition of the necessary test procedures, or even proper software development. Incomplete requirements often result in incomplete implementation.

Early evaluation of the testability of an application's requirements can be the basis for defining a testing strategy. While reviewing the testability of the requirements, testers might determine, for example, that using a capture/playback tool would be ideal, allowing execution of some of the tests in an automated fashion. Determining this early allows enough lead time to evaluate and implement automated testing tools.

To offer another example: During an early evaluation phase, it could be determined that some requirements relating to complex and diversified calculations may be more suitable tested with a custom test harness (see [Item 37](#)) or specialized scripts. Test harness development and other such test-preparation activities will require additional lead time before testing can begin.

Moving test procedures closer to the requirements-definition phase of an **iteration**^[10] carries some additional responsibilities, however, including prioritizing test procedures based on requirements, assigning adequate personnel, and understanding the testing strategy. It is often a luxury, if not impossible, to develop all test procedures immediately for each requirement, because of time, budget, and personnel constraints. Ideally, the requirements and subject-matter expert testing teams are both responsible for creating example test scenarios as part of the requirements definition, including scenario outcomes (the expected results).

^[10] An **iteration**, used in an **iterative development process**, includes the activities of requirement analysis, design, implementation and testing. There are many iterations in an iterative development process. A single iteration for the whole project would be known as the **waterfall model**.

Test-procedure development must be prioritized based on an iterative implementation plan. If time constraints exist, test developers should start by developing test procedures for the requirements to be implemented first. They can then develop "draft" test procedures for all requirements to be completed later.

Requirements are often refined through review and analysis in an iterative fashion. It is very common that new requirement details and scenario clarifications surface during the design and development phase. Purists will say that all requirement details should be ironed out during the requirements phase. However, the reality is that deadline pressures require development to start as soon as possible; the luxury of having complete requirements up-front is rare. If requirements are refined later in the process, the associated test procedures also need to be refined. These also must be kept up-to-date with respect to any changes: They should be treated as "living" documents.

Effectively managing such evolving requirements and test procedures requires a well-defined process in which test designers are also stakeholders in the requirement process. See [Item 4](#) for more on the importance of communicating requirement changes to all stakeholders.

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.

In the last case, a defect report should be written. However, if a requirement change process is not being followed, it can be difficult to tell which of the aforementioned scenarios is actually occurring. Is the problem in the software, the requirement, the test procedure, or all of the above? To avoid guesswork, all requirement changes must be openly evaluated, agreed upon, and communicated to all stakeholders. This can be accomplished by having a **requirement-change process** in place that facilitates the communication of any requirement changes to all stakeholders.

If a requirement needs to be corrected, the change process must take into account the ripple effect upon design, code, and all associated documentation, including test documentation. To effectively manage this process, any changes should be baselined and versioned in a configuration-management system.

The change process outlines when, how, by whom, and where change requests are initiated. The process might specify that a change request can be initiated during any phase of the life cycle—during any type of review, walk-through, or inspection during the requirements, design, code, defect tracking, or testing activities, or any other phase.

Each change request could be documented via a **change-request form**—a template listing all information necessary to facilitate the change-request process—which is passed on to the **Change Control Board (CCB)**. Instituting a CCB helps ensure that any changes to requirements and other change requests follow a specific process. A CCB verifies that change requests are documented appropriately, evaluated, and agreed upon; that any affected documents (requirements, design documents, etc.) are updated; and that all stakeholders are informed of the change.

The CCB usually consists of representatives from the various management teams, e.g., product management, requirements management, and QA teams, as well as the testing manager and the configuration manager. CCB meetings can be conducted on an as-

needed basis. All stakeholders need to evaluate change proposals by analyzing the priority, risks, and tradeoffs associated with the suggested change.

Associated and critical impact analysis of the proposed change must also be performed. For example, a requirements change may affect the entire suite of testing documentation, requiring major additions to the test environment and extending the testing by numerous weeks. Or an implementation may need to be changed in a way that affects the entire automated testing suite. Such impacts must be identified, communicated, and addressed before the change is approved.

The CCB determines whether a change request's validity, effects, necessity, and priority (for example, whether it should be implemented immediately, or whether it can be documented in the project's central repository as an enhancement). The CCB must ensure that the suggested changes, associated risk evaluation, and decision-making processes are documented and communicated.

It is imperative that all parties be made aware of any change suggestions, allowing them to contribute to risk analysis and mitigation of change. An effective way to ensure this is to use a **requirements-management tool**,^[11] which can be used to track the requirements changes as well as maintain the traceability of the requirements to the test procedures (see the testability checklist in [Item 2](#) for a discussion of traceability). If the requirement changes, the change should be reflected and updated in the requirements-management tool, and the tool should mark the affected test artifact (and other affected elements, such as design, code, etc.), so the respective parties can update their products accordingly. All stakeholders can then get the latest information via the tool.

^[11] There are numerous excellent requirement management tools on the market, such as Rational's RequisitePro, QSS's DOORS, and Integrated Chipware's RTM: Requirement & Traceability Management.

Change information managed with a requirements-management tool allows testers to reevaluate the testability of the changed requirement as well as the impact of changes to test artifacts (test plan, design, etc.) or the testing schedule. The affected test procedures must be revisited and updated to reflect the requirements and implementation changes.

Previously identified defects must be reevaluated to determine whether the requirement change has made them obsolete. If scripts, test harnesses, or other testing mechanisms have already been created, they may need to be updated as well.

A well-defined process that facilitates communication of changed requirements, allowing for an effective test program, is critical to the efficiency of the project.

Item 5: Beware of Developing and Testing Based on an Existing System

In many software-development projects, a legacy application already exists, with little or no existing requirement documentation, and is the basis for an architectural redesign or platform upgrade. Most organizations in this situation insist that the new system be developed and tested based exclusively on continual investigation of the existing application, without taking the time to analyze or document how the application functions. On the surface, it appears this will result in an earlier delivery date, since little or no effort is "wasted" on requirements reengineering or on analyzing and documenting an application that already exists, when the existing application in itself supposedly manifests the needed requirements.

Unfortunately, in all but the smallest projects, the strategy of using an existing application as the requirements baseline comes with many pitfalls and often results in few (if any) documented requirements, improper functionality, and incomplete testing.

Although some functional aspects of an application are self-explanatory, many domain-related features are difficult to reverse-engineer, because it is easy to overlook business logic that may depend on the supplied data. As it is usually not feasible to investigate the existing application with every possible data input, it is likely that some intricacy of the functionality will be missed. In some cases, the reasons for certain inputs producing certain outputs may be puzzling, and will result in software developers providing a "best guess" as to why the application behaves the way it does. To make matters worse, once the actual business logic is determined, it is typically not documented; instead, it is coded directly into the new application, causing the guessing cycle to perpetuate.

Aside from business-logic issues, it is also possible to misinterpret the meaning of user-interface fields, or miss whole sections of user interface completely.

Many times, the existing baseline application is still live and under development, probably using a different architecture along with an older technology (for example, desktop vs. Web versions); or it is in production and under continuous maintenance, which often includes defect fixing and feature additions for each new production release. This presents a "moving-target" problem: Updates and new features are being applied to the application that is to serve as the requirements baseline for the new product, even as it is being reverse-engineered by the developers and testers for the new application. The resulting new application may become a mixture of the different states of the existing application as it has moved through its own development life cycle.

Finally, performing analysis, design, development, and test activities in a "moving-target" environment makes it difficult to properly estimate time, budgets, and staffing required for the entire software development life cycle. The team responsible for the new application cannot effectively predict the effort involved, as no requirements are available to clarify what to build or test. Most estimates must be based on a casual understanding of the application's functionality that may be grossly incorrect, or may need to suddenly change if the existing application is upgraded. Estimating tasks is difficult enough when based on an excellent statement of requirements, but it is almost impossible when so-called "requirements" are embodied in a legacy or moving-target application.

On the surface, it may appear that one of the benefits of building an application based on an existing one is that testers can compare the "old" application's output over time to that produced by the newly implemented application, if the outputs are supposed to be the same. However, this can be unsafe: What if the "old" application's output has been wrong for some scenarios for a while, but no one has noticed? If the new application is behaving correctly, but the old application's output is wrong, the tester would document an invalid defect, and the resulting fix would incorporate the error present in the existing application.

If testers decide they can't rely on the "old" application for output comparison, problems remain. Or if they execute their test procedures and the output differs between the two applications, the testers are left wondering which output is correct. If the requirements are not documented, how can a tester know for certain which output is correct? The analysis that should have taken place during the requirements phase to determine the expected output is now in the hands of the tester.

Although basing a new software development project on an existing application can be difficult, there are ways to handle the situation. The first step is to manage expectations. Team members should be aware of the issues involved in basing new development on an existing application. The following list outlines several points to consider.

- *Use a fixed application version.* All stakeholders must understand why the new application must be based on one specific version of the existing software as described and must agree to this condition. The team must select a version of the existing application on which the new development is to be based, and use only that version for the initial development.

Working from a fixed application version makes tracking defects more straightforward, since the selected version of the existing application will determine whether there is a defect in the new application, regardless of upgrades or corrections to the existing application's code base. It will still be necessary to verify that the existing application is indeed correct, using domain expertise, as it is important to recognize if the new application is correct while the legacy application is defective.

- *Document the existing application.* The next step is to have a domain or application expert document the existing application, writing at least a paragraph on each feature, supplying various testing scenarios and their expected output. Preferably, a full analysis would be done on the existing application, but in practice this can add considerable time and personnel to the effort, which

may not be feasible and is rarely funded. A more realistic approach is to document the features in paragraph form, and create detailed requirements only for complex interactions that require detailed documentation.

It is usually not enough to document only the user interface(s) of the current application. If the interface functionality doesn't show the intricacies of the underlying functional behavior inside the application and how such intricacies interact with the interface, this documentation will be insufficient.

- *Document updates to the existing application.* Updates—that is, additional or changed requirements—for the existing baseline application from this point forward should be documented for reference later, when the new application is ready to be upgraded. This will allow stable analysis of the existing functionality, and the creation of appropriate design and testing documents. If applicable, requirements, test procedures, and other test artifacts can be used for both products.

If updates are not documented, development of the new product will become "reactive": Inconsistencies between the legacy and new products will surface piecemeal; some will be corrected while others will not; and some will be known in advance while others will be discovered during testing or, worse, during production.

- *Implement an effective development process going forward.* Even though the legacy system may have been developed without requirements, design or test documentation, or any system-development processes, whenever a new feature is developed for either the previous or the new application, developers should make sure a system-development process has been defined, is communicated, is followed, and is adjusted as required, to avoid perpetuating bad software engineering practices.

After following these steps, the feature set of the application under development will have been outlined and quantified, allowing for better organization, planning, tracking, and testing of each feature.

Chapter 2. Test Planning

The cornerstone of a successful test program is effective test planning. Proper test planning requires an understanding of the corporate culture and its software-development processes, in order to adapt or suggest improvements to processes as necessary.

Planning must take place as early as possible in the software life cycle, because lead times must be considered for implementing the test program successfully. Gaining an understanding of the task at hand early on is essential in order to estimate required resources, as well as to get the necessary buy-in and approval to hire personnel and acquire testing tools, support software, and hardware. Early planning allows for testing schedules and budgets to be estimated, approved, and then incorporated into the overall software development plan.

Lead times for procurement and preparation of the testing environment, and for installation of the system under test, testing tools, databases, and other components must be considered early on.

No two testing efforts are the same. Effective test planning requires a clear understanding of all parts that can affect the testing goal. Additionally, experience and an understanding of the testing discipline are necessary, including best practices, testing processes, techniques, and tools, in order to select the test strategies that can be most effectively applied and adapted to the task at hand.

During test-strategy design, risks, resources, time, and budget constraints must be considered. An understanding of estimation techniques and their implementation is needed in order to estimate the required resources and functions, including number of personnel, types of expertise, roles and responsibilities, schedules, and budgets.

There are several ways to estimate testing efforts, including ratio methods and comparison to past efforts of similar scope. Proper estimation allows an effective test team to be assembled—not an easy

task, if it must be done from scratch—and allows project delivery schedules to most accurately reflect the work of the testing team.

Item 6: Understand the Task At Hand and the Related Testing Goal

Testing, in general, is conducted to verify that software meets specific criteria and satisfies the requirements of the end user. Effective testing increases the probability that the application under test will function correctly under all circumstances and will meet the defined requirements, thus satisfying the end users of the application. Essential to achieving this goal is the detection and removal of defects in the software, whether through inspections, walk-throughs, testing, or excellent software development practices.

A program is said to function correctly when:

- Given valid input, the program produces the correct output as defined by the specifications.
- Given invalid input, the program correctly and gracefully rejects the input (and, if appropriate, displays an error message).
- The program doesn't hang or crash, given either valid or invalid input.
- The program keeps running correctly for as long as expected.
- The program achieves its functional and nonfunctional requirements. (For a discussion of nonfunctional requirements, see [Chapter 9](#)).

It is not possible to test all conceivable combinations and variations of input to verify that the application's functional and nonfunctional requirements have been met under every possible scenario.

Additionally, it is well known that testing alone cannot produce a quality product—it is not possible to "test quality into a product." Inspections, walk-throughs, and quality software engineering processes are all necessary to enhance the quality of a product.

Testing, however, may be seen as the final "quality gate."

Test strategies (see [Item 7](#)) must be defined that will help achieve the testing goal in the most effective manner. It is often not possible to fix all known defects and still meet the deadlines set for a project, so defects must be prioritized: It is neither necessary nor cost-effective to fix all defects before a release.

The specific test goal varies from one test effort to another, and from one test phase to the next. For example, the goal for testing a program's functionality is different from the goal for performance or configuration testing. The test planner, usually the test manager or test lead, must ask: What is the goal of the specific testing effort? This goal is based on criteria the system must meet.

Understanding the task at hand, its scope, and its associated testing goals are the first steps in test planning. Test planning requires a clear understanding of every piece that will play a part in achieving the testing goal.

How can the testing team gather this understanding? The first inclination of the test manager might be to take all of the documented requirements and develop a test plan, including a testing strategy, then break down the requirements feature-by-feature, and finally, task the testing team with designing and developing test procedures. However, without a broader knowledge of the task at hand, this is an ill-advised approach, since it is imperative that the testing team first understand all of the components that are a part of the testing goal. This is accomplished through the following:

- *Understanding the system.* The overall system view includes understanding of the functional and nonfunctional requirements (see [Item 2](#)) the system under test must meet. The testing team must understand these requirements to be effective. Reading a disjointed list of "The system shall..." statements in a requirements-specification document will hardly provide the

necessary overall picture, because the scenarios and functional flow are often not apparent in such separate and isolated statements. Meetings involving overall system discussions, as well as documentation, should be made available to help provide the overall picture. Such documents would include proposals discussing the problem the proposed system is intended to solve. Other documents that can help further the understanding of the system may include statements of high-level business requirements, product management case studies, and business cases. For example, systems where there is no tolerance for error, such as medical devices where lives are at stake, will require a different approach than some business systems where there is a higher tolerance for error.

- *Early involvement.* The test manager, test lead, and other test-team members as necessary should be involved during the system's inception phase, when the first decisions about the system are made, in order to gain insights into tasks at hand. Such involvement adds to understanding of customer needs, issues, potential risks, and, most importantly, functionality.
- *Understanding corporate culture and processes.* Knowledge of the corporate culture and its software-development processes is necessary to adapt or suggest improvements as needed. Though every team member should be trying to improve the processes, in some organizations process improvement is the responsibility primary of QA and a Process Engineering Group.

The test manager must understand the types of processes that are supported in order to be able to adapt a testing strategy that can achieve the defined goal. For example:

- Does the testing effort involve a testing team independent from the development team, as opposed to having test engineers integrated with the development team?
- Is an "extreme programming^[1]" development effort underway, to which testing methods must be adapted?

^[1] For a description of extreme programming, see [Item 29, Endnote 1](#).

- Is the testing team the final quality gate—does the testing team give the green light regarding whether the testing criteria have been met?
- *Scope of Implementation.* In addition to comprehending the problem the system is intended to solve and the corporate culture, the team must understand the scope of the implementation in order to scale the testing scope accordingly.
- *Testing expectations.* What testing expectations does management have? What type of testing does the customer expect? For example, is a user-acceptance testing phase required? If so, which methodology must be followed, if any are specified, and what are the expected milestones and deliverables? What are the expected testing phases? Questions such as these are often answered in a project-management plan. The answers should all end up in the test plan.
- *Lessons learned.* Were any lessons learned from previous testing efforts? This is important information when determining strategies and for setting realistic expectations.
- *Level of effort.* What is the expected scale of the effort to build the proposed system? How many developers will be hired? This information will be useful for many purposes, such as projecting the complexity and scale of the testing effort, which can be based on a developer-to-tester ratio. See [Item 12](#) for a discussion of test-effort estimation.
- *Type of solution.* Will the ultimate, most complex solution be implemented, or a more cost-effective solution that requires less development time? Knowing this will help the test planner understand the type of testing required.
- *Technology choices.* What technologies have been selected for the system's implementation, and what are the potential issues associated with them? What kind of architecture will the system use? Is it a desktop application, a client-server application, or a

Web application? This information will help in determining test strategies and choosing testing tools.

- *Budget*. What is the budget for implementing this product or system, including testing? This information will be helpful in determining the types of testing possible given the level of funding. Unfortunately, budgets are often determined without any real effort at estimating testing costs, requiring the test manager to adapt the testing effort to fit a predetermined budget figure.
- *Schedule*. How much time has been allocated for developing and testing the system? What are the deadlines? Unfortunately, schedules too often are determined without any real test-schedule estimation effort, requiring the test manager to adapt a testing schedule to the fit a predetermined deadline.
- *Phased solution*. Does the implementation consist of a **phased** solution, with many releases containing incremental additions of functionality, or will there be one big release? If the release is phased, the phases and priorities must be understood so test development can be matched to the phases and implemented according to the iterations.

A broad understanding of what the system-to-be is to accomplish, its size and the corresponding effort involved, customer issues, and potential risks allows the test manager to comprehend the testing task at hand and architect the testing goal and its associated testing framework, eventually to be documented and communicated in a test plan or test-strategy document.

In addition, lead time is required to determine budgets and schedules and to meet other needs, such as procuring hardware and software required for the test environment, and evaluating, purchasing, and implementing a testing tool. The earlier the need for a test tool can be established, the better the chances that the correct tool can be acquired and applied effectively.

Item 7: Consider the Risks

Test-program **assumptions**, **prerequisites**, and **risks** must be understood before an effective testing strategy can be developed. This includes any events, actions, or circumstances that may prevent the test program from being implemented or executed according to schedule, such as late budget approvals, delayed arrival of test equipment, or late availability of the software application.

Test strategies include very specific activities that must be performed by the test team in order to achieve a specific goal. Many factors must be considered when developing test strategies. For example, if the application's architecture consists of several tiers, they must be considered when devising the testing strategy.

Test strategies generally must incorporate ways to minimize the risk of cost overruns, schedule slippage, critical software errors, and other failures. During test-strategy design, constraints on the task at hand (see [Item 6](#)), including risks, resources, time limits, and budget restrictions, must be considered.

A test strategy is best determined by narrowing down the testing tasks as follows:

- *Understand the system architecture.* Break down the system into its individual layers, such as user interface or database access. Understanding the architecture will help testers define the testing strategy for each layer or combination of layers and components. For further discussion of the system architecture, see [Chapter 4](#).
- *Determine whether to apply GUI testing, back-end testing, or both.* Once the system architecture is understood, it can be determined how best to approach the testing—through the Graphical User Interface (GUI), against the back-end, or both. Most testing efforts will require that testing be applied at both levels, as the GUI often contains code that must be exercised and verified.

When determining the testing strategy, testers should keep in mind that the overall complexity of, and degree of expertise required for, business-layer (back-end) testing is much greater than for user-interface testing employed on the front-end parts of the application. This is because more complex language and technology skills are required to write tests that access the business layer—for example, self-sufficient, experienced C++ programmers may be needed if the business layer is written in C++. GUI tools and GUI testing, on the other hand, do not require extensive programming

background; general programming skills or subject matter expertise (depending on the type of tester) may be the main requirements.

When determining the type of testing strategy to deploy, testers should consider at what point additional testing yields diminishing returns. For example, suppose an application is being developed for which GUI tests can successfully execute approximately 75% of the functionality, and approximately 25% can be tested using targeted business-layer tests (targeted at the highest-risk areas of the application) over the same period of time. It would not be necessary (or reasonable) to test the remaining 75% of the business layer. The effort would require a much higher number of skilled development resources, increase costs, and probably consume more time. It would be excessive and unnecessary to perform low-level testing on the remaining 75% because that functionality is exercised via the GUI, which will also have been tested.

Ideally, the GUI scripts will have been developed in such a manner that they rarely break. If not, then it may be necessary to perform further business-level testing.

The example of a 75:25 proportion of GUI to business-layer testing may require two GUI automaters and one business-layer test developer, in addition to several domain expert testers, who use manual techniques all depending on the size and complexity of the application. (For a discussion of Automated Testing, see [Chapter 8](#).) In this example, five or more skilled C++ developers may be required to properly test the entire application at the business layer, depending on the number of layers, or logical tiers, in the application. They would need to work in lockstep with the development team, since the underlying business-layer interfaces can (and often do) change from build to build without affecting the GUI.

Low-level testing would not be necessary for the 75% of the application that follows straightforward record retrieval and record update models. However, GUI testing would still be required to make sure there are no data-handling issues or other minor flaws in the GUI itself.

The previous example demonstrates why it is important to apply testing strategies that make sense considering risk, complexity, and necessity. There is no hard-and-fast rule, however; each project requires its own individual analysis.

- *Select test-design techniques.* Narrowing down the types of testing techniques to be used can help reduce a large set of possible input combinations and variations. Various test-design techniques are available, and which to use must be determined as part of the testing strategy. Some test-design techniques are discussed in [Chapter 5](#).
- *Select testing tools.* When developing the testing strategy, testers must determine the types of vendor-provided testing tools to be used, if any, given the testing task at hand; how the tools will be used; and which team members will be using them. It may also be necessary to build a tool instead of buying one. See [Chapter 7](#) for a discussion of automated testing tools, including build-versus-buy decisions.
- *Develop in-house test harnesses or scripts.* The test designers may decide to develop an in-house automated test harness (see [Item 37](#)) or test tool, given the knowledge of the task at hand and the types of automated testing tools available on the market. When the testing task does not lend itself to a vendor-provided automated testing tool, in-house script and test harness development may be necessary.
- *Determine test personnel and expertise required.* Based on the test strategy outline, the required test personnel and expertise must be determined. If developing a test harness, a developer (i.e., a tester who has development skills) must be included on the testing team. If part of the testing strategy involves automation using capture/playback tools, automation skills will be required. Also needed is someone with domain expertise—knowledge in the subject matter to be tested. Not having the right skills in a testing team can significantly endanger the success of the testing effort. See [Chapter 3](#) for a discussion of the testing team.
- *Determine testing coverage.* Testers must understand the testing coverage required. In some cases, for example, there may be contractual agreements that list all functional requirements to be tested, or code-coverage requirements. In other cases, testers must determine the testing coverage, given the resources, schedules, tools, task at hand, and risks of not testing an item. At the outset, testers should document what the testing will cover and what it won't.
- *Establish release criteria.* Stating the testing coverage is closely related to defining release or exit criteria. **Release criteria** indicate when testing can be considered complete, so it is important that they are documented upfront. For example, a release criterion might state that the application can ship with cosmetic defects, whereas all urgent and "show-stopper" defects must be fixed before release. Another release criterion might state that a certain high-priority feature must be working properly before the release can go out.

- *Set the testing schedule.* The testing strategy must be tailored to the time allotted to the testing effort. It's important that a detailed schedule be included in the testing strategy to avoid implementing a strategy that doesn't meet the required schedule.
- *Consider the testing phases.* Different test strategies apply to different testing phases. For example, during the functional testing phase, tests verify that the functional requirements have been met, while during the performance testing phase, performance tests verify that performance requirements have been met. Are there plans in place to have the system alpha tested or beta tested? Understanding the testing phases is necessary in order to develop adequate testing strategies for each phase.

These are just some of the issues to consider when developing a testing strategy. Since there is never enough time to test a system exhaustively, it is important to understand **project risks** when defining a testing strategy. Risk scenarios must be planned for, isolated to some degree, and managed. To this end, the test team must prioritize requirements and assess the risk inherent in each. The team must review the critical functions and high-risk elements identified for the system, and consider this information when determining the priority order of the test requirements.

When determining the order of test procedure development, the test team should review requirements to ensure that requirement areas have been prioritized, from most critical to least critical functionality, by the product-management or requirements team. Input from end users should have been considered in determining relative importance of functions. The prioritized requirements list should be made available to the testing team.

In addition to prioritizing areas of functionality, it is helpful to group requirements into related functional paths, scenarios, or flow. This makes it easier to assign test tasks to the various test engineers.

The criteria listed below for determining the order in which to group requirements are based on the recommendations of Rational Software Corporation.^[1]

^[1] Rational Unified Process 5.0 (Rational Software Corporation, retrieved Sept. 7, 2002 from <http://www.rational.com/products/rup/index.jsp>).

- *Risk level.* After risk assessment, test requirements are prioritized to ensure mitigation of high risks to system performance or the potential exposure of the company to liability. High-risk issues may include functions that prohibit

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.

- *Frequency of use.* The potential failure of functionality that is used most often in an application (the "core" of the application) poses a high risk.
- *Untestable requirements.* Functional or nonfunctional requirements that cannot be tested pose a high risk to the success of the system. However, if the testing team has verified the testability of the requirements during the requirements phase (see [Chapter 1](#)), this issue may be minimized.

As risks are identified, they must be assessed for impact and then mitigated with strategies for overcoming them, because risks may be realized even though all precautions have been taken. The test team must carefully examine risks in order to derive effective test and mitigation strategies for a particular application.

Risk assessments must consider probability of a risk being realized and usage patterns that may cause the problem to appear, as well as mitigation strategies. The magnitude or impact of the potential problem should be evaluated. Risk mitigation strategies should be defined for those system requirements where there is the greatest probability of the risk being realized, balanced with amount of use and the importance of an area. Usually, the primary areas of an application should be allowed to contain few if any defects, while rarely used and less-important items may be allowed more latitude.

It is difficult to assess risk in great detail. A large number of people should contribute to this process, such as the customer representative (product management), end users, and the requirements, development, testing, and QA teams.

The degree of risk needs to be communicated so that decisions are informed ones. If the risk is too high, a recommendation can be made that the project not proceed in its current form, and be modified or discontinued altogether.

Risk analysis provides information that can assist the test manager or test-team lead in making difficult decisions such as assignment of testers based on skill level, effort to be expended, and risk and quality goals. After performance of a risk assessment in which a deliverable is determined to have high risk factors, and failure to deliver might have a strong negative impact, a decision may be made to assign experienced tester(s) to this area.

One testing strategy that helps mitigate risks is to focus the testing on the parts of the system that are likely to cause most of the problems. The test engineer or test manager must identify the parts of an implementation cycle that pose the greatest risk, and the functionality that most likely will cause problems or failures.

On the other hand, for a deliverable that has low risk factors and low impacts, inspections may cover all the testing necessary for this particular deliverable. Low-risk, low-impact deliverables can also provide opportunities for new or less experienced testers to gain practice with minimal risk.

Careful examination of goals and risks is necessary for development of an appropriate set of test strategies, which in turn produces a more predictable, higher-quality outcome.

Item 8: Base Testing Efforts on a Prioritized Feature Schedule

Software feature implementations must be prioritized for each incremental software release, based on customer needs or the necessity to deliver some high-risk items first. However, test-procedure planning and development should be based not only on priority and risk, as discussed in [Item 7](#), but also on the software feature implementation schedule, as the latter dictates the order in which features are made available for testing.

It is important that the software development schedule, including the feature implementation sequence, is decided early and made available to the testing team so it can plan accordingly. This is especially true when time is limited, to avoid wasting time developing tests for features that won't be made available until later releases. Frequent changes to the feature schedule should be avoided, as each modification requires changes in development and testing plans and schedules.

Feature prioritization is especially crucial for phased releases. In most cases, the development schedule should strive to deliver the most needed features first. Those features, in turn, should be tested first.

Feature lists can be prioritized based on various criteria:

- *Highest to lowest risk.* As described in [Item 7](#), it is important to consider levels of risk when developing a project schedule and the associated testing strategy. Focusing development and testing on the highest-risk features is one way to prioritize.
- *Highest to lowest complexity.* Prioritizing by complexity—attempting to develop and test the most complex features first—may minimize schedule overruns.
- *Customer needs.* For most projects, there is a tendency to prioritize feature delivery based on customer needs, since this is usually required to fuel the marketing and sales activities associated with a product.
- *Budget constraints.* The vast majority of software projects go over their budgets. It's important to consider the allotted testing budget when prioritizing the features for a given release. Some features will be more important to the success of a program than others.
- *Time constraints.* It's important to consider time constraints when prioritizing the features for a given release.

- *Personnel constraints.* When prioritizing the feature schedule, planners should consider availability of personnel. Key personnel required to implement a specific feature might be missing from the team because of budget constraints or other issues. While prioritizing features, it's important to consider not only "what" but "who."

A combination of several of these approaches is typically necessary to arrive at an effective feature schedule. Each feature can be given a "score" for risk, complexity, customer need, and so on to arrive at a total value, or "weight," for that feature. Once this is done for each feature, the list can be sorted by weight to arrive at a prioritized list of features.

Item 9: Keep Software Issues in Mind

when developing testing plans, it is important for the testing team to understand the software issues affecting project development and delivery. These include:

- *Beta or pre-release products.* The development team may be implementing new features on a beta version of a particular technology or operating system. When a development team is working with the beta version of some technology, chances are that no automated testing tool is available that specifically supports all of the beta technology. The test plan must take this into account, and must provide for evaluation of all technologies used by development team as early as possible in the life cycle to determine whether available testing tools will suffice.
- *New or cutting-edge technologies.* Implementation of new technologies inevitably causes disruptions in development, with the potential for ripple effects throughout many stages of the software development life cycle, including testing activities. Sometimes a new technology can present so many problems that it may be necessary to reengineer a portion of the product. For example, suppose the development group has implemented a solution using the beta version of an operating system, but when the release version becomes available, it has been rearchitected in such a way that it requires the development team to reengineer parts of the implementation. This, in turn, requires the testing team to run another detailed regression test, to modify its gray-box^[1] testing efforts, and forces the testing team to redevelop automated testing scripts that had already been created. Issues such as these must be considered during test planning.

^[1] Gray-box testing exercises the application, either through the user interface or directly against the underlying components, while monitoring internal component behavior to determine the success or failure of the test. See [Item 16](#).

- *Staged implementation.* Prior to completion of the initial version of a system, functionality becomes available in pieces. Testing must be coordinated with feature delivery. For example, the application may provide a user interface for entering a set of data, but the interface for viewing that data may not be available until much later. The test plan must accommodate such conditions by providing alternative ways of testing whether the application correctly processes and stores the data.

- *Defects.* Defects may prevent testing from proceeding in numerous areas. It may be impossible for test procedures to be executed in their entirety: The system may fail prior to the execution of all test steps. This is especially true during the early phases of software construction, when the number of defects is high. To properly handle this situation, it is necessary to communicate to the development team that defects are preventing testing from continuing, so repairing these defects can be accorded higher priority.
- *Patches and service packs.* Operating system vendors and other third-party software suppliers often provide updates to correct problems and introduce new features. If the application under test will be affected by such updates, it is important to factor this into the test plan. For example, when a new version of a popular Web browser is released, many users will upgrade to this new version. The test plan must make provisions for acquiring the new browser as soon as possible and testing the compatibility of the application with this browser, as well as with older versions. Similarly, when an operating system service pack is released, many users will install the service pack. The application must be tested for compatibility with this update as soon as possible. Testers should not rely upon assurance that a service pack is supposed to be backward compatible.

Item 10: Acquire Effective Test Data

During the creation of detailed test designs (discussed in [Chapter 5](#)), test data requirements are incorporated into test cases, which in turn are part of the test procedures.^[1] An effective test strategy requires careful acquisition and preparation of test data. Functional testing can suffer if test data is poor. Conversely, good data can help improve functional testing. Good test data can be structured to improve understanding and testability. Its contents, correctly chosen, can reduce maintenance effort. When requirements are vague, preparation of high-quality test data can help to focus the business.

^[1] Per ANSI/IEEE Standard 829-1987, test design identifies the test cases, which contain the test data.

A good **data dictionary** and detailed **design documentation** can be useful in developing sample data. In addition to providing data-element names, the data dictionary may describe data structures, cardinality, usage rules, and other useful information. Design documentation, particularly database schemas, can also help identify how the application interacts with data as well as relationships between data elements.

Given the sheer number of possibilities, it is usually not possible to test all possible combinations and variations of inputs and outputs to verify that the application's functional and nonfunctional requirements have been met under every conceivable condition. However, various test-design techniques are available to help narrow down the data input and output combinations and variations. One such test technique is **data flow coverage**. It is designed to incorporate the flow of data into the selection of test-procedure steps, helping to identify test paths that satisfy some characteristic of data flow for all applicable paths.

Boundary-condition testing is another testing technique. Test data is prepared that executes each **boundary condition** (limit on quantity or content of allowable data, set in the software design). The behavior of systems usually changes at their boundaries. Errors congregate at the boundaries; therefore, testing system behavior at these boundaries is a very effective technique.

Boundary conditions should be listed in the requirement statements—for example, "the system will support a range of 1 to 100 items in a pick-list control." In this example, the boundary conditions tested could be 100+1 (outside the boundary), 100-1 (inside the boundary), 100 (on the boundary); and 1 (on the boundary), null

(no input), and 0 (outside the boundary). When boundary conditions are not specified in the system requirements, the test plan must incorporate tests that actually find the boundaries. These types of tests are very common, since developers often don't know the boundaries until they are determined via focused testing.

Ideally, such issues would have been resolved while determining the testability of the requirements. Given the complexity of today's technologies, however, determining system boundaries during the requirements phase often is not possible. One alternative way to do so is to develop and test a prototype of the software.

Test data must be designed to allow each system-level requirement to be tested and verified. A review of test-data requirements should address several data concerns, including those listed below.^[2]

^[2] Adapted from the SQA Suite Process, Rational Unified Process 5.0 in Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process* (Reading, Mass.: Addison-Wesley, Feb. 1999).

- *Depth.* The test team must consider the quantity and size of database records needed to support tests. The test team must identify whether 10 records within a database or particular table are sufficient, or 10,000 records are necessary. Early life-cycle tests, such as unit or build verification, should use small, handcrafted databases, which offer maximum control and the most-focused effort. As the test effort progresses through the different phases and types of tests, the size of the database should increase as appropriate for particular tests. For example, performance and volume tests are not meaningful if the production environment database contains 1,000,000 records, but tests are performed against a database containing only 100 records. For additional discussion on performance testing and the importance that it be done early in the development lifecycle, see [Chapter 9](#).
- *Breadth.* Test engineers must investigate variations within the data values (e.g., 10,000 different accounts containing different data values, and a number of different types of accounts containing different *types* of data). A well-designed test will account for variations in test data, whereas tests for which all the data is similar will produce limited results.

With regard to records containing different data *values*, testers may need to consider, for example, that some accounts may have negative balances, or

balances in the low range (e.g., \$100s), moderate range (\$1,000s), high range (\$100,000s), and very high range (\$10,000,000s). With regard to records containing different data *types*, testers must consider, for example, bank customer accounts that might be classified in several ways, including savings, checking, loans, student, joint, and business accounts.

- *Scope.* The scope of test data is pertinent to the accuracy, relevance, and completeness of the data. For example, when testing queries used to identify the various kinds of accounts at a bank that have balance-due amounts greater than \$100, not only should there be numerous accounts in the test data that meet this criterion, but the tests must also reflect additional data, such as reason codes, contact histories, and account owner demographic data. The use of complete test data enables the test procedure to fully validate and exercise the system and to support the evaluation of results. The test engineer must also verify that the inclusion of a record returned as a result of a query (e.g., accounts with balance-due amounts greater than \$100) is valid for the specific purpose of the query, and not due to a missing or inappropriate value.
- *Data integrity during test execution.* The test team must maintain data integrity while performing tests. The test team must be able to segregate data, modify selected data, and return the database to its initial state throughout test operations. The test team must make sure that when several test engineers are performing tests at the same time, one test won't adversely affect the data required for the other tests. For example, if one test team member is modifying data values while another is running a query, the result of the query may not be as expected. One way to prevent one tester's work from interfering with another's is to assign a separate test database or data file to each tester, if feasible and cost-effective. Another way is to assign separate testing tasks to each tester, focusing each on a specific area of functionality that does not overlap with others being tested.
- *Conditions.* Data sets should be created that reflect specific "conditions" in the domain of the application, meaning certain patterns of data that would otherwise be arrived at only after performing specific operations over time. For example, financial systems commonly perform year-end closeouts. Storing data in the year-end condition enables the system to be tested in a year-end closeout state without having to first enter the data for the entire year. Creating test data that is already in the closeout state eases testing, since testers can simply load the closeout test data without having to perform many operations to get data into the closeout state.

When identifying test-data requirements, it is beneficial to develop a table listing the test procedures in one column and test-data requirements in another column. Among the requirements, it is important to note the size of the data set needed, and how long it will take to generate the test data. While a small data subset is good enough for functional testing, a production-sized database is required for performance testing. It can take a long time to acquire production-size data, sometimes as long as several months.

The test team also must plan the means for obtaining, generating, or developing the test data. The mechanism for refreshing the test database to an original state, to enable all testing activities including regression testing, also must be devised and documented in the project test plan. Testers must identify the names and locations of the applicable test databases and repositories necessary to exercise and test the software application.

Data usually must be prepared prior to testing. **Data preparation** may involve the processing of raw data text, or files; consistency checks; and in-depth analysis of data elements, including defining data to test case-mapping criteria, clarifying data-element definitions, confirming primary keys, and defining acceptable data parameters. In order to prepare the data, as well as to develop environment setup scripts and test-bed scripts, the test team must obtain and modify all needed test databases. Ideally, existing customer data is available that includes realistic combinations and variations of data scenarios. An advantage of using actual customer data is that it may include some combinations or usage patterns that had not been considered by test designers. Having actual customer data available during testing can be a useful reality check for the application.

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

large files (especially log files on client-server systems) post-test. Archiving is almost a necessity in current testing environments.

- Identify network characteristics of the customer environment, such as the use of leased lines, modems, Internet connections, and protocols such as Ethernet, IPX, and TCP/IP.
- In the case of a client-server or Web-based system, identify the required server operating systems, databases, and other components.
- Identify the number of automated test tool-licenses required by the test team.
- Identify other software needed to execute certain test procedures, such as word processors, spreadsheets, and report writers.
- When specifying test-environment hardware, consider test-data requirements, including the size of the test databases if applicable. It is important to ensure that the machines possess sufficient capacity and that resources needed to install the data, such as a tape drive or network connection, are available.
- Consider special resources needed for configuration testing, such as removable hard drives and image-retrieval software.

Following these preparatory activities, the test team develops a test-environment design that consists of a graphic representation of the test-environment architecture, together with a list of the components required to support that architecture. The list of components must be reviewed to determine which components are already in place, which can be shifted from other locations within the organization, and which must be procured. The list of components that must be procured comprises a **test equipment purchase list**. It enumerates quantities required, unit price information, and maintenance and support costs. The test team may want to include a few backup components, in order to ensure that testing can continue in the event of hardware failure.

Item 12: Estimate Test Preparation and Execution Time^[1]

^[1] Elfriede Dustin *et al.*, *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), Section 5.3.

Before it is possible to complete that ideal test plan and the best testing strategy, it is necessary to estimate how long it will take to prepare and execute the tests. It is important to remember to include testing time estimates when scoping out the overall software development schedule.

Historically, development and overall project effort have been the predominant focus of software development program estimation. The time required for product quality assurance disciplines, such as software testing, has often been loosely estimated in relation to the amount of anticipated development effort or overall project effort. However, because of variability in testing protocols, it is usually insufficient to estimate testing time in this way; a variety of factors that will affect a particular testing effort must be taken into account.

The test effort applied to a particular project depends on a number of variables. These include the culture or "test maturity" of the organization, the complexity of the software application under test, the scope of test requirements defined for the project, the skill levels of the individuals performing the tests, and the type of test-team organization supporting the test effort. These variables can be used as inputs into complex models supported by automated estimation tools in order to develop test-team resource estimates.

However, given the large number of variables that affect the magnitude of the test effort, the use of complex equations is not generally useful, as estimation formulas are rarely precise.^[2] Such methods of projecting the level of effort for a test program do not normally produce an outcome that reflects the actual amount of effort that applies. Moreover, they are too cumbersome. Simpler estimation models, on the other hand, can serve well.

^[2] For more information on this topic, see J.P. Lewis, "Large Limits to Software Estimation," *ACM Software Engineering Notes* 26:4 (July, 2001): 54–59.

Ideally, test estimation should always begin with either a **work breakdown structure** (WBS) or a detailed list of testing tasks from which a "testing WBS" can be built.

To be most effective, the methods presented here must be adapted to the organization's needs and processes.

Development Ratio Method

Given the predominant focus within the industry on estimating software development efforts, the exercise of sizing a test program is commonly based on the Development Ratio Method. This quick and easy way to gauge the level of effort required for a test program is centered upon the number of software developers planned for the effort. The size of the test team is calculated using the ratio of developers to test engineers on a project. The term **developers** in this case includes personnel dedicated to performing design-, development-, and unit-level test activities.

The result derived from this ratio depends on the type of software development effort, as shown in [Table 12.1](#). The ratios in [Table 12.1](#) are applicable when the scope of the test effort involves functional and performance testing at the integration and system test levels. Note that the ratio of test team members to developers depends on the type and complexity of software being developed. For example, when a commercial vendor creates software intended for a large worldwide audience, the emphasis on testing increases, so the ratio of test engineers to developers increases. A test effort for a software product with a smaller market requires a lower ratio of test engineers to developers, all other factors being equal.

Additionally, at different times during the development life-cycle, the ratio may change. During the later testing phases, when schedules are tight and deadlines are near, developers, temps, and product management members might be assigned to help with testing. At such times, the ratio of testers to developers will increase, and the number of testers may even exceed the number of developers.

During test planning, the projected ratios must be seen as approximations or ideals. The ratios actually implemented will vary based on available budget, buy-in of decision makers to the testing effort, complexity of the application, test effectiveness, and many more factors.

Table 12.1. Test-Team Size Calculated Using the Development-Ratio Method^[a]

Product Type	Number of Developers	Ratio of Developers to Testers	Number of Testers
Commercial Product (Large Market)	30	3:2	20
Commercial Product (Small Market)	30	3:1	10
Development & Heavy COTS Integration for Individual Client	30	4:1	7
Government (Internal) Application Development	30	5:1	6
Corporate (Internal) Application Development	30	4:1	7

^[a] Ratios will differ depending on system complexity or when building systems where there is no tolerance for error, such as medical device systems or air traffic control systems.

Project Staff Ratio Method

Another quick way to estimate the number of personnel required to support a test program is to use the Project Staff Ratio method, as detailed in [Table 12.2](#). This method is based on historical metrics, and uses the number of people planned to support an entire project—including requirements, configuration, process, development, and QA—to calculate the size of the test team. This method may be especially valuable when the number of people performing development work changes frequently or is difficult to quantify. In calculating the test-team sizes for [Table 12.2](#), it is assumed that the scope of the test effort will include requirements reviews, configuration testing, process involvement, and functional and performance testing at the integration and system test levels. A baseline value of 50 in the "Project Staffing Level" column is selected to simplify calculation of the associated values in the "Test Team Size" column. The number of personnel on a project, of course, may be more or fewer.

Table 12.2. Test-Team Size Calculated Using the Project Staff Ratio Method

Development Type	Project Staffing Level	Test Team Size Factor	Number of Testers
Commercial Product (Large Market)	50	27%	13
Commercial Product (Small Market)	50	16%	8
Application Development for Individual Client	50	10%	5
Development & Heavy COTS Integration for Individual Client	50	14%	7
Government (Internal) Application Development	50	11%	5
Corporate (Internal) Application Development	50	14%	7

Test Procedure Method

Another way to estimate the level of effort required to support a test program is to use test program sizing estimates, in the form of the number of test procedures planned for a project. This method is somewhat limited because it focuses solely on the number of test procedures to be developed and executed, ignoring other important factors that affect the scale of the testing effort. To get a more comprehensive metric, this method must be used in combination with one of the other methods listed here.

To employ the Test Procedure Method, the organization must first develop a historical record of the various development projects that have been performed, with associated development sizing values such as function points, number of test procedures used, and the resulting test effort measured in terms of personnel hours. Development sizing values may be documented in terms of lines of code (LOC), lines of code equivalent, function points, or number of objects produced.

The test team then determines requirements and the relationship between historical development sizing values and the numbers of test procedures used in those projects, and calculates an estimate for the number of test procedures required for the new project. Next, the test team determines the historical relationship between

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.

It is also important to take into account the relationship between the number of test procedures applied to a project and the scope of test requirements. Successful employment of this method requires that the requirements and scope of the project have been defined beforehand. Unfortunately, test estimates often are requested, and needed, well before requirements have been finalized.

Task Planning Method

Another way to estimate a test program's required level of effort involves the review of historical records with regard to the number of personnel hours expended to perform testing for a similar type of test effort. This method differs from the Test Procedure Method in that it focuses on testing *tasks*. Both methods require a highly structured environment in which various details are tracked and measured, such as the factors described under "[Other Considerations](#)" later in this Item. Earlier test efforts would need to have included time recording in accordance with a work-breakdown structure, so that historical records document the effort required to perform the various tasks.

In [Table 12.4](#), an example new project estimated to require 1,120 test procedures is compared against a historical baseline. The historical record indicates that an average project involving 860 test procedures required 5,300 personnel hours, a factor of 6.16. This factor is used to estimate the number of hours required to perform 1,120 test procedures. (This is the same historical comparison used for the Test Procedure Method, reflected in [Table 12.3](#).)

Table 12.4. Test-Personnel Hours Calculated Using the Task-Planning Method			
	Number of Test Procedures	Factor	Test-Personnel Hours
Historical Record (Similar Project)	860	6.16	5,300
New Project Estimate	1,120	6.16	6,900

The test team then reviews the historical record to break out the hours expended on the various test program tasks within the test phases. A summary of hours for each phase is depicted in [Table 12.5](#).

Table 12.5. Hours Expended in Each Testing Phase, Task-Planning Method

Phase	Historical Value	% of Project	Preliminary Estimate	Adjusted Estimate
1 Project Startup	140	2.6	179	179
2 Early Project Support (requirements analysis, etc.)	120	2.2	152	152
3 Decision to Automate Testing	90	1.7	117	-
4 Test Tool Selection and Evaluation	160	3	207	-
5 Test Tool Introduction	260	5	345	345
6 Test Planning	530	10	690	690
7 Test Design	540	10	690	690
8 Test Development	1,980	37	2,553	2,553
9 Test Execution	870	17	1,173	1,173
10 Test Management and Support	470	9	621	621
11 Test Process Improvement	140	2.5	173	-
PROJECT TOTAL	5,300	100%	6,900	6,403

To derive these figures, the test team first develops a preliminary estimate of the hours required for each phase, using the historical percentage factor. (If no historical data is available, it may be necessary to base these estimates on testers' past experience.) The right-hand column is for revisions as conditions change. In the example, the new project is mandated to use a particular automated testing tool, making Steps 3 and 4 superfluous. The test team is also advised that the project will not be provided any funding to cover test process improvement activities. The hours for Step 11 are therefore deleted in the revised estimate.

The next step, depicted in [Table 12.6](#), is to compute the test-team size based on the adjusted personnel-hour estimate of 6,403 hours. The test team size is calculated to be equivalent to 3.1 test engineers over the twelve-month project. In the event that the test team is staffed with exactly three full time test personnel throughout the duration of the test effort, the team must achieve a slightly higher level of

productivity than that of previous test teams in order to perform the test effort within the given schedule, if that is possible. More likely, the team must focus initially on only high risk items, while lower risk items will have to be addressed later on. The testing strategy must be adjusted accordingly, and reasons for the adjustment, such as not enough personnel, must be communicated.

The test team could implement a different staffing approach, which applies two full time test personnel plus the fractional time of two other test engineers. The fractional time could be split a number of ways, including 80% of one person and 80% of a second person.

Table 12.6. Test-Team Size Based on Personnel-Hour Estimate

	Number of Test Procedures	Personnel-Hour Estimate	Adjusted Estimate	Performance Period	Number of Testers
New Project Estimate	1,120	5.71	6,403	12 months (2,080 hrs)	3.1

Other Considerations

Regardless of the estimation method selected, experienced test professionals should consider any unusual circumstances that may operate against relying completely on projections from past testing efforts. For example, when using the Development Ratio Method in an organization that has not emphasized testing in the past—and therefore has not yet defined mature testing processes—it may be necessary to adjust the ratios accordingly. Similarly, when using the Task Planning Method, if the test team has recently experienced significant turnover, it may be prudent to adjust the factor derived. Issues to consider when estimating test efforts include the following:

- *Organization.* Test culture or test maturity of the organization.
- *Scope of test requirements.* Tests that must be performed can include functional-requirement testing, server-performance testing, user-interface testing, program module performance testing, program module complexity analysis, program code coverage testing, system load performance testing, boundary testing, security testing, memory-leak testing, response time performance testing, and usability testing, among others.
- *Test-engineer skill level.* Technical skill level of the individuals performing test.

- *Test-tool proficiency.* The use of automated testing introduces a new level of complexity that a project's test team may not have previously experienced. Tools require a learning curve. Test-script programming expertise is required. This may be new to the test team, and the team may lack experience in writing code. Even if the test team has experience with one kind of automated test tool, a different tool may be required on the new project.
- *Business knowledge.* Test-team personnel's familiarity with the application business area, also called "domain knowledge."
- *Test-team organization.* The test-team's organizational type can be a factor, since some team structures are more effective than others.^[3]

^[3] See Dustin et al., *Automated Software Testing*, Chapter 5 for more information on test-team organization types.

- *Scope of test program.* An effective automated test program amounts to a development effort in itself, complete with strategy and goal planning, test-requirement definition, analysis, design, and coding.
- *Start of test effort.* Test planning and activity should begin early in the project. This means that test engineers must be involved in analysis and design review activities in order to prevent analysis and design errors. Early involvement allows the test team to more completely understand requirements and design, to architect the most appropriate test environment, and to generate a more thorough test design. Early involvement not only supports effective test design, which is critical especially when utilizing an automated test tool, but also provides opportunities for early detection of errors and prevents migration of errors from requirement specification to design, and from design into code.
- *Number of incremental software builds planned.* Many industry software professionals have a perception that the use of automated test tools makes the software test effort less significant in terms of personnel hours, or less complex in terms of planning and execution. Savings from the use of automated test tools will take time to accrue. In fact, at the first use of a particular automated test tool by a test team, very little savings may be realized. Savings are realized in subsequent builds of a software application.
- *Process definition.* Using defined (documented) processes improves efficiency in test-engineering operations. Lack of defined processes has the opposite effect, translating to longer learning curves for junior test engineers.

- *High-risk applications.* The scope and breadth of testing on software applications where a software failure poses a risk to human life is greater than for software applications that do not pose such high risks.

Insufficient time for test development and execution may lead to inefficiency in test-engineering operations and require additional test engineering to correct errors or oversights.

While the test effort required to support a software application project depends on a wide variety of variables, several simple, common methods of test effort estimation work well. This is because these simplified estimation methods reflect the standardized distribution of the effects of all these influences over time on the test effort.

The Development Ratio Method, which uses the size of the development team to determine the appropriate size for the test team, is perhaps the most common method of test-effort estimation. The Project Staff Ratio Method, which calculates test-team size based on overall project size, is generally the easiest way to determine the size of the test team. The Test Procedure and Task Planning methods require the maintenance of historical records in order to produce a sizing estimate based on the actual experience of an organization.

Finally, when sizing a test effort, it is important to review the list of test-effort sizing factors in order to consider unusual circumstances.

To assist in future planning, it is often useful to track actual team hours per task on the current project. This data can be quite useful in future test-team sizing efforts. However, the problem with all the methods provided is that rarely are any two software development projects alike. Complexity, developer's experience, technology, and numerous other variables change with most projects.

Chapter 3. The Testing Team

The capabilities of the testing team can greatly affect the success, or failure, of the testing effort. An effective testing team includes a mixture of technical and domain expertise relevant to the software problem at hand. It is not enough for a testing team to be technically proficient with the testing techniques and tools necessary to perform the actual tests. Depending on the complexity of the domain, a test team should also include members who have a detailed understanding of the problem domain. This knowledge enables them to create effective test artifacts and data and to effectively implement test scripts and other test mechanisms.

In addition, the testing team must be properly structured, with defined roles and responsibilities that allow the testers to perform their functions with minimal overlap and without uncertainty regarding which team member should perform which duties. One way to divide testing resources is by specialization in particular application areas and nonfunctional areas. The testing team may also have more role requirements than it has members, which must be considered by the test manager.

As with any team, continual evaluation of the effectiveness of each test team member is important to ensuring a successful test effort. Evaluation of testers is performed by examining a variety of areas, including the types of defects generated, and the number and types of defects missed. It is never good practice to evaluate a test engineer's performance using numbers of defects generated alone, since this metric by itself does not tell the whole story. Many factors must be considered during this type of evaluation, such as complexity of functionality tested, time constraints, test engineer role and responsibilities, experience, and so on. Regularly evaluating team members using valid criteria makes it possible to implement improvements that increase the effectiveness of the overall effort.

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 13.1. Test Program Roles

Test Manager	
Responsibilities	Skills
<ul style="list-style-type: none"> Liaison for interdepartmental interactions: Representative of the testing team Customer interaction, if applicable Recruiting, staff supervision, and staff training Test budgeting and scheduling, including test-effort estimations 	<ul style="list-style-type: none"> Understands testing process or methodology Familiar with test-program concerns including test environment and data management, trouble reporting and resolution, and test design and development Understands manual testing techniques and automated testing best practices
<ul style="list-style-type: none"> Test planning, including development of testing goals and strategy Vendor interaction Test-tool selection and introduction 	<ul style="list-style-type: none"> Understands application business area, application requirements
<ul style="list-style-type: none"> Cohesive integration of test and development activities 	<ul style="list-style-type: none"> Skilled at developing test goals, objectives, and strategy
<ul style="list-style-type: none"> Acquisition of hardware and software for test environment 	<ul style="list-style-type: none"> Familiar with different test tools, defect-tracking tools, and other test-support COTS tools and their use
<ul style="list-style-type: none"> Test environment and test product configuration management. 	<ul style="list-style-type: none"> Good at all planning aspects, including people, facilities, and schedule
<ul style="list-style-type: none"> Test-process definition, training and continual improvement 	
<ul style="list-style-type: none"> Use of metrics to support continual test-process improvement 	
<ul style="list-style-type: none"> Test-program oversight and progress tracking 	
<ul style="list-style-type: none"> Coordinating pre- and post-test meetings 	
Test Lead	

Responsibilities	Skills
<ul style="list-style-type: none"> Technical leadership for the test program, including test approach 	<ul style="list-style-type: none"> Understands application business area and application requirements
<ul style="list-style-type: none"> Support for customer interface, recruiting, test-tool introduction, test planning, staff supervision, and cost and progress status reporting 	<ul style="list-style-type: none"> Familiar with test-program concerns including test-data management, trouble reporting and resolution, test design, and test development
<ul style="list-style-type: none"> Verifying the quality of the requirements, including testability, requirement definition, test design, test-script and test-data development, test automation, test-environment configuration; test-script configuration management, and test execution 	<ul style="list-style-type: none"> Expertise in a variety of technical skills including programming languages, database technologies, and computer operating systems
<ul style="list-style-type: none"> Interaction with test-tool vendor to identify best ways to leverage test tool on the project 	<ul style="list-style-type: none"> Familiar with different test tools, defect-tracking tools, and other COTS tools supporting the testing life cycle, and their use
<ul style="list-style-type: none"> Staying current on latest test approaches and tools, and transferring this knowledge to test team 	
<ul style="list-style-type: none"> Conducting test-design and test-procedure walk-throughs and inspections 	
<ul style="list-style-type: none"> Implementing test-process improvements resulting from lessons learned and benefits surveys 	
<ul style="list-style-type: none"> Test Traceability Matrix (tracing the test procedures to the test requirements) 	
<ul style="list-style-type: none"> Test-process implementation 	
<ul style="list-style-type: none"> Ensuring that test-product documentation is complete 	
Usability^[2] Test Engineer	

Responsibilities	Skills
<ul style="list-style-type: none"> Designs and develops usability testing scenarios Administers usability testing process 	<ul style="list-style-type: none"> Proficient in designing test suites Understanding usability issues
<ul style="list-style-type: none"> Defines criteria for performing usability testing, analyzes results of testing sessions, presents results to development team 	<ul style="list-style-type: none"> Skilled in test facilitation
<ul style="list-style-type: none"> Develops test-product documentation and reports 	<ul style="list-style-type: none"> Excellent interpersonal skills
<ul style="list-style-type: none"> Defines usability requirements, and interacts with customer to refine them 	<ul style="list-style-type: none"> Proficient in GUI design standards
<ul style="list-style-type: none"> Participates in test-procedure walk-throughs 	

Manual Test Engineer

Responsibilities	Skills
<ul style="list-style-type: none"> Designs and develops test procedures and cases, with associated test data, based upon functional and nonfunctional requirements 	<ul style="list-style-type: none"> Has good understanding of GUI design
<ul style="list-style-type: none"> Manually executes the test procedures 	<ul style="list-style-type: none"> Proficient in software testing
<ul style="list-style-type: none"> Attends test-procedure walk-throughs 	<ul style="list-style-type: none"> Proficient in designing test suites
<ul style="list-style-type: none"> Conducts tests and prepares reports on test progress and regression 	<ul style="list-style-type: none"> Proficient in the business area of application under test Proficient in testing techniques Understands various testing phases
<ul style="list-style-type: none"> Follows test standards 	<ul style="list-style-type: none"> Proficient in GUI design standards

Automated Test Engineer (Automater/Developer)

Responsibilities	Skills
<ul style="list-style-type: none"> Designs and develops test procedures and cases based upon requirements 	<ul style="list-style-type: none"> Good understanding of GUI design
<ul style="list-style-type: none"> Designs, develops and executes reusable and maintainable automated scripts 	<ul style="list-style-type: none"> Proficient in software testing

<ul style="list-style-type: none"> • Uses capture/playback tools for GUI automation and/or develops test harnesses using a development or scripting language, as applicable 	
<ul style="list-style-type: none"> • Follows test-design standards 	<ul style="list-style-type: none"> • Proficient in designing test suites
<ul style="list-style-type: none"> • Conducts/Attends test procedure walk-throughs 	<ul style="list-style-type: none"> • Proficient in working with test tools
<ul style="list-style-type: none"> • Executes tests and prepares reports on test progress and regression 	<ul style="list-style-type: none"> • Programming skills
<ul style="list-style-type: none"> • Attends test-tool user groups and related activities to remain abreast of test-tool capabilities 	<ul style="list-style-type: none"> • Proficient in GUI design standards

Network Test Engineer

Responsibilities	Skills
<ul style="list-style-type: none"> • Performs network, database, and middleware testing 	<ul style="list-style-type: none"> • Network, database and system administration skills
<ul style="list-style-type: none"> • Researches network, database, and middle ware performance monitoring tools • Develops load and stress test designs, cases, and procedures • Supports walk-throughs or inspections of load and stress test procedures 	<ul style="list-style-type: none"> • Expertise in a variety of technical skills, including programming languages, data base technologies, and computer operating systems
<ul style="list-style-type: none"> • Implements performance monitoring tools on ongoing basis • Conducts load and stress testing 	<ul style="list-style-type: none"> • Product evaluation and integration skills • Familiarity with network sniffers, and available tools for load and stress testing

Test Environment Specialist

Responsibilities	Skills
<ul style="list-style-type: none"> • Responsible for installing test tool and establishing test-tool environment 	<ul style="list-style-type: none"> • Network, database and system administration skills
<ul style="list-style-type: none"> • Responsible for creating and controlling test environment by using environment setup scripts 	<ul style="list-style-type: none"> • Expertise in a variety of technical skills, including programming and scripting

	languages, database technologies, and computer operating systems
<ul style="list-style-type: none"> Creates and maintains test database (adds/restores/deletes, etc) 	<ul style="list-style-type: none"> Test tool and database experience
<ul style="list-style-type: none"> Maintains requirements hierarchy within test-tool environment 	<ul style="list-style-type: none"> Product evaluation and integration skills
Security Test Engineer	
Responsibilities	Skills
<ul style="list-style-type: none"> Responsible for security testing of the application 	<ul style="list-style-type: none"> Understands security testing techniques Background in security Security test tool experience
Test Library and Configuration Specialist^[a]	
Responsibilities	Skills
<ul style="list-style-type: none"> Test-script change management 	<ul style="list-style-type: none"> Network, database, and system administration skills
<ul style="list-style-type: none"> Test-script version control 	<ul style="list-style-type: none"> Expertise in a variety of technical skills including programming languages, database technologies, and computer operating systems
<ul style="list-style-type: none"> Maintaining test-script reuse library Creating the various test builds, in some cases 	<ul style="list-style-type: none"> Configuration-management tool expertise
	<ul style="list-style-type: none"> Test-tool experience

^[2] The term **usability** refers to the intuitiveness and ease-of-use of an application's user interface.

^[a] This type of configuration is often done by a separate configuration management department.

It is important that the appropriate roles be assigned to the right people, based on their skill sets and other strengths. Individual testers can specialize in specific areas

of the application, as well as nonfunctional areas. Assigning testers to specific areas of the application may enable them to become experts in those specific areas. In addition to focusing on particular functional testing areas, different team members should specialize in specific nonfunctional testing areas, performance testing, compatibility, security, and concurrency.

Teams that use automated test tools should include personnel with software-development skills. Automated testing requires that test scripts be developed, executed, and managed. For this reason, the skills and activities pertinent to performing manual testing differ from those required for performing automated software testing. Because of this difference, manual test roles should be listed separately in the definition of roles and responsibilities. This does not mean that an "automated test engineer" won't be expected to do some manual testing from time to time.

If multiple products are under test, testers can maximize continuity by keeping specific team members assigned to the same product domains for some period of time. It can be difficult for testers to switch between significantly different technical or problem domains too often. Testers should be kept on either the same type of products (e.g., Web vs. desktop applications) or in the same problem areas, depending on where their strengths lie. The most effective testers understand the business and system requirements, in addition to being technically adept.

[Table 13.2](#) gives an example of a test-team structure. Basic skills are assumed and not listed. For example, if working in a Windows environment, it is assumed that the team members have Windows skills; if working in a UNIX environment, it is assumed the team members have basic UNIX skills.

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

fulfilling those positions. Also noted are the products to which each position on the team is assigned.

Item 1 of this book emphasized the importance of the testing team's involvement from the beginning of the product's life cycle. If early involvement of testers has become an established practice in an organization, it is possible (and necessary) to define and document the role of the testing team during each phase of the life cycle, including the deliverables expected from the testing team upon completion of each phase.

Item 14: Require a Mixture of Testing Skills, Subject-Matter Expertise, and Experience

The most effective testing team consists of team members with a mixture of expertise, such as subject matter, technology, and testing techniques, plus a mixture of experience levels, such as beginners and expert testers. Subject-matter experts (SMEs) who understand the details of the application's functionality play an important role in the testing team.

The following list describes these concepts in more detail.

- *Subject matter expertise.* A technical tester might think it is feasible to learn the subject matter in depth, but this is usually not the case when the domain is complex. Some problem domains, such as tax law, labor contracts, and physics, may take years to fully understand. It could be argued that detailed and specific requirements should include all the complexities possible, so that the developer can properly design the system and the tester can properly plan the testing. Realistically, however, budget and time constraints often lead to requirements that are insufficiently detailed, leaving the content open to interpretation. Even detailed requirements often contain internal inconsistencies that must be identified and resolved.

For these reasons, each SME must work closely with the developer and other SMEs on the program (for example, tax-law experts, labor-contracts experts, physicists) to parse out the intricacies of the requirements. Where there are two SMEs, they must be in agreement. If two SMEs cannot agree, a third SME's input is required. A testing SME will put the final stamp of approval on the implementation, after appropriate testing.

- *Technical expertise.* While it is true that a thorough grasp of the domain is a valuable and desirable trait for a tester, the tester's effectiveness will be diminished without some level of understanding of software (including test) engineering. The most effective subject-matter expert testers are those who are also interested and experienced in the technology—that is, those who have taken one or more programming courses, or have some related technical experience. Subject-matter knowledge must be complemented with technical knowledge, including an understanding of the science of software testing.

Technical testers, however, require a deeper knowledge of the technical platforms and architectural makeup of a system in order to test successfully. A technical tester should know how to write automated scripts; know how to write a test harness; and understand such technical issues as compatibility, performance, and installation, in order to be best prepared to test for compliance. While it is beneficial for SMEs to possess some of this knowledge, it is acceptable, of course, for them to possess a lower level of technical expertise than do technical testers.

- *Experience level.* A testing team is rarely made up exclusively of expert testers with years of expertise—nor would that necessarily be desirable. As with all efforts, there is room for apprentices who can be trained and mentored by more-senior personnel. To identify potential areas for training and advancement, the test manager must review the difference between the skill requirements and an individual's actual skills.

A junior tester could be tasked with testing the lower-risk functionality, or cosmetic features such as the GUI interface controls (if this area is considered low-risk). If a junior tester is tasked with testing of higher-risk functionality, the junior tester should be paired with a more-senior tester who can serve as a mentor.

Although technical and subject-matter testers contribute to the testing effort in different ways, collaboration between the two types of testers should be encouraged. Just as it would take a technical tester a long time to get up to speed on all the details of the subject matter, it would take a domain or subject-matter expert a long time to become conversant with the technical issues to consider during testing. Cross-training should be provided to make the technical tester acquainted with the subject matter and the subject-matter expert familiar with the technical issues.

Some testing tasks may require specific skills within the technical or subject-matter area. For example, a tester who is experienced, or at least familiar, with usability testing techniques should be responsible for usability testing. A tester who is not skilled in this area can only guess what makes an application usable. Similarly, in the case of localization testing, an English speaker can only guess regarding the correct translation of a Web site into another language. A more effective localization tester would be a native speaker of the language into which the site has been translated.

Item 15: Evaluate the Tester's Effectiveness^[1]

^[1] Adapted from Elfriede Dustin, "Evaluating a Tester's Effectiveness," [Stickyminds.com](http://www.stickyminds.com) (Mar. 11, 2002). See <http://www.effectivesoftwaretesting.com>.

Maintaining an effective test program requires that the implementation of its elements, such as test strategy, test environment, and test-team make-up, be continuously evaluated, and improved as needed. Test managers are responsible for ensuring that the testing program is being implemented as planned and that specific tasks are being executed as expected. To accomplish this, they must track, monitor, and evaluate the implementation of the test program, so it can be modified as needed.

At the core of test-program execution are the test engineers. The ability of testers to properly design, document, and execute effective tests, accurately interpret the results, document any defects, and track them to closure is critical to the effectiveness of the testing effort. A test manager may plan the perfect testing process and select the ideal strategy, but if the test-team members do not effectively execute the testing process (for example, participating effectively in requirements inspections and design walk-throughs) and complete all strategic testing tasks as assigned (such as executing specific test procedures), important defects may be discovered too late in the development life cycle, resulting in increased costs. Worse, defects may be completely overlooked, and make their way into production software.

A tester's effectiveness can also make a big difference in relationships with other project groups. A tester who frequently finds bogus errors, or reports "user errors" when the application works as expected but the tester misunderstands the requirement, or (worst of all) often overlooks critical defects loses credibility with

other team members and groups, and can tarnish the reputation of an entire test program.

Evaluating a tester's effectiveness is a difficult and often subjective task. Besides the typical elements in any employee's performance, such as attendance, attentiveness, attitude, and motivation, there are specific testing-related measures against which a tester can be evaluated. For example, all testers must be detail oriented and possess analytical skills, independent of whether they are technical testers, subject-matter experts, security testers, or usability testers.

The evaluation process starts with recruitment. The first step is to hire a tester with the skills required for the roles and responsibilities assigned to each position. (See [Item 13](#) for a discussion on roles, responsibilities, and skills.)

In the case where a testing team is "inherited" rather than hired for the project, evaluation is more complicated. In such a case it is necessary for the manager to become familiar with the various testers' backgrounds, so the team members can be tasked and evaluated based on their experience, expertise, and backgrounds. It may become necessary to reassign some team members to other roles as their abilities become better known.

A test engineer's performance cannot be evaluated unless there are specified roles and responsibilities, tasks, schedules, and standards. The test manager must, first and foremost, state clearly what is expected of the test engineer, and by when.

Following is a typical list of expectations that must be communicated to testers.

- *Observe standards and procedures.* The test engineer must be aware of standards and procedures to be followed, and processes must be communicated. Standards and procedures are discussed in [Item 21](#).
- *Keep schedules.* Testers must be aware of the test schedule, including when test plans, test designs, test procedures, scripts, and other testing products must be delivered. In addition, the delivery schedule of software components to testing should be known by all testers.
- *Meet goals and perform assigned tasks.* Tasks must be documented and communicated, and deadlines must be scheduled, for each tester. The test manager and the test engineer must agree on the assigned tasks.
- *Meet budgets.* For testers evaluating testing tools or other technology that must be purchased, the available budget must be communicated so the tester can work within that range and avoid wasting time evaluating products that are too expensive.

Expectations and assignments differ depending on the task at hand and the skill set of the tester. Different types of tests, test approaches, techniques, and outcomes may be expected.

Once expectations are set, the test manager can start comparing the work of the test team against the established goals, tasks, and schedules to measure effectiveness of implementation. Following is a list of points to consider when evaluating a tester's effectiveness.

- *Subject-matter expert vs. technical expert.* The expertise expected from a subject-matter expert is related to the domain of the application, while a technical tester is concerned with the technical issues of the application.

When a technical tester functions as an automater, automated test procedures should be evaluated based on defined standards that must be followed by the test engineers. For example, the supervisor might ask: Did the engineer create maintainable, modular, reusable automated scripts, or do the scripts have to be modified with each new system build? Did the tester follow best practices, such as making sure the test database was baselined and could be restored when the automated scripts need to be rerun? If the tester is developing custom test scripts or a test harness, the tester will be evaluated on some of the same criteria as a developer, including readability and reliability of the code.

A tester who specializes in the use of automated tools, yet does not understand the intricacies of the application's functionality and underlying concepts, will usually be ineffective. Automated scripts based only on high-level knowledge of the application will often find less-important defects. It is important that the automater understand the application's functionality in order to be an effective member of the testing team.

Another area for evaluation is technical ability and adaptability. Is the test engineer capable of picking up new tools and becoming familiar with their capabilities? Testers should be trained regarding the various capabilities of a testing tool, if they are not already thoroughly familiar with them.

- *Experienced vs. novice tester.* As previously mentioned, the skill level of the tester must be taken into account. For example, novice testers may overlook some errors, or not realize they are defects. It is important to assign novice testers to lower-risk testing areas.

Inexperienced testers are not alone in overlooking defects. Experienced testers may ignore some classes of defects based on past experience ("the product has always done that") or the presence of work-arounds.

Appropriately or not, testers may become "acclimated" to familiar errors, and may not report defects that seem unimportant to them but may be unacceptable to end users.

- *Functional vs. nonfunctional testing.* A tester's understanding of the various testing techniques available (see [Chapter 5](#)) and knowledge of which technique is most effective for the task at hand should be evaluated. If the tester doesn't understand the various techniques and applies a technique inappropriately, test designs, test cases, and test procedures will be adversely affected.

Functional testing can additionally be based on a review of the test procedures. Typically, testers are assigned to test procedures for testing specific areas of functionality based on assigned requirements. Test procedure walk-throughs and inspections should be conducted that include the requirements, testing, and development teams. During the walk-through, it should be verified that all teams agree on the behavior of the application.

The following questions should be considered during an evaluation of functional test procedures:

- How completely are the test-procedure steps mapped to the requirements steps? Is traceability complete?
- Are the test input, steps, and output (expected result) correct?
- Are major testing steps omitted in the functional flow of the test procedure?
- Has an analytical thought process been applied to produce effective test scenarios?
- Have the test-procedure creation standards been followed?
- How many revisions have been required as a result of misunderstanding or miscommunication before the test procedures could be considered effective and complete?
- Have effective testing techniques been used to derive the appropriate set of test cases?

During a test-procedure walk-through, the "depth" or thoroughness of the test procedure should be verified. In other words, what does the test

procedure test? Does it test functionality only at a high level, or does it really dig deep down into the underlying functionality of the application?

To some extent, this is related to the depth of the requirement steps. For example, a functional requirement might state, "The system should allow for adding records of type A." A high-level test procedure establishes that the record can be added through the GUI. A more-effective test procedure also includes steps that test the areas of the application affected when this record is added. For instance, a SQL statement might verify that the record appears correctly in the database tables. Additional steps could verify the record type. There are numerous other testing steps to be considered, such as verifying the system's behavior when adding multiple records of type A—whether duplicates are allowed, for example.

If test procedures are at a very high level, it is important to confirm that the requirements are at the appropriate level and pertinent details are not missing. If there is a detail in the requirement that is missing in the test procedure, the test engineer might need coaching on how to write effective test procedures. Or, it could be that the engineer did not adequately understand the requirement.

Different criteria apply to evaluating functional testing than to nonfunctional testing. For example, nonfunctional tests must be designed and documented in a different manner than functional test procedures.

- *Testing phase.* Different tasks are to be performed by the tester depending on the testing phase (alpha test, beta test, system test, acceptance test, and so on).

During system testing, the tester is responsible for all testing tasks described in this book, including the development and execution of test procedures, tracking defects to closure, and so on. Other testing phases may be less comprehensive.

During alpha testing, for example, a tester might be tasked with simply recreating and documenting defects reported by members of a separate "alpha testing team," which is usually the company's independent testing (Independent Verification and Validation, or IV&V) team.

During beta testing, a tester might be tasked with documenting the beta-test procedures to be executed, in addition to recreating and documenting defects

found by other beta testers. (Customers are often recruited to become beta testers.)

- *Phase of the development life cycle.* As mentioned throughout this book, testers should be involved from the beginning of the life cycle. Evaluation of tester performance should be appropriate to each phase. For example, during the requirements phase, the tester can be evaluated based on defect-prevention efforts, such as identification of testability issues or requirements inconsistencies.

While a tester's evaluation can be subjective, many variables related to the phase of testing must be considered, rather than jumping to the first seemingly obvious conclusion. For example, when evaluating the test engineer during the requirements phase, it is important to consider the quality of the requirements themselves. If the requirements are poorly written, even an average tester can find many defects. However, if the requirements are well laid out and their quality is above average, only an exceptional tester is likely to find the most subtle defects.

- *Following of instructions and attention to detail.* It is important to consider how well a test engineer follows instructions and pays attention to detail. Reliability and follow-through must be monitored. If test procedures must be updated and executed to ensure a quality product, the test manager must be confident that the test engineers will carry out this task. If tests have to be automated, the test manager should be confident that progress is being made.

Weekly status meetings where engineers report on their progress are useful to track and measure progress. In the final stages of a testing phase, these meetings may be held daily.

- *Types of defects, defect ratio, and defect documentation.* The types of defects found by the engineer must be considered during the evaluation. When using this metric to evaluate a tester's effectiveness, some factors to keep in mind include the skill level of the tester, the types of tests being performed, the testing phase being conducted, and the complexity and the maturity of the application under test. Finding defects depends not only upon the skill of the tester, but also on the skill of the developer who wrote, debugged, and unit tested the code, and on the walk-through and inspection teams that reviewed the requirements, design, and code. Ideally, they will have corrected most defects before formal testing.

An additional factor to evaluate in this context is whether the test engineer finds errors that are complex and domain related, or only cosmetic. Cosmetic defects, such as missing window text or control placement, are relatively easy to detect and become high priority during usability testing, whereas more complicated problems relating to data or cause-effect relationships between elements in the application are more difficult to detect, require a better understanding of the application, and become high priority during functional testing. On the other hand, cosmetic-defect fixes, since they are most visible, may have a more immediate effect on customer happiness.

The test manager must consider the area for which the tester is responsible. The tester responsible for a specific area where the most defects are discovered in production should not necessarily be assumed to have performed poorly. If the tester's area is very complex and error-prone and the product was released in a hurry, failure to catch some defects may be understandable.

The *types* of defects discovered in production also matter. If they could have been discovered by a basic test within the existing test-procedure suite, and if there was plenty of time to execute the test procedure, this would be a major oversight by the tester responsible for this area. However, before passing judgment, some additional questions should be considered:

- Was the test procedure supposed to be executed manually? The manual tester may have become tired of executing the same test procedures over and over, and after many trials concluded it should be safe not to execute the tests because that part of the application has always worked in the past.
- Was the software delivered under pressure of a deadline that could not be changed even though it ruled out a full test cycle? Releases should not be allowed without having met the release criteria, time pressures notwithstanding.
- Was this test automated? Did the automated script miss testing the step containing the error? In such a case, the automated scripts must be reevaluated.
- Was the defect discovered using some combination of functional steps that are rarely executed? This type of defect is more understandable.

Additionally, it may be necessary to review the test goals, risks of the project, and assumptions made when the test effort started. If it had been

decided not to conduct a specific type of test because of time constraints or low risk, then the tester should not be held responsible. This risk should have been taken with full knowledge of the possibility of problems.

Effectiveness can also be evaluated by examining how a defect is documented. Is there enough detail in the documented defect for a developer to be able to recreate the problem, or do developers have a difficult time recreating one specific tester's defects? Standards must be in place that document precisely what information is required in defect documentation, and the defect tracking life cycle must be well communicated and understood. All testers must follow these standards. (For a discussion of the defect tracking life cycle, see [Item 50](#).)

For each issue uncovered during evaluation of a tester, the cause of the issue should be determined and a solution should be sought. Each issue must be evaluated with care before a judgment regarding the tester's capability is made. After careful evaluation of the entire situation, and after additional coaching has been provided where called for, it will be possible to evaluate how detail oriented, analytical, and effective this tester is. If it is determined that the tester lacks attention to detail or analytical skills or there are communication issues, that tester's performance may need to be closely monitored and reviewed, and there may be a need for additional instruction and training, or other appropriate steps need to be taken.

Testers' effectiveness must be constantly evaluated to ensure the success of the testing program.

Test Engineer Self-Evaluation

Test engineers should assume responsibility for evaluating their own effectiveness. The following list of issues can be used as a starting-point in developing a process for test-engineer self-evaluation, assuming roles and responsibilities along with task assignments are understood:

- Consider the types of defects being discovered. Are they important, or are they mostly cosmetic, low-priority defects? If the tester consistently uncovers only low-priority defects—such as, during functional testing, non-working hot keys, or typographical errors in the GUI—the effectiveness of the test procedures should be reassessed. Keep in mind that during other

testing phases (for the examples mentioned here, during usability testing), the priority of certain defects will change.

- Are test procedures detailed enough, covering the depth, and combinations and variations of data and functional paths, necessary to catch the higher-priority defects? Do tests include invalid data as well as valid data?
- Was feedback regarding test procedures received and incorporated from requirements and development staff, and from other testers? If not, the test engineer should ask for test-procedure reviews, inspections, and walkthroughs involving those teams.
- Does the test engineer understand the range of testing techniques available, such as boundary-values testing, equivalence partitioning, and orthogonal arrays, well enough to select the most effective test procedures?
- Does the engineer understand the intricacies of the application's functionality and domain well? If not, the tester should ask for an overview or additional training. A technical tester may ask for help from a Subject-Matter Expert (SME).
- Are major defects being discovered too late in the testing cycle? If this occurs regularly, the following points should be considered:

Does the initial testing focus on low-priority requirements? Initial testing should focus on the high-priority, highest-risk requirements.

Does the initial testing focus on regression testing of existing functionality that was working previously and rarely broke in the past? Initial testing should focus on code changes, defect fixes, and new functionality. Regression testing should come later. Ideally, the regression-testing efforts can be automated, so test engineers can focus on the newer areas.

- Are any areas under testing exhibiting suspiciously low defect counts? If so, these areas should be re-evaluated to determine:

Whether the test coverage is sufficiently robust.

Whether the types of tests being performed are most effective. Are important steps missing?

Whether the application area under test has low complexity, so that indeed it may be error-free.

Whether the functionality was implemented in such a manner that it is likely no major defects remain—for example, if it was coded by the most senior developers and has already been unit and integration tested well.

Consider the Defect Workflow:

- Each defect should be documented in a timely manner (i.e., as soon as it is discovered and verified)
- Defect-documentation standards must be followed. If there aren't any defect-documentation standards, they should be requested from the engineer's manager. The standards should list all information that must be included in documenting a defect to enable the developer to reproduce it.
- If a new build is received, initial testing should focus on retesting the defects. It is important that supposedly fixed defects be retested as soon as possible, so the developers know whether their repair efforts are successful.
- Comments received from the development team regarding the quality of defect reports should be continually evaluated. If the reports are often said to lack required information, such as a full description of testing steps required to reproduce the errors, the testers should work on providing better defect documentation.
- Testers should be eager to track defects to closure.
- Examine the comments added to defect documentation to determine how developers or other testers receive it. If defects are often marked "works as expected" or "cannot be reproduced," it could also signal some problems:
 - The tester's understanding of the application may be inadequate. In this case, more training is required. Help may be requested from domain SMEs.
 - The requirements may be ambiguous. If so, they must be clarified. (Most commonly, this is discovered during the requirements or test-procedure walk-through and inspections.)
 - The engineer's documentation skills may not be as effective as necessary. Inadequate documentation may lead to misunderstanding of the identified defect. The description may need additional steps to enable developers to reproduce the error.
 - The developer may be misinterpreting the requirement.
 - The developer may lack the patience to follow the detailed documented defect steps to reproduce the defect.

- The tester should monitor whether defects are being discovered in that person's test area after the application has gone to production. Any such defects should be evaluated to determine why they were missed:
 - Did the tester fail to execute a specific test procedure that would have caught this defect? If so, why was the procedure overlooked? Are regression tests automated?
 - Was there no test procedure that would have caught this defect? If so, why not? Was this area considered low risk? The test procedure creation strategy should be reevaluated, and a test procedure should be added to the regression test suite to catch errors like the one in question. The tester should discuss with peers or a manager how to create more effective test procedures, including test design, strategy, and technique.
 - Was there not enough time to execute an existing test procedure? If so, management should be informed—before the application goes live or is shipped, not after the fact. This sort of situation should also be discussed in a post-test/pre-installation meeting, and should be documented in the test report.
- Do other testers during the course of their work discover defects that were this tester's responsibility? If so, the tester should evaluate the reasons and make adjustments accordingly.

There are many more questions a tester can ask related to testing effectiveness, depending on the testing phase and testing task at hand, type of expertise (technical vs. domain), and tester's experience level.

An automater might want to be sure to become familiar with automation standards and best automation practices. A performance tester might request additional training in the performance-testing tool used and performance testing techniques available.

Self-assessment of the tester's capabilities and the improvement steps that follow are important parts of an effective testing program.

Chapter 4. The System Architecture

Proper testing of an application requires more than simply verifying the simulated or re-created user actions. Testing the system through the user interface only, without understanding the system's internal structure and components, is typically referred to as **black-box testing**. By itself, black-box testing is not the most effective way to test. In order to design and implement the most effective strategy for thoroughly investigating the correct functioning of an application, the testing team must have a certain degree of knowledge of the system's internals, such as its major architectural components. Such knowledge enables the testing team to design better tests and perform more effective defect diagnosis. Testing a system or application by directly targeting the various modules and layers of the system is referred to as **gray-box testing**.

Understanding the components and modules that make up an entire system enables the testing team to narrow down its effort and focus on the specific area or layer where a defect is present, increasing the efficiency of the defect-correction activities of the development staff. A black-box tester is limited to reporting on the effect or symptom of a defect, since this tester must rely on the error messages or other information displayed by the interface, such as "the report cannot be generated." A black-box tester also has a more difficult time identifying false positives and false negatives. A gray-box tester, on the other hand, not only sees the error message through the user interface but also has the tools to diagnose the problem and can report on the source of the defect. Understanding the system architecture also allows focused testing to be performed, targeted at architecturally sensitive areas of the application such as the database server or core calculation modules.

Just as it is important for the testing team to be involved during the requirement-writing process, as discussed in [Chapter 1](#), so too must the testing team review the architecture of the application. This allows the team to identify potential testability issues early in the project's life cycle. For example, if an application's architecture makes heavy use of third-party products, this may make the system difficult to test and diagnose, since the organization does not control the source code for these components and cannot modify them. The testing team must identify these types of issues early on to allow for development of an effective testing strategy that takes them into consideration. Overly complex architectures, such as those that make use of many loosely connected off-the-shelf products, can also result in systems whose defects cannot readily be isolated or reproduced. Again, the testing team needs to detect these issues early, to allow for better planning.

The system itself, if implemented correctly, can make for an easier testing process in many ways. Logging or **tracing** mechanisms can be extremely useful in tracking application behavior during development and testing. In addition, different operating **modes**, such as debug and release modes, can be useful in detecting and diagnosing problems with the application even after it has gone live.

Item 16: Understand the Architecture and Underlying Components

An understanding of the architecture and underlying components of an application allows the test engineer to help pinpoint the various areas of the application that produce particular test outcomes. Such an understanding allows the tester to conduct gray-box testing, which can complement the black-box testing approach. During gray-box testing, the tester can identify specific parts of the application that are failing. For example, the test engineer is able to probe areas of the system that are more prone to failure because of their complexity, or simply due to the instability of "fresh" code.

Following are some examples of how a thorough understanding of the system architecture can assist a test engineer:

- *Enhanced defect reporting.* For the most part, test procedures are based on requirements, and therefore have a somewhat fixed path through the system. When an error occurs along this path, the ability of the tester to include information relevant to the system architecture in the defect report can be of great benefit to the system's development staff. For example, if a certain dialog box fails to display, the tester's investigation could determine that it was due to a problem retrieving information from the database, or that the application failed to make a connection to a server.
- *Improved ability to perform exploratory testing.* Once a test has failed, the tester usually must perform some focused testing, perhaps by modifying the original test scenario to determine the application's "breaking point," the factors that cause the system to break. During this exercise, architectural knowledge of the system under test can be of great help to the tester, enabling the test engineer to perform more useful and specific testing—or perhaps to skip additional testing altogether, when knowledge of the underlying components provides adequate information about the problem. For example, if it is known that the application has encountered a connection problem with the database, it is not necessary to attempt the operation with different data values. Instead, the tester can focus on the connection issues.

- *Increased testing precision.* Gray-box testing is designed to exercise the application, either through the user interface or directly against the underlying components, while monitoring internal component behavior to determine the success or failure of the test. Gray-box testing thus naturally produces information relevant to the cause of the defect.

Here are the most common types of problems that can be encountered during testing:

- A component encounters a failure of some kind, causing the operation to be aborted. The user interface typically indicates that an error has occurred.
- The test execution produces incorrect results, different from the expected outcome. Somewhere in the system, a component has processed data incorrectly, causing the erroneous results.
- A component fails during execution, but does not notify the user interface that an error has occurred, which is known as a false positive. For example, data is entered but is not stored in the database, yet no error is reported to the user.
- The system reports an error, but it actually has processed everything correctly—the test produces a false negative.

In the first case—an error leads to aborting the operation—it is important to display useful and descriptive error messages, but this often does not occur. For example, if a database error occurs during an operation, the typical user interface displays a cryptic message like "failed to complete operation," without any details as to why. A much more useful error message would give more information, such as, "failed to complete operation due to a database error." Internally, the application may also have an error log with even more information. Knowledge of the system components allows the tester to use all available tools, including log files and other monitoring mechanisms, to more precisely test the system, rather than depending entirely on user-interface messages.

There are several ways that the testing team can gain an understanding of the architecture. Perhaps the best way is for the team to participate in architecture and design reviews where the development staff presents the proposed architecture. In addition, testers should be encouraged to review architecture and design documentation and to ask questions of the developers. It is also important that the

testing team review any changes to the architecture after each release, so that any impact on the testing effort can be assessed.

Item 17: Verify That the System Supports Testability

Most large systems are made up of many subsystems, which in turn consist of code residing in one or more layers and other supporting components, such as databases and message queues. Users interact with the boundaries, or user interface, of the system, which then interacts with other subsystems to complete a task. The more subsystems, layers, and components there are in a system, the more difficult it may be to isolate a problem while testing that system.

Consider the following example. The user interface takes input from the user, and, using the services of various layers of code in the application, eventually writes that input to a database. Later, another subsystem, such as a reporting system, reads the data and performs some additional processing to produce a report. If something goes wrong in any part of this process, perhaps due to a problem with the user's data or a concurrency problem, the location of the defect can be difficult to isolate, and the error may be difficult to recreate.

When the architecture of an application is first being conceptualized, the testers should have an opportunity to ask questions about how they can follow input through a path within the system. For example, if a certain function causes a task to be launched on another server, it is useful for the tester to have a way to verify that the remote task was indeed launched as required. If the proposed architecture makes it difficult to track this kind of interaction, then it may be necessary to reconsider the approach, perhaps using a more dependable, and testable, architecture. The testing strategy must account for these types of issues, and may require an integrated test effort in some cases with staff from other development efforts, including third-party component vendors.

The testing team must consider all aspects of the proposed architecture, and how they may or may not contribute to effective and efficient testing of the application. For example, although third-party components, such as off-the-shelf user-interface controls, can reduce development time by obviating large sections of work on architecturally significant components, their use may have negative implications for testability. Unless the source code is available—and can be modified—it may be difficult to follow paths through third-party components, which may not provide tracing or other diagnostic information. If using a third-party component is proposed for the application's architecture, it is important to prototype the

implementation and verify that there are ways to monitor the flow of control through that component. Third-party components may also prevent the use of some testing tools, which in turn can adversely affect the testing effort.

Investigating the testability of an application's architecture while it exists only on paper can greatly reduce the number of testing surprises that may be encountered later on. If the testing implications of one or more aspects of the architecture are unclear, the test team should insist on a prototype that allows the testers to experiment with various testing techniques. Feedback from this exercise can ensure that the application is developed in such a way that its quality can be verified.

Item 18: Use Logging to Increase System Testability

One of the most common ways to increase the testability of an application is to implement a logging, or tracing, mechanism that provides information about what components are doing, including the data upon which they are operating, and about application states or errors encountered while the application is running. Test engineers can use such information to track the processing flow during the execution of a test procedure and to determine where errors are occurring in the system.

As the application executes, all components write log entries detailing what **methods** (also known as **functions**) they are currently executing and the major objects they are manipulating. Entries are typically written to a disk file or database, properly formatted for analysis or debugging at some point in the future, after the execution of one or more test procedures. In a complex client-server or Web system, log files may be written on several machines, so it is important that the logs include enough information to determine the path of execution between machines.

The tracing mechanism must write enough information to the log to be useful for analysis and debugging, but not so much information that it creates an overwhelming and unhelpful volume of information, making it difficult to isolate important entries.

A log **entry** is simply a formatted message that contains key information that can be used during analysis. A well-formed log entry includes the following pieces of information:

- *Class name and method name.* This can be a function name if the function is not a member of any class. This information is important for determining a path of execution through several components.
- *Host name and process ID.* This will allow log entries to be compared and tracked if they are generated from events on different machines or from different processes on the same machine.
- *Time-stamp of the entry (to the millisecond or better).* An accurate time-stamp on all entries allows the events to be correlated, if they occur in parallel or on different machines, which may cause them to be entered in the log out of sequence.
- *Messages.* One of the most important pieces of the entry is the message. It is a description, written by the developer, of what was happening at the time in the application. A message can also be a description of an error encountered during execution, or a result code from an operation. Other types of messages include the logging of persistent entity IDs or keys of major domain objects. This allows tracking of objects through the system during execution of a test procedure.

By reviewing the items written to the log file by every method or function of every component in the system, the execution of a test procedure can be traced through the system and correlated with the data in the database (if applicable) on which it is operating. In the case of a serious failure, the log records indicate the responsible component. In the case of a computational error, the log file lists all of the components that participated in the execution of the test procedure, and the IDs or keys of all entities used. Along with the entity data from the database, this should be enough information to allow development personnel to isolate the error in the source code.

Following is an example of log records from an application that has retrieved a customer object from a database:

```
Function: main (main.cpp, line 100)
Machine: testsrvr (PID=2201)
Timestamp: 1/10/2002 20:26:54.721
Message: connecting to database [dbserver1,
customer_db]
```

```
Function: main (main.cpp, line 125)
Machine: testsrvr (PID=2201)
Timestamp: 1/10/2002 20:26:56.153
```

```
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.

Item 19: Verify That the System Supports Debug and Release Execution Modes

While a software system is under construction, there obviously will be many malfunctions. Problems may surface anytime during developer testing, particularly in the unit and integration test phases and during formal tests by the testing team.

When defects are discovered, there must be a way to diagnose the problem, preferably without any modifications to the application itself. To make this process efficient, the application must support different operating modes, particularly a **debug mode** to help developers and testers diagnose problems as they are encountered, and a **release mode**, an optimized version of the system stripped of most or all of its debugging-related features. The application is usually delivered to end users in release mode.

Depending on the languages, tools, and platforms involved in an application's development, there are several ways it can be debugged.

- *Source-code inspection.* If a problem is not terribly complex, a visual inspection of the source code may be enough to determine the problem. For instance, if an incorrect title bar appears on an application window, the developer can simply look in the source code for the definition of the window's title bar text and change it accordingly.
- *Logging output.* If the application has a logging mechanism, as described in [Item 18](#), the developer can run the application, re-create the error, and then examine the log entries that were generated. Better still, the tester will have provided a log-file excerpt in the defect report. If the excerpt contains enough information for the developer to determine the cause of the problem, the defect can then be corrected in the source code. Otherwise, the developer may have to consider one of the other debugging techniques listed here.
- *Real-time debugging.* In this most powerful debugging technique, the developer "attaches" a debugger to a running instance of the application, and monitors variables and program flow in the debugging environment while interacting directly with the application. Defects are re-created by executing the steps in a defect report, the application code is examined while it is running to determine the source of the defect.

Which of these techniques to use is up to the developer, but the information in the defect report usually makes one approach obvious. For example, if the problem is a

simple cosmetic issue, a quick examination of the source code is usually all that is necessary. Deeper, more complex issues typically require real-time debugging.

Real-time debugging is not always possible, however. In C++, for example, real-time debugging requires a **debug build** of the system. (Debug code also performs many supplemental checks, such as checks for stack and heap corruption—common when the system is under development.) As mentioned previously, the development environment can be "attached" to the debug build of the application while it is running, allowing the developer to monitor the code while interacting with the application. The application's behavior and variables may thus be thoroughly examined—sometimes the only way to properly diagnose and correct a defect.

In addition, a debug build of an application issues more-verbose messages about problems encountered with respect to memory and other lower-level system functions. For example, when low-level errors or unexpected conditions (**assertions**) are encountered while the application is running, it typically will display a dialog box describing the problem as it occurs. Such low-level, diagnostic messages are not meant for end users and usually will not be displayed by a release build of the system; instead, if there is a problem, the system may simply crash or exhibit unpredictable behavior.

Once the system is ready to ship, it is necessary to disable or remove debugging features. In some languages, this involves rebuilding the system. This is necessary because debugging features, such as logging, usually result in slower performance and higher memory usage, and could exhaust system resources, such as disk space, if left unattended. In addition, excess logging can pose a security risk by possibly exposing information to system intruders.

Since it is possible that an application may exhibit new defects when built in release mode, it is important for the test team to be aware of which build **mode** it is testing. For the most part, only the development staff should work with debug builds; the test team should work with release builds. This will ensure that the release build of the software—the version of the system that will be delivered to customers—receives appropriate attention.

The presence of logging mechanisms must also be considered for a release build of the system. The logging mechanism can be removed from the application altogether, or, preferably, configured to do less logging or none at all. Although it may seem that all logging should be removed from the release version of an

application, there can be considerable value in leaving some form of logging enabled. When a configurable logging mechanism is in place in the production version of an application, defects can still effectively be diagnosed even after the application has gone live. Defects discovered by customers can be re-created and diagnosed with the assistance of log files, a more reliable approach than simply attempting to re-create a problem by relying on the user's input and the error messages produced at the application's user interface.

An application's logging mechanism can be made configurable through the use of a configuration file, which specifies the level at which logging should take place. A simple logging system would have two levels, **errors** and **debug information**. In the application's code, a level would be associated with each log entry, and the configuration file would specify the level of logging. Typically, only errors are logged in release mode, whereas both errors and debug statements are logged in debug mode. The following pseudocode illustrates log entries made at each of these two logging levels (errors and debug):

```
write_debug_log_entry "connecting to database"
result = connect_to_database_server "dbserver"
if result == connection_failed then
    write_error_log_entry "failed to connect to
database"
else
    write_debug_log_entry "connected to database"
```

The preceding pseudocode demonstrates the logging of both debug and error messages. Debug messages are written to show application flow and data, while error messages are written when problems are encountered (in this example, failure to connect to the database).

When the application is in debug mode, all messages displaying flow, data, and errors are included in the log file. In release mode, only errors are logged. It follows that, in release mode, the log file should not grow too large, since no debug messages are written and errors should not occur frequently after the application has been through several testing cycles.

To ensure that no defects are introduced into the application by the logging mechanism, a regression test must be run to verify that the system behaves the same with the logging mechanism turned on fully (logging both errors and debug messages) or in a reduced capacity (logging errors only, or no logging) before moving the application into production.

Even if logging is removed from the release build of the application, there should be other diagnostic capabilities, since problems can still occur. Diagnosis without logging can be accomplished through the use of system-provided mechanisms, such as the Windows Event Log or UNIX syslog service.

Separate debug and release modes will allow an application to be diagnosed and debugged at different levels of detail depending on the situation. While the application is under development, it is preferable to have full diagnostic logging and debugging capabilities to assist in the isolation and correction of defects. When the application is in production, less diagnostic information is necessary, but some is still desirable in case end users encounter errors. Debug and release builds, as well as configurable logging mechanisms, make this kind of flexibility possible.

Chapter 5. Test Design and Documentation

Test design and documentation are among the major functions of a testing team. As discussed in [Chapter 1](#), these activities begin not when a software build is placed into the hands of the testing team, but as soon as the first requirements have been approved and baselined. As requirements and system design are refined over time and through system-development iterations, so are the test procedures refined to cover the new or modified requirements and system functions.

Test procedures can benefit from a structured approach that defines the level at which test procedures will operate—black-box or gray-box; the format for test-procedure documentation; and the testing techniques selected, such as examination of boundary conditions and exploratory testing. In addition, test scripts and other testing software development can benefit from a structured approach that decouples logic from data and implements accepted software design principles.

In evaluating the requirements and the software design, by understanding the architecture, and by reviewing prototypes or an existing application, the testing team can "divide and conquer" the various testing tasks, keeping budgets and schedules in mind. The "what," "how," "when," and "who" of feature testing can be decided by determining a strategy, breaking down the tasks, using a specific technique, and prioritizing the test procedures in a parallel and iterative fashion.

Item 20: Divide and Conquer

With the requirements specifications in hand and some design knowledge to work with, the tester is ready to design and develop the test procedures. When a project has a huge number of requirements, deciding where to start with developing test procedures can seem overwhelming. This Item provides an approach to help break down the testing tasks.

Before test design begins, it is necessary to consider the testing phase in which the tests will be executed. There will be different types of tests appropriate for testing usability, performance, security, and other phases of testing, functional and non-functional. Nonfunctional testing is addressed in [Chapter 9](#).

The first step in designing test procedures is to review the test plan, if not already familiar with it, in order to understand the context and framework within which the test objectives, scope, and approach for this testing project are defined. The test plan also lists available resources, such as test equipment and tools; the testing

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.

Tests must be designed for GUI testing or back-end testing, or both, depending on the testing strategy previously determined. GUI tests naturally differ from back-end tests.

- ***Who should develop the tests?*** Once it has been determined what must be tested, when it is to be tested, and how the test is to be accomplished, it will be easier to decide to whom the various testing tasks should be assigned, based on the specific roles and responsibilities of the various testers. For a discussion of tester roles and responsibilities see [Item 13](#).

The following additional questions and issues may also arise during the course of test-procedure design and development.

- ***What if requirements have not been written?*** Without requirements from which to derive test cases, it may be necessary to interview customer representatives, technical staff, and review any available documentation, such as user manuals for the current system (if available yet) or for a legacy system, in order to better understand the application being built. (See [Item 5](#) for a discussion of dangers of basing testing on a legacy product.) If there is a legacy system, design documentation may be available from that system's development process. However, keep in mind that design documents don't necessarily state user requirements correctly. In some cases, it may be necessary to employ reverse engineering of the requirements, and possibly exploratory testing (discussed in [Item 27](#)).
- ***Black-box and gray-box testing.*** Testing that exercises most parts of the system by invoking various system calls through the user interface is called **black-box** testing. For example, if the user adds a record to the application's database by entering data via the user interface, various layers—such as the database layer, the user-interface layer, and business-logic layers—are executed. The black-box tester validates the correct behavior only by viewing the output of the user interface. (See [Chapter 4](#) for more about black-box and gray-box testing.)

Be aware, however, that in some cases errors may not be reported via the user interface due to defects in the error-reporting mechanism of the software. For example, if the application fails to insert a record in the database but does not report an error to the user interface, the user interface receives a "false positive" from the underlying code and continues on without displaying an error message to the user.

Since user-interface, or black-box, testing does not exhibit all defects, gray-box testing must also be applied.^[1] Gray-box testing, addressed in Item 16, requires test designers to possess knowledge of the underlying components that make up the system.

^[1] Keep in mind that black-box and gray-box testing combined are still not sufficient to produce a quality product. Testing processes, procedures, inspections, and walk-throughs, along with unit and integration testing, are all important parts of effective testing. Black-box and gray-box testing alone cannot identify all of the defects in a software program.

- *Will a test harness have to be developed?* Some components of a system can be tested only by a test harness. For example, consider a calculation engine that allows for thousands of input combinations. Testing requires a test design different from user-interface or black-box testing: The numbers of combinations and variations of inputs are too huge to test through the user interface. Time and other considerations may require development of a test harness to directly test the calculation engine by entering a large set of input values and verifying the outputs that result. See Item 37 for further discussion.
- *What types of testing techniques should be used?* Various functional-testing techniques are available for deriving the test data and input values required. One technique for verifying large numbers of data combinations is **orthogonal array** testing. Orthogonal array testing allows the selection of the combinations of test parameters that provide maximum coverage using a minimum number of test cases.^[2]

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

Other testing techniques that can help narrow down a large set of possible input combinations possible are equivalence class partitioning and boundary value analysis (see Item 25). Combining these techniques can result in a narrowed-down, but statistically valuable, set of test inputs.

Another instance when different types of tests are required is when a **commercial off-the-shelf** (COTS) tool is employed. The proper integration of the COTS tool with the custom code itself must be verified. If, for example, a COTS tool has been certified to execute SQL queries proficiently, tests must still be run to verify that the content of the modified SQL queries is correct.

- *Should a capture/playback tool or other testing tool be used?* While a test harness might be strategically applied for testing the calculation engine or other back-end functions, an automated testing (capture/playback) tool might prove useful for GUI regression testing.

Defining a testing strategy requires determining which areas of the application should be tested using an automated capture/playback tool, which require a test harness, and which must be tested manually. Testing exclusively with the capture/playback feature of an automated testing tool should be avoided, for reasons discussed in [Item 36](#).

- *Which tests should be automated?* The test engineer should perform a careful analysis of the application when deciding which tests warrant automation and which should be executed manually. This can help to avoid incurring unnecessary costs, as some tests can be more expensive to automate than to execute manually.

Here are some examples of tests that could be automated:

- *Tests that are executed more than once.* By contrast, a test that is executed only once is often not worth automating.
- *Tests that evaluate high-risk conditions.* Low-risk factors may not be worth testing by automated means. Consider a functional test where there is a minuscule likelihood a user will execute the specific scenario to be tested. For example, the system under test may allow the user to perform actions that, from a business point of view, would not make much sense. The risk associated with scenarios of this type is low, since the number of users affected by its potential failure is low. It doesn't make sense for an automater to concentrate on such issues when designing tests; instead, the focus should be on higher-risk items.
- *Tests that are run on a regular basis.* Examples include **smoke** (build-verification) tests, regression tests, and mundane tests (tests that

include many simple and repetitive steps and must be conducted repeatedly).

- *Tests that would be impossible, or prohibitively expensive, to perform manually.* For example, automation may be called for when verifying the output of a calculation engine simulating 1,000 multiple-user accesses, during stress and performance testing, during memory-leak detection, or during path-coverage testing.
- *Tests that have multiple data values for the same actions* (data-driven tests).
- *Baseline tests to be run on different configurations.*
- *Tests with predictable results.* It is not cost-effective to automate tests whose outcomes are unpredictable.
- *Tests on a system that is somewhat stable,* with functionality, implementation, and technology that are not constantly changing. Otherwise, maintenance will outweigh the benefits of automation.
- *What kind of test data is needed?* Once the testing tasks are understood in detail, a set of test data can be defined that provides input to the tests. It is important that test data is chosen carefully—incorrect or oversimplified test data can result in missed or erroneously identified defects, requiring unnecessary work and reducing test coverage. (For more about acquiring test data, see [Item 10](#).)

Answering the questions listed in this Item will help the test planner break down the testing tasks. It is important to consider the available budgets and schedules.

Item 21: Mandate the Use of a Test-Procedure Template and Other Test-Design Standards^[1]

^[1] Adapted from Elfriede Dustin et al., "Web Engineering Using the RSI Approach," in *Quality Web Systems: Performance, Security, and Usability* (Boston, Mass.: Addison-Wesley, 2002), 52.

For repeatability, consistency, and completeness, the use of a test-procedure template should be mandated when applicable. The following is an example test-procedure template:

TEST PROCEDURE ID:	TEST NAME:
Follow naming convention—TP ID is based on requirement use cases, but starts with T- instead of R-	High-level description of what is being tested
DATE EXECUTED:	TEST ENGINEER INITIALS:
TEST PROCEDURE AUTHOR:	
TEST OBJECTIVE	<p>— Briefly describe the purpose of the procedure:</p>
RELATED USE CASE(s)/REQUIREMENT NUMBER(S)	<p>— List any use-case name(s) and number(s) being tested under this test objective:</p>
PRECONDITIONS/ASSUMPTIONS/DEPENDENCIES	<p>— List any conditions, assumptions, and dependencies to be satisfied before these test-procedure steps can be executed. This may be done in a bulleted-list format. Often the same preconditions as in the use case apply here:</p>
VERIFICATION METHOD:	

Figure 21.1. Test Procedure Template

Functional Testing Steps					Automated / Manual (underline selection)			
Step	User Action (Inputs)	Expected Results	Trace Log Information	Actual Results	Test Data Required	Pass / Fail	Use Case Step # or Requirement Number	
Security Testing Steps					Automated / Manual (underline selection)			
Step	User Action (Inputs)	Expected Results	Trace Log Information	Actual Results	Test Data Required	Pass / Fail	Use Case Step # or Requirement Number	
Performance Testing Steps					Automated / Manual (underline selection)			
Step	User Action (Inputs)	Expected Results	Trace Log Information	Actual Results	Test Data Required	Pass / Fail	Use Case Step # or Requirement Number	
Compatibility Testing Steps					Automated / Manual (underline selection)			
Step	User Action (Inputs)	Expected Results	Trace Log Information	Actual Results	Test Data Required	Pass / Fail	Use Case Step # or Requirement Number	
Usability Testing Steps					Automated / Manual (underline selection)			
Step	User Action (Inputs)	Expected Results	Trace Log Information	Actual Results	Test Data Required	Pass / Fail	Use Case Step # or Requirement Number	

The primary elements of the standard test procedure, shown in Figure 21.1, are:

- *Test Procedure ID*: Use a naming convention for test-procedure IDs.
- *Test Name*: Provide a description of the test procedure.

- *Date Executed*: State when the test procedure was executed.
- *Test Engineer Initials*: Provide the initials of the engineer executing the test procedure.
- *Test Procedure Author*: Identify the developer of the test procedure.
- *Test Objective*: Outline the objective of the test procedure.
- *Related Use Case(s)/Requirement Number(s)*: Provide the identification number of the requirement validated by the test procedure.
- *Preconditions/Assumptions/Dependencies*: Provide the criteria, or prerequisites, to be met before the test procedure can be run, such as specific data setup requirements. This field is completed when the test procedure is dependent on a previous test procedure. This field is also completed when two test procedures would conflict if performed at the same time. Note that the precondition for a use case often becomes the precondition for a test procedure.
- *Verification Method*: This field may include certification, automated or manual test, inspection, demonstration, or analysis.
- *User Action (Inputs)*: Here, the goals and expectations of a test procedure are clearly defined. This can be accomplished by documenting the steps needed to create a test. The entry in this field may be similar to software-development pseudocode. Completing this field allows for clarification and documentation of the test steps required to verify the use case.
- *Expected Results*: Define the results expected when executing the particular test procedure.
- *Trace Log Information*: Document the behavior of back-end components. During execution, Web system components, for example, write log entries detailing the functions they are executing and the major objects with which they are interacting. These log entries can be captured within the test procedure.
- *Actual Results*: This field may have a default value, such as "Same as Expected Result," that is changed to describe the actual result if the test-procedure fails.
- *Test Data Required*: Reference the set of test data required to support execution of the test procedure. See [Item 26](#) for more detail.

Nonfunctional test considerations are often an afterthought in many testing efforts. However, it is important that nonfunctional tests be considered from the beginning of the testing life cycle. Note that the test-procedure documentation form depicted in [Figure 21.1](#) is divided into five sections—one for functional testing steps and the remainder for the nonfunctional areas of security, performance and scalability,

compatibility, and usability. See Item 41 for more information on nonfunctional test considerations.

Test-design standards must be documented, communicated, and enforced so that everyone involved conforms to the design guidelines and produces the required information.^[2] Test-procedure creation standards or guidelines are necessary whether developing manual or automated test procedures. The standards for manual test procedures should include an example of how much detail a test procedure should contain. The level of detail may be as simple as outlining the steps to be taken—for example:

^[2] 4. Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), Section 7.3.4.

Step 1. Click on the File menu.

Step 2. Select Open.

Step 3. Select a directory on the computer's local disk.

Depending on the size of the application being tested, and given the limited schedules and budgets, writing extensive test procedures may be too time-consuming, in which case the high-level test descriptions may be sufficient.

Test-procedure standards can include guidelines on how the expected result of the test is supposed to be documented. The standards should address several questions: Will test results require screen prints? Will tests require sign-off by a second person who observes the execution and result of the test?

When it comes to automated test design standards, standards should be based on best coding practices such as modularity, loose coupling, concise variable and function naming conventions, and so on. These practices are typically the same as those for general software development.

In some situations, not all test scenarios may be documented in the same detail, as a template would require. Some test procedures in complex applications such as financial systems, whose possible inputs to calculations can number in the tens of thousands, may require a different approach to documentation, such as using a spreadsheet.

Item 22: Derive Effective Test Cases from Requirements^[1]

^[1] Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), 1-114.

A functional test exercises an application with the intent to uncover non-conformance with end-user requirements. This type of testing activity is central to most software test efforts. The primary objective in the functional-testing phase is to assess whether the application does what it is supposed to do in accordance with specified requirements.

A good practice for developing test procedures for the functional testing phase is to base them on the functional requirements. Additionally, some of the test procedures created for this testing phase can be modified for use in testing the nonfunctional aspects of the application, such as performance, security, and usability.

In [Chapter 1](#), the importance of having testable, complete, and detailed requirements was discussed. In practice, however, having a perfect set of requirements at the tester's disposal is a rarity. In order to create effective functional test procedures, the tester must understand the details and intricacies of the application. When, as is often the case, these details and intricacies are inadequately documented in the requirements, the tester must conduct an analysis of them.

Even when detailed requirements are available, the flow and dependency of one requirement to the other is often not immediately apparent. The tester must therefore explore the system in order to gain a sufficient understanding of its behavior to create the most effective test procedures.

In general, the tester must analyze how any change to any part of the application, such as to a variable or a field, affects the rest of the application. It is not good enough to simply verify aspects of the change itself—for example, that the system allows for the required change, using a subset of input combinations and variations, and that the change is saved correctly; rather, an effective test must also cover all other areas affected by this change.

For example, consider the following requirement statement:

"The system must allow the user to edit the customer name on the customer record screen."

The customer name field and its restrictions are also documented, ideally in the data dictionary. Some testing steps for verifying that the requirement has been met are:

1. Verify that the system allows the user to edit the customer name on the customer-record screen via the user interface, by clicking on the customer record screen and confirming that the customer name can be edited.
2. Try all positive and negative combinations—e.g., all equivalence classes, all boundaries—of customer names allowed. Test a subset of combinations and variations possible within the data dictionary's restrictions.
3. Run a SQL query verifying that the update is saved correctly in the appropriate table(s).

The above steps comprise a good basic test. However, something is missing in order to fully verify this requirement.

Aside from performance and other nonfunctional issues, the question that needs to be answered is, "how is the system otherwise affected when the customer name is changed?" Is there another screen, functionality, or path that uses or is dependent upon the customer name? If so, it will be necessary next to determine how those other parts of the application are affected. Some examples:

- Verify that the "create order" functionality in the order module is now using this changed customer name.
- Add an order record, and verify that the new record has been saved using the new customer name.
- Perform any other possible functions making use of the changed customer name, to verify that it does not adversely affect any other previously working functionality.

Analysis and testing must continue until all affected areas have been identified and tested.

When requirements are not documented in detail, this seemingly simple strategy for effective functional software testing is often overlooked. In fact, in most cases the requirements are not documented in sufficient detail to clearly define relationships between requirements and functional paths, which is critical to test-procedure development. Often, therefore, effective test procedures cannot be designed from the requirements statements alone.

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.

It is imperative that functional test procedure creation be prioritized based on highest-risk and highest-usage functionality. See [Item 8](#) for more detail.

Effective test-case design requires understanding of system variations, flows, and scenarios. It is often difficult to wade through page after page of requirements documents in order to understand connections, flows, and interrelationships. Analytical thinking and attention to detail are required to understand the cause-and-effect connections within the system intricacies. It is insufficient to design and develop high-level test cases that execute the system only at a high level; it is important to also design test procedures at the detailed, gray-box level.

Item 23: Treat Test Procedures As "Living" Documents

Often, test engineers labor to develop test procedures only to execute them once or twice before they become obsolete due to changes in the requirements, design, or implementation. Given the pressures of having to complete the testing, testers continue their tasks without ever revisiting the test procedures. They hope that simply by using intuition and analytical skills they will cover the most important parts of the system. The problems with this type of approach are that if the test procedures become outdated, the initial work creating these tests is wasted, and additional manual tests executed without having a procedure in place cannot be repeated. It is therefore important to treat test procedures as "living" and not static documents, to avoid having them become "shelfware."

Most software projects are developed in an iterative, incremental manner, and test procedures are often developed the same way. It could be counterproductive and infeasible to develop a comprehensive and detailed set of test procedures in the beginning of the project,^[1] as the development life cycle brings changes during each iteration. Additionally, time constraints often do not allow the testing team to develop a comprehensive set of test procedures.

^[1] The emphasis here is on *detailed* and *comprehensive* test procedures. Design of test procedures should begin as soon as requirements become available, and since requirements are often living documents, it follows that test procedures must be considered living documents as well.

In an iterative and incremental development life cycle, numerous software builds are delivered based on a build schedule, and it can be difficult for test-procedure writers to keep up. When the first build is completed and delivered for testing, the

expectation is that procedures are available to test it. It doesn't make sense to pursue complete test-procedure development for the entire system if the development team is waiting for a build (a subset of the system) to be tested. In these situations, test procedures must be developed that are targeted for the current build, based on the requirements planned for that build.

It is seldom the case that all requirements, design details, and scenarios are documented and available early in the project; some may not even have been conceptualized yet. Seldom is there one document that lists all functional paths or scenarios of a system. Therefore, test procedures must evolve with each phase of the development process. More details surface during the architecture and design phases, and sometimes even during the implementation phase, as issues are discovered that should have been recognized earlier. Test procedures must be augmented or modified to accommodate this additional information. As requirements change, the testers must be made aware of the changes so they can adjust test procedures accordingly.

System functionality may change or be enhanced during the development life cycle. This may affect several test procedures, which must be redesigned to examine the new functionality. As new functionality becomes available, procedures must be included for testing it.

As defects are found and corrected, test procedures must be updated to reflect the changes and additions to the system. As with new functionality, fixes for defects sometimes change the way the system works. For example, test procedures may have included work-arounds to accommodate a major system bug. Once the fix is in place, the test procedures must be modified to adapt to the change and to verify that the functionality is now implemented correctly.

There may be occasions when testers feel they have had enough time to develop a comprehensive set of test procedures, that there are tests in place for all the known requirements, and that the requirements-to-test traceability matrix is complete. No matter how complete the coverage seems to be, however, with any system of at least moderate complexity, a new test scenario can usually be imagined that hasn't been considered before. The scenarios of how a system can be executed can be endless. Therefore, when a new scenario is encountered, it must be evaluated, assigned a priority, and added to the "living" set of test procedures.

As with any living or evolving document, test procedures should be stored in a version control system.

Item 24: Utilize System Design and Prototypes

Prototypes serve various purposes. They allow users to see the expected result of a feature implementation, providing a preview of the look and feel of the application. They allow users to give feedback on what is being developed, which can be used to revise the prototype and make the final product more acceptable to the user.

Prototypes can be helpful in detecting inconsistencies in requirements. When defining detailed requirements, sometimes contradictions can be hard to spot if they appear in sections many pages apart in one or more documents, or when more than one individual develops or maintains the requirements. Although rigorous approaches and manual inspections can minimize incompleteness and contradictions, there are practical limits to their effectiveness. The creation of a prototype and its design can help in discovering inconsistencies or incompleteness and provide a basis for developing custom tools to uncover any additional issues early in the development process.

Prototyping high-risk and complex areas early allows the appropriate testing mechanism (e.g., a test harness) to be developed and refined early in the process. Attempts to engineer complex testing mechanisms late in the cycle are typically unsuccessful. With a prototype in place early in the development cycle, it is possible to begin defining the outline and approach of the test harness, using the prototype as a model that is one step closer to the actual application.

Designs and prototypes are helpful in refining test procedures, providing a basis for additions and modifications to requirements. They thus become a basis for creating better, more-detailed test procedures. Use of prototypes provides additional information that allows for test-procedure enhancements. Prototypes can provide a level of detail that a static requirements document is unable to provide.

Prototypes are also useful in the design of automated tests using functional testing tools. A prototype can help testers determine where the automated testing tool will run into compatibility issues. This allows time for work-arounds to be investigated and automated test designs to be adjusted prior to the delivery of a software build. Sometimes a prototype can reveal a flaw in the automated testing tool early on, allowing sufficient lead-time to acquire a patch from the tool vendor.

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.

- *Equivalence partitioning* identifies ranges of inputs and initial conditions that are expected to produce the same result. Equivalence partitioning relates to the commonality and variances among the different situations in which a system is expected to work. If situations are equivalent, or essentially similar, to one another, it is adequate to test only one of them, not all. Although equivalence is usually intuitively obvious, it is necessary to be careful about what is assumed to be equivalent.

Consider the following example, which demonstrates equivalence applied to boundaries:

A password field allows 8 digits; any more are invalid. Since values that are on the same side of a boundary are members of the same "equivalence class," there is no point to testing many members of the same equivalence class (e.g., passwords with 10 digits, 11 digits, 12 digits, etc.), since they will produce the same result.

- *Path analysis* is used to test the internal paths, structure, and connections of a product. It can be applied at two levels. One level is code-based, or **white-box**, testing, and is usually performed during unit test. The **unit test** is done close to the point of creation or modification of the code, usually by the author or programmer. (See [Chapter 6](#) for more detail on unit testing.)

Path analysis also can be applied at a second level: at the functional or black-box level. The source code may not be available, and black-box testing usually is performed by system testers or user-acceptance testers. Even if the source code is available, testing at the black-box level is performed according to the specified requirements without evaluating the specifics of their implementation in the source code.

- *Boundary-value (BV) analysis* can complement the path-analysis approach. BV testing is used mostly for testing input edit logic. So-called **negative testing**, a variation of BV, checks that the processes for filtering out invalid data are working acceptably. Boundary conditions should be determined as part of designing effective BV test cases, since many defects occur on the boundaries.

Boundaries define three sets or classes of data: good, bad, and borderline (also known as **in-bounds**, **out-of-bounds**, and **on-bounds**).

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 parameters would require twenty-seven test cases (3³). Are all twenty-

seven of those tests needed? They are if the tester suspects a fault that depends on a specific set of values of all three parameters (a fault, for example, that would occur only for the case when A=1, B=1, and C=1). However, it's probably more likely that a fault depends on the values of only two of the parameters. In that case, the fault might occur for each of these three test cases: (A=1, B=1, and C=1); (A=1, B=1, and C=2); and (A=1, B=1, and C=3). Since the value of C in this example appears to be irrelevant to the occurrence of this particular fault, any one of the three tests will suffice. Given that assumption, the following array shows the nine test cases required to catch all such faults:

Case #	A	B	C
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

The array is orthogonal because for each pair of parameters all combinations of their values occur once. That is, each possible pair of parameters (A and B), (B and C), and (C and A) is shown once. In terms of pairs, it can be said that this array has a strength of 2. It doesn't have a strength of 3 because not all possible three-way combinations occur—for example, the combination (A=1, B=2, and C=3) doesn't appear. What is important here is that it covers all of the pair-wise possibilities.

It is up to the test designer's judgment to select a representative data sample that truly represents the range of acceptable values. This is sometimes very difficult when there are numerous interrelationships among values. In such cases, the tester may consider adding random samples of possible data sets to the already generated set.

It is generally not possible, feasible, or cost-effective to test a system using all possible permutations of test-parameter combinations. Therefore, it is important to

use boundary-value analysis in conjunction with other testing techniques, such as equivalence partitioning and orthogonal arrays, to derive a suitable test data set.

Item 26: Avoid Including Constraints and Detailed Data Elements within Test Procedures

[Item 21](#) discusses the importance of using of a test-procedure template for defining and documenting the test-procedure steps.

The template should be written in a generic manner, to keep it maintainable. It is not good practice to include specific test data in a test procedure, since that would result in unnecessary duplication: For each test data scenario, all test-procedure steps would have to be repeated, with the only changes being the differing data inputs and expected results. Such unnecessary duplication could become a maintenance disaster—for example, were a data element to change, the change would need to be propagated to all test procedures. Consider a Web URL referenced in the documentation for all test procedures. If this URL changes, it must be modified in all of the test-procedure documents—a time-consuming and potentially error-prone task.

To make a test procedure easily maintainable, specific test-data inputs and related expected outputs should be kept in a separate scenario document. Test-data scenario information may be kept in a spreadsheet or database, with each row defining a separate test case. These scenarios can be attached to the test procedure, with example scenarios for each test procedure that feature concrete data. When deriving concrete data, refer to the data dictionary in order to reference data constraints. By helping ensure that test procedures are reusable, these procedures avoid difficult maintenance problems that would be posed were test procedures written using very low-level detail, hard-coded with data that ties the procedures to this specific scenario.

[Figure 26.1](#) provides a simplified example of a test-data scenario spreadsheet used for verifying user IDs and passwords during the system's log-on function.^[1]

Assume a test procedure exists that documents all the steps necessary for logging on. In the test procedure, the steps will include the data elements—user ID and password. Whenever those data elements are referenced, the tester should refer to the scenario spreadsheet. Note that in actual use, the spreadsheet should include "Actual Outcome" columns (not displayed in [Figure 26.1](#)) so testers can document the outcomes during test execution without having to use yet another document.

^[1] Not all possible test data combinations are listed here.

Figure 26.1. Example Test-Scenario Spreadsheet

Test Procedure ID (cross-reference to the test procedure to which this test data relates)			
User ID	Expected Outcome	Password	Expected Outcome
One more digit than allowed—meeting all other constraints	Rejected	One more digit than allowed	Rejected
One less digit than allowed—meeting all other constraints	Accepted	One less digit than allowed	Accepted
Exactly the same number of allowed digits—meeting all other constraints	Accepted	Exactly the number of allowed digits	Accepted
Use combinations of characters that are allowed—meeting all other constraints	Accepted	Use combinations of characters that are allowed	Accepted
Use characters that are not allowed	Rejected	Use characters that are not allowed	Rejected
Use zero input	Rejected	Use zero input	Rejected
Use null input	Rejected	Use null input	Rejected

Documenting the test data scenarios described in [Figure 26.1](#) for each separate test procedure would require a huge effort. Keeping the test data in a separate spreadsheet or database, referenced in the main test procedure, allows for easier maintenance and more-effective test documentation.

Test-procedure documentation in which data elements are kept outside of the main test procedure, residing instead in a separate location, is especially helpful for automated testing, and can serve as a foundation for other test procedures as well. For both manual and automated test procedures, data elements should be kept separate from their test scripts, using variables instead of hard-coded values within the script, as discussed in [Chapter 8](#).

Item 27: Apply Exploratory Testing

As discussed in [Item 22](#), complete and detailed requirements that define each relationship and dependency are a rarity in today's development environments. A tester is often required to use his analytical skills to determine the intricacies of the application. Sometimes, exploratory testing is required in order to gain knowledge needed for designing effective tests.

The functionality is explored because it is not well understood before testing begins. Exploratory testing is most useful when not much is known about the system under test, such as when the functional specifications are informal or absent, or when there is not enough time to come up with detailed designed and documented test procedures. Exploratory tests can enhance the testing described in [Item 22](#).

Exploratory testing identifies test conditions based on an iterative approach. The pattern of problems found early in exploratory testing helps focus the direction of later test efforts. For example, if an initial investigation indicates that one area of the system is quite buggy while another area is relatively clean, testing is refocused based on this feedback.

Defect-prone areas identified during exploratory testing should be evaluated to determine their correlation to complex areas of the software. Unlike well-thought-out and -planned test procedures, exploratory tests are not defined in advance or carried out precisely according to a plan.

All testing efforts require exploratory testing at one time or another, whether test procedures are based on the most-detailed requirements or no requirements are specified. As testers execute a procedure, discover a bug, and try to recreate and analyze it, some exploratory testing is inevitably performed to help determine the cause. Additionally, during the defect-discovery process, the relationship of a defect to other areas of the application is explored and researched. This usually requires stepping outside the boundaries of the defined test-procedure steps, as all possible test scenarios are rarely documented—it wouldn't be economically feasible to do so.

Another occasion for exploratory testing is when a tester must determine the thresholds (maximums and minimums) of specific features. Consider the following example.

A requirement states: "The field must allow for the creation of pick-list items."

Assuming the data dictionary defines the term "pick-list items" and its associated restrictions and limitations, the tester's first question should be, "How many pick-list items are allowed?" In our example, this question was overlooked during the requirements phase. (Such matters can be missed even during walk-throughs and inspections.)

The tester researches the existing requirements and asks the developers, but the answer is not available. The customer representative has only a vague idea. It becomes necessary for the tester to write and execute a procedure that determines how many pick-list items the field allows. As the tester adds pick-list items, a point comes when the application's performance starts to degrade, or the application simply does not allow the addition of another pick-list item. The tester has now determined the threshold number. After verifying with a customer representative that this number is acceptable, the tester publishes the number for all stakeholders. The tester can now continue executing the original test of the relationship between the field (and its pick-list items) to the application, as described in [Item 22](#).

Exploratory tests cannot be pre-planned, as they are conducted on a case-by-case basis in reaction to encountered conditions. However, it is a good practice to document the exploratory tests before or after execution, adding them to the suite of regression tests (regression testing is discussed in [Item 39](#)) for repeatability and reusability and in order to measure adequate coverage.

Exploratory testing "rounds out" the testing effort: It enhances the functional testing described in [Item 22](#), but is not a very powerful technique by itself. Using exploratory testing, often testing coverage cannot be determined or measured, and important functional paths can be missed. It is important that functional-testing procedures be enhanced with new knowledge gained during exploratory testing, treating them as "living documents" as discussed in [Item 23](#).

The most powerful testing effort combines a well-planned, well-defined testing strategy with test cases derived by using functional analysis and such testing techniques as equivalence, boundary testing, and orthogonal-array testing (discussed in [Item 25](#)), and is then enhanced with well-thought-out exploratory testing.

There is never enough time in a testing cycle to document all possible test scenarios, variations, and combinations of test procedures. Exploratory testing is a valuable technique for ensuring that most important issues are covered in the testing life cycle.

Chapter 6. Unit Testing

Unit testing is the process of exercising an individual portion of code, a **component**, to determine whether it functions properly. Almost all developers perform some level of unit testing before regarding a component or piece of code as complete. Unit testing and integration testing are instrumental for the delivery of a quality software product; yet they are often neglected, or are implemented in a cursory manner.

If unit testing is done properly, later testing phases will be more successful. There is a difference, however, between casual, ad-hoc unit testing based on knowledge of the problem and structured, and repeatable unit testing based on the requirements of the system.

To accomplish structured and repeatable unit testing, executable unit-test software programs must be developed, either prior to or in parallel with development of the software. These test programs exercise the code in ways necessary to verify that it provides the functionality specified by the requirements and that it works as designed. Unit-test programs are considered to be part of the development project, and are updated along with the requirements and source code as the project evolves.

Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing. Discovering defects while a component is still in the development stage offers a significant savings in time and costs: It will not be necessary to place the defect into the defect-tracking system, recreate it, and research it. Rather, it can be fixed in place by one developer prior to release of the software component.

The unit-testing approach discussed here is based on a lightweight, pragmatic method of testing applicable to most software projects. Other, more complicated approaches to unit testing, such as path-coverage analysis, may be necessary for very high-risk systems. However, most projects have neither the time nor the budget to devote to unit testing at that level.

The unit-testing approach covered here differs from **pure unit testing**, the practice of isolating a component from *all* external components that it may call to do its work, allowing the unit to be tested completely on its own. Pure unit testing requires that all underlying components be **stubbbed**^[1] to provide an isolated environment, which can be very time consuming and carries a high maintenance

penalty. Since the unit-testing approach discussed here does not isolate the underlying components from the component under test, this type of unit testing results in some integration testing as well, since the unit under test calls lower-level components while it operates. This approach to unit testing is acceptable only if those underlying components have already been unit tested and proven to work correctly. If this has been done, any unit-test failures are most likely to be in the component under test, not in the lower-level components.

^[1] **Stubbing** is the practice of temporarily creating placeholders for subroutines or other program elements. The stubs return predetermined (hard-wired) results that allow the rest of the program to continue functioning. In this way, program elements can be tested before the logic of all the routines they call has been completed.

Item 28: Structure the Development Approach to Support Effective Unit Testing

Software engineers and programmers must be accountable for the quality of their work. Many view themselves as producers of code who are not responsible for testing the code in any formal way; that, in their minds, is the job of the system-testing team. In reality, programmers must be responsible for producing high-quality initial products that adhere to the stated requirements. Releasing code to the testing team when it contains a high number of defects usually results in long correction cycles, most of which can be dramatically reduced through the proper use of unit testing.

Although it is possible that knowledge of the code could lead to less-effective unit tests,^[1] this is generally not applicable if the component is performing specific functions related to documented requirements of the system. Even when a component performs only a small part of a functional requirement, it is usually straightforward to determine whether the component fulfills its portion of the requirement properly.

^[1] The author of the code, being thoroughly versed in its inner workings and intended design, may write tests only for scenarios it is likely to pass. Having another person unit test the code introduces a different viewpoint and may catch problems that would not occur to the original author.

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

language) that are called by code in other layers to interact with the database.

^[1] **Wrapping up** refers to the practice of putting code or data in one location that is referenced when needed. This way, any changes can be made in that one location, rather than every place where it is used.

- **Domain objects.** A set of classes representing entities in the system's problem domain, such as an "account" or an "order," a domain object typically interacts with the database layer. A domain object contains a small amount of code logic, and may be represented by one or more database tables.
- **Business processing.** This refers to components or classes implementing business functionality that makes use of one or more domain objects to accomplish a business goal, such as "place order" or "create customer account."
- **User interface.** The user-visible components of the application that provide a means to interact with the system, this layer can be implemented in a variety of ways. It may be a window with several controls, a Web page, or a simple command-line interface, among other possibilities. The user interface is typically at the "top" of the system's layers.

The above list is a somewhat simplified example of a layered implementation, but it demonstrates the separation of functionality across layers from a "bottom-up" perspective. Each layer consists of several code modules that work together to perform the functionality of the layer.

During the development of such a system, it is usually most effective to assign a given developer to work with a single layer, designing it to communicate with components in other layers via a documented and defined **interface**. In a typical interaction, a user chooses to perform some action via the user interface, and the user-interface (UI) layer calls the business-processing (BP) layer to carry out the action. Internally, the BP layer uses domain objects and other logic to process the request on behalf of the UI layer. During the course of this processing, domain objects will interact with the database-abstraction layer to retrieve or update information in the database. There are many advantageous features in this approach, including the separation of labor across layers, a defined interface for performing work, and the increased potential for reusing layers and code.

Each layer typically includes one or more unit test programs, depending on the size and organization of the layer. In the preceding example, a domain object unit test program would, when executed, attempt to manipulate each domain object just as if the BP layer were manipulating it. For example, the following pseudocode outlines a unit test for a domain-object layer in an order processing system that features three types of elements: "customer," "order," and "item." The unit test attempts to create a customer, an order, and an item, and to manipulate them.

```
// create a test customer, order, and item
try
{
    Customer.Create("Test Customer");
    Order.Create("Test Order 1");
    Item.Create("Test Item 1");
    // add the item to the order
    Order.Add(Item);
    // add the order to the customer
    Customer.Add(Order);
    // remove the order from the customer
    Customer.Remove(Order);
    // delete the customer, order, and item
    Item.Delete();
    Order.Delete();
    Customer.Delete();
}
catch(Error)
{
    // unit test has failed since one of the operations
    // threw an error - return the error
    return Error;
}
```

Similarly, the BP layer includes a unit test that exercises its functionality in the same way that the user interface does, as in the following pseudocode:

```
// place an order
try
{
    OrderProcessingBP.PlaceOrder( "New Customer" ,
ItemList );
}
```

```
catch(Error)
{
    // unit test has failed since the operation
    // threw an error - return the error
    return Error;
}
```

A natural result of unit testing a layered system without isolating underlying components is some integration testing of the component under test with its associated underlying components, since a higher layer will call the services of a lower layer in order to do its work. In the preceding examples, the BP component uses the customer, order, and item domain objects to implement the place-order functionality. Thus, when the unit test executes the place-order code, it is also indirectly testing the customer, order, and item domain objects. This is a desirable effect of unit testing, since it allows unit-test failures to be isolated to the particular layer in which they occur. For example, suppose the domain objects unit test is successful, but the BP unit test fails. This most likely indicates either an error in the BP logic itself or a problem with the integration between the two layers. Without the domain object unit test, it would be more difficult to tell which layer in fact has a problem.

As mentioned earlier, unit tests should be based on the defined requirements of the system, using use cases or other documentation as guides. A functional requirement typically has implementation support in many layers of the system, each layer adding some piece necessary to satisfying the requirement, as determined during the design phase. Given this multi-layer involvement, each unit test for each affected layer must test the components to make sure they properly implement their pieces of the requirement.

For example, the order-processing system described earlier might have a requirement entitled "Discontinue Item." To satisfy this requirement, the system needs a BP component that can load an item and discontinue it, and can check whether any open orders include this item. This in turn requires that the domain object and database layers allow the item object to be discontinued, perhaps through a `discontinue()` method, and that the order object support searching for items in the order using an ID. Each layer participates in satisfying the requirement by providing methods or implementations.

Preferably, a representative test for each requirement is included in the unit-test program for each applicable layer, to demonstrate that the layer provides the

functionality necessary to satisfy the requirement. Using the previous example, the unit tests for each layer would include a "Test Discontinue Item" method that attempts to test the discontinue-item process against the components whose functionality relates to the requirement.

In addition to testing successful cases of the requirement, **error cases** (also known as **exceptions**) should be tested to verify that the component gracefully handles input errors and other unexpected conditions. For example, a particular requirement states that the user must provide a full name which, as defined in the data dictionary, should not exceed 30 characters. The unit test would try a name of acceptable length, and would also attempt to specify a name of 31 characters, to verify that the component restricts the input to 30 characters or fewer, the boundary specified in the data dictionary.

Item 29: Develop Unit Tests in Parallel or Before the Implementation

Popularized by the development style known as **extreme programming**,^[1] the concept of developing unit tests prior to the actual software is useful. Under this approach, requirements guide unit-test development, so they must be defined prior to the development of unit tests. A single requirement usually has implications for many unit tests in the system, which must check the component under test for adherence to the part of the requirement it needs to fulfill. See [Item 28](#) for a discussion of how requirements affect unit testing in a multi-layered system.

^[1] Generally speaking, **extreme programming** (XP) is a way to develop software. XP changes the way programmers work together. One of the major parts of XP is that programmers work in pairs, and that testing is an intrinsic part of the coding process. For more information, see Kent Beck, *Extreme Programming Explained: Embrace Change* (Boston, Mass.: Addison-Wesley, 2000).

There are many benefits to developing unit tests prior to implementing a software component. The first, and most obvious, is that unit testing forces development of the software to be pursued in a way that meets each requirement. The software is considered complete when it provides the functionality required to successfully execute the unit test, and not before; and the requirement is strictly enforced and checked by the unit test. A second benefit is that it focuses the developer's efforts on satisfying the exact problem, rather than developing a larger solution that also happens to satisfy the requirement. This generally results in less code and a more

straightforward implementation. A third, more subtle benefit is that the unit test provides a useful reference for determining what the developer intended to accomplish (versus what the requirements state). If there is any question as to the developer's interpretation of the requirements, it will be reflected in the unit test code.

To properly take advantage of this technique, the requirement documentation must for the most part be complete prior to development. This is preferred because developing software prior to the specification of requirements for a particular function can be risky. Requirements should be specified at a somewhat detailed level, so developers can easily determine which objects and functions are required.^[2] From the requirement documentation, the developer can lay out a general unit-test strategy for the component, including tests for success and failure.

^[2] The RSI (Requirements-Services-Interfaces) approach to use case analysis is an effective way to document requirements from both user and system perspectives. For more information on requirements definition and RSI, see Elfriede Dustin et al., *Quality Web Systems* (Boston, Mass.: Addison-Wesley, 2002), Chapter 2.

To ease the development of unit tests, developers should consider an **interface-based** approach to implementing components. It is good software engineering practice to design software around interfaces, rather than around how components function internally. Note that component or software interfaces are not the same as *user interfaces*, which present and retrieve information from the user through graphical or textual means. Component interfaces usually consist of functions that can be called from other components and perform a specific task given a set of input values. If function names, inputs, and outputs are specified and agreed upon, then the implementation of the component may proceed. Designing the interface between components and higher-level functions first allows the developer to design the component from a high level, focusing on its interaction with the outside world. It may also be helpful for the development of the unit test, since the component's interface can be **stubbed**, meaning that each function called by the interface is written to simply return a predetermined, hard-coded result, with no internal logic. For example, consider the following interface:

```
class Order
{
    Create(orderName);
```

```
    Delete();
    AddItemToOrder(Item);
}
```

The functions referenced in the interface are determined based on requirements that state the system must provide a way to create an order, delete an order, and add items to an order. For writing the unit test, among other purposes, the interface can be stubbed, as in the following example:

```
Create(orderName)
{
    return true;
}
Delete()
{
    return true;
}
AddItemToOrder(Item)
{
    return true;
}
```

The empty functions, or interface stubs, don't actually do anything useful; they simply return "true." The benefit of stubbing a component is that a unit test can be written (and compiled, if necessary) against the interface and will still work properly once the functions are actually implemented.

Unit tests can also assist in the development of the interface itself, since it is useful at times to see how a component will be actually used in code, not just in a design document. Implementing the unit test may lead to refinements and "ease-of-use" enhancements to the component, as the process of implementing code against an interface tends to highlight deficiencies in its design.

In practice, it may be difficult to always create unit tests as the first step. In some situations, parallel development of unit tests and implementation is necessary (and acceptable). There are numerous possible reasons: It may not be immediately obvious how to design the best interface for the component based on the requirements; or the requirements may not be entirely complete, because of outstanding questions or other nonrequirement factors such as time constraints. In such cases, every attempt should still be made to define the component's interface as completely as possible up-front, and to develop a unit test for the known parts of

the interface. The remaining portions of the component and the associated unit tests can evolve as development continues on the component.

Updates to the requirements should be handled as follows: First, the unit test is modified with the new requirements, which may require additional functions in the component's interface, or additional values to be taken or returned by the interface. In conjunction with unit-test development, the interface is updated with stubbed implementations of the new parts, to allow the unit test to function. Finally, the component itself is updated to support the new functionality—at which point the developer has an updated component that works with the new requirements, along with an updated unit test.

Item 30: Make Unit-Test Execution Part of the Build Process

Most software systems of significant size are composed of source code that must be built, or **compiled**^[1] into executables that can be used by the operating system. A system usually contains many executable files, which may call upon one another to accomplish their work. In a large system, the time it takes to compile the entire code base can be quite significant, stretching into hours or days depending on the capabilities of the hardware performing the build.

^[1] In most development environments, the term **compiling** describes the act of producing an executable module from a set of source-code files, such as a collection of C++ files.

In many development environments, each developer must also build a **local version** of the software on the developer's own machine, then make the necessary additions and modifications to implement new functionality. The more code there is to compile, the longer it takes to build—time spent by each developer building each local version of the system. In addition, if there is some defect in a lower layer of the system, the local version may not function properly, which could lead the developer to spend extensive time debugging the local version.

As discussed in Item 29, unit-test programs are valuable for ensuring that the software functions as specified by the requirements. Unit-test programs can also be used to verify that the latest version of a software component functions as expected, prior to compiling other components that depend on it. This eliminates wasted build time, while also allowing developers to pinpoint which component of the system has failed and start immediately investigating that component.

In a layered software architecture, as described in [Item 28](#), layers build upon each other, with higher layers calling down to lower layers to accomplish a goal and satisfy a requirement. Compiling a layered system requires that each lower layer be present—compiled and ready for use—for the next-higher layer to successfully compile and run. This kind of bottom-up build process is common, and allows for reuse of layers, as well as separation of responsibilities among developers.

If unit tests have been written for the components in each layer, it is usually possible for the build environment to automatically execute the unit-test programs after the build of a layer is finished. This could be done in a make-file or a post-build step, for example. If the unit test executes successfully, meaning that no errors or failures are detected in the layer's components, the build continues to the next layer. If the unit test fails, however, the build stops at the layer that failed. Tying a successful build to successful unit-test execution can avoid a lot of wasted compiling and debugging time, while ensuring that the unit tests are actually executed.

It is quite common that unit tests are not updated, maintained, and executed on a regular basis after they are written. Requiring each build to also execute the associated unit tests avoids these problems. This comes with a price, however. When project schedules are tight, especially during bug-fix and testing cycles, there can be considerable pressure to turn fixes around very quickly, sometimes in a period of minutes. Updating the unit-test programs to allow the layer to build can seem like a nuisance, even a waste of time at that particular moment. It is important to keep in mind, however, that the minimal time spent updating a unit test can prevent hours of debugging and searching for defects later. This is especially important if pressure is high and source code is being modified at a fast pace.

Many development projects use **automated builds** to produce regular releases of systems, sometimes on a nightly basis, that include the latest changes to the code. In an automated-build situation, the failure of a component to compile properly halts the build until someone can rectify the issue with the source code. This is, of course, unavoidable, since a syntactical error in the source code must be examined and corrected by a developer in order for the build to proceed.

Adding automated unit-test execution to the build adds another dimension of quality to the build, beyond simply having a system that is syntactically correct and therefore compiles. It ensures that the product of an automated build is in fact a

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

underlying components that it is very difficult to manage them any way other than as part of the layer. Having them reside in the same workspace or project also makes it easier to automatically execute them at the end of each build.

The reusable portions of a unit-testing framework, however, can exist elsewhere, and simply be called by each unit-test program as needed.

Chapter 7. Automated Testing Tools

Automated testing tools can enhance the testing effort—as long as expectations are managed, tool issues are understood, and a tool compatible with the system-engineering environment is selected. It is also important that the tool match the task at hand, and that the tests lend themselves to automation. Many types of testing tools are available for use throughout the various development life cycle phases, including the highly touted capture/playback testing tools.

Sometimes it is determined after research and evaluation that no tool available on the market completely suits the project's needs. In that case, it is necessary to decide whether to develop a unique solution, in the form of scripts or custom tools, or to rely solely on manual testing.

In other cases, there may be commercially available tools that would work for the testing effort, but offer more features than are needed, adding significant cost. This is another situation in which building a custom tool may be reasonable. However, careful analysis is required to verify that building the tool will not be more expensive in the long run than buying it.

There are also situations that mandate a custom-built tool, especially if a particular system component poses a high risk, or is proprietary and cannot be tested by an off-the-shelf tool.

In some cases, particularly when investing large amounts of money in automated testing tools, the needs of the entire organization must be considered. The tool that is selected or built must work properly with the selected technologies, and must fit into the project's budget and time frame allowed for introducing a tool. Evaluation criteria must be established, and the tool's capabilities must be verified according to those criteria prior to purchase.

Examining these issues early in the life-cycle will avoid costly changes later.

Item 31: Know the Different Types of Testing-Support Tools^[1]

^[1] Adapted from Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), Section 3.2.

While functional testing tools (also called "capture/playback tools") are much hyped in the testing industry, it is important to be aware of the other types of tools available to support the testing life cycle. This item provides an overview of 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 these 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

system performance, and functionality across the three components. Many network test tools allow the test engineer to monitor, measure, test, and diagnose performance across an entire network.

- **GUI-testing tools (capture/playback tools).** Many automated GUI testing tools are on the market. These tools usually include a record-and-playback feature, which allows the test engineer to create (record), modify, and run (play back) automated tests across many environments. Tools that record the GUI components at the user-interface control, or "widget," level (not the bitmap level) are most useful. The record activity captures the keystrokes entered by the test engineer, automatically creating a script in a high-level language in the background. This recording is a computer program, referred to as a test script. Using only the capture and playback features of such a tool uses only about one-tenth of its capacity, however. To get the best value from a capture/playback tool, engineers should take advantage of the tool's built-in scripting language. See [Item 36](#) for more on why not to rely on capture/playback alone.

The recorded script must be modified by the test engineer to create a reusable and maintainable test procedure. The outcome of the script becomes the baseline test. The script can then be played back on a new software build to compare its results to the baseline.

A test tool that provides recording capability is usually bundled with a comparator, which automatically compares actual outputs with expected outputs and logs the results. The results can be compared pixel-by-pixel, character-by-character, or property-by-property, depending on the test comparison type, and the tool automatically pinpoints the difference between the expected and actual result. For example, Rational Robot and Mercury's Winrunner log a positive result in the test log as a pass and depict it on the screen in green, while a negative result is represented in red.

- *Load, performance, and stress testing tools* . Performance-testing tools allow the tester to examine the response time and load capabilities of a system or application. The tools can be programmed to run on a number of client machines simultaneously to measure a client-server system's response times when it is accessed by many users at once. Stress testing involves running the client machines in high-stress scenarios to determine whether and when they break.
- *Specialized tools* . Different application types and architectures will require specialized testing of architecturally specific areas. For example, a Web

application will require automated link testers to verify that there are no broken links, and security test programs to check the Web servers for security related problems.

Item 32: Consider Building a Tool Instead of Buying One

Suppose a testing team has automated 40 percent of the testing for a project using an off-the-shelf capture/playback tool, only to realize that adequate testing coverage cannot be achieved by using this tool. In situations like these, in addition to asking the development team to add **testability hooks** (code inserted into the program specifically to facilitate testing), the team must consider building a new automated testing "tool" that enhances the capture/playback tool's reach.

Inadequacy of the testing tool is not the only reason why this type of automated testing might be insufficient. Sometimes the application may not lend itself to automation using a capture/playback tool, either because there is no compatible tool on the market, or because the task at hand is too complex for the capture/playback tool to handle. These, too, are cases requiring custom solutions.

Whether to buy or build a tool requires management commitment, including budget and resource approvals. The course of action must be carefully evaluated: Pros and cons of building versus buying must be weighed. Initially, it might seem easier and cheaper to simply build a tool rather than buying one, but building a tool from scratch can be just as expensive, or even more expensive, than buying one, since the custom tool itself must be tested and maintained.

Nonetheless, there are certain situations that offer no other choice than to build the tool:

- *Operating system incompatibility* . If the tool-evaluation suggestions in [Item 34](#) have been followed and it has been determined that no tool currently on the market is compatible with the various operation systems in use, there is no other choice than to consider building a custom tool that will work in the specific environment under test.
- *Application incompatibility* . The application under test may contain an element, such as a third-party add-on, known to cause compatibility problems with any capture/playback tool on the market. If a work-around cannot be created, it may not be possible to use the capture/playback tool. Even if a work-around can be created, it may not be worth the effort; instead of investing money in a capture/playback tool that is not 100-percent

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

tools available on the Internet. Some are dedicated testing tools, such as DejaGNU, TETware, xUnit, and Microsoft's Web Application Stress Tool (WAST), while others, such as macro-recorders, XML editors, and specialized scripting tools, are not designed specifically for testing but may be useful in certain situations.

Finally, major tool vendors often offer lower-priced "light" (less-capable) versions of their tools, in addition to their major line of tools. For example, Mercury offers the Astra line, Rational offers Visual Test, and Watchfire offers the Linkbot Personal Edition. These simplified versions may be sufficient for certain testing efforts, when the budget doesn't allow purchasing the more-expensive, more-powerful tools.

Item 33: Know the Impact of Automated Tools on the Testing Effort^[1]

^[1] Adapted from Elfriede Dustin et al., *Automated Software Testing* (Reading, Mass.: Addison-Wesley, 1999), Chapter 2.

Automated testing tools are merely a part of the solution—they aren't a magic answer to the testing problem. Automated testing tools will never replace the analytical skills required to conduct testing, nor will they replace manual testing. Automated testing must be seen as an enhancement to the manual-testing process.

Along with the idea of automated testing come high expectations. A lot is demanded of technology and automation. Some people have the notion that an automated test tool should be able to accomplish everything from test planning to test execution, without much manual intervention. While it would be great if such a tool existed, there is no such capability on the market today. Others wrongly believe that it takes only one test tool to support all test requirements, regardless of environment parameters such as operating system or programming language. However, environment factors do put limits on manual and automated testing efforts.

Some may incorrectly assume that an automated test tool will immediately reduce the test effort and schedule. While automated testing is of proven value and can produce a return on investment, there isn't always an immediate payback. Sometimes an automation effort might fail, because of unrealistic expectations, incorrect implementation, or selection of the wrong tool.

The following list corrects some common misconceptions about automated testing that persist in the software industry, and provides guidelines for avoiding the automated-testing euphoria.

- *Multiple tools are often required.* Generally, a single test tool will not fulfill all the automated testing requirements for an organization, unless that organization works only with one type of operating system and one type of application. Expectations must be managed. It must be made clear that currently there exists no one single tool on the market that is compatible with all operating systems and programming languages.

This is only one reason why several tools are required to test the various technologies. For example, some tools have problems recognizing third-party user-interface controls. There are hundreds of third-party add-on controls available with which a developer could implement an application. A tool vendor might claim to be compatible with the main language in which the application is developed, but it is not possible to know whether the tool is compatible with all of the third-party add-ons that may be in use, unless the vendor specifically claims compatibility with them all. It's almost impossible to create a single test tool that is compatible with all third-party add-ons.

If a testing tool is already owned but the system architecture has not yet been developed, the test engineer can give the developers a list of third-party controls that are supported by the test tool. See [Item 34](#) for a discussion on efficiently evaluating a tool and purchasing the correct tool for the development environment.

- *Test effort does not decrease.* A primary impetus for introducing an automated test tool in a project may be the hope of reducing the testing effort. Experience has shown that a learning curve is associated with attempts to apply automated testing to a new project. Test-effort savings do not necessarily come immediately. The first time through, it takes more effort to record the scripts and then edit them to create maintainable and repeatable tests than it would to just execute their steps manually. However, some test or project managers who may have read only the test-tool literature may be over-anxious to realize the potential of the automated tools more rapidly than is feasible.

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,

or data elements could be cut off at the edges.) Then, the actual content printed on the paper must be verified visually. [Chapter 5](#) discusses which tests to automate.

- *A test tool will not allow for automation of every possible test combination.* A commonly-believed fallacy is that a test tool can automate 100 percent of the requirements of any given test effort. Given an endless number of permutations and combinations of system and user actions possible with modern applications, there is not enough time to test every possibility, no matter how sophisticated the automated testing tool is.

Even if it were theoretically possible, no test team has enough time or resources to support 100-percent automation of all testing for an entire application.

- *The tool's entire cost includes more than its shelf price.* Implementation of automated testing incurs not only the tool's purchase cost, but also training costs, costs of automated script development, and maintenance costs.
- *"Out-of-the-box" automation is not efficient.* Many tool vendors try to sell their products by exaggerating the tool's ease of use. They overlook any learning curve associated with the use of the new tool. Vendors are quick to point out that the tool can simply capture (record) a test engineer's keystrokes and create a script in the background, which can then simply be played back. But effective automation is not that simple. Test scripts generated by the tool during recording must be modified manually, requiring tool-scripting knowledge, in order to make the scripts robust, reusable, and maintainable.
- *Training is required.* Sometimes a tester is handed a new testing tool only to have it end up on the shelf or be implemented inefficiently, because the tester hasn't had any training in using the tool. When introducing an automated tool to a new project, it's important that tool training be incorporated early in the schedule and regarded as one of the important milestones. Since testing occurs throughout the system development life cycle, tool training should occur early in the cycle. This ensures that the automated-testing tool can be applied to earlier as well as later testing phases, and that tool issues can be brought up and resolved early. In this way, testability and automation capabilities can be built into the system under test from the start.

Sometimes tool training is initiated too late in a project to be useful. Often, for example, only the capture/playback portion of the testing tool ends up

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 on 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

needs as well as developing a list of tool-evaluation criteria. What does the system-engineering environment look like? What application development technologies are in use? These questions should be part of the tool-selection process, feeding into the development of evaluation criteria.

Other times, testers may be instructed to look for a testing tool to support the specific project they are working on, such as a Web project requiring Web-testing tools, while the organization as a whole produces desktop or client-server applications. It is best not to limit the automated test tool selection criteria to one single project, because obtaining such a tool may be an investment good for only that project, with the test tool becoming "shelfware" after the immediate project has been completed.

Once the type of testing tool has been decided on, criteria can be further defined. For example, if a tool is to be used across an entire organization, the test engineer must make sure it is compatible with as many operating systems, programming languages, and other aspects of the organization's technical environment as possible. The test engineer must review the organization's system-engineering environment by addressing the questions and concerns specified in this chapter and documenting the findings.

- *Identify the various system architectures.* During the survey of the system-engineering environment, the test engineer must identify the technical application architecture, including middleware, databases, and operating systems most commonly used within the organization or the particular project. The test engineer also must identify the languages used to develop the GUI for each application, along with all third-party add-ons used. Additionally, the test engineer must gain an understanding of the detailed architecture design, which can influence performance requirements. A review of prevalent performance requirements, including performance under heavy loads, intricate security mechanisms, and measures of availability and reliability for a system, is beneficial.

The specific selection criteria on any given effort will depend on the applicable system requirements of the applications under test. If it is not possible to use particular automated test tools across several projects or applications, then it might be necessary to develop the selection criteria relative to the most significant applications under test within the organization.

- *Determine whether more than one tool is required.* The test team responsible for implementing a test tool must be sure to include its own expectations in the selection criteria. As mentioned in [Item 32](#), a single tool generally cannot satisfy all test-tool interests and requirements within an organization. The tool chosen should at a minimum satisfy the more-immediate requirements. As the automated test-tool industry continues to evolve and grow, test-tool coverage of system requirements is likely to expand, and the chances may improve for a single tool to provide most of the functionality desired. Eventually, one tool may be ideal for most GUI testing; another tool may cover the bulk of performance testing; and a third tool may do most of what's needed for usability testing. For now, multiple test tools must be considered, depending on the testing phase, and expectations regarding the tools' capabilities must be managed.

It may be that no single tool is compatible with all operating systems and programming languages in a given environment. For example, when testing a LINUX client connecting to an IBM mainframe using 3270 sessions and a Java client, it will be difficult to find a tool that does LINUX, Java, and 3270 terminal emulation. More often, several tools are required to test the various technologies. In some cases, no tool on the market is compatible with the target environment, and it will be necessary to develop a custom automated-testing solution. Vendor-provided automated testing tools may have to be excluded from the test strategy in favor of a tool or test harness developed in-house. (For discussion of developing test harnesses, see [Item 37](#).)

- *Understand how data is managed by the application(s) under test.* The test team must understand the data managed by the target applications and define how the automated test tool supports data verification. A primary purpose of most applications is to transform data into meaningful information and present it to the user in graphical or text form. The test team must understand how this transformation is to be performed by the application so that test strategies can be defined to verify the correctness of the transformation.
- *Review help-desk problem reports.* When an application, or version of an application, is in operation, the test team can monitor help-desk trouble reports in order to study the most prevalent problems reported by users of the application. If a new version of the application is being developed, the test team can focus the testing effort on the most frequently reported problems of the operational system, including identifying a test tool that supports this area of testing, among other criteria.

- *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.

Item 35: Test the Tools on an Application Prototype

Given the broad spectrum of technologies used in software development, it is important to verify that testing-tool candidates function properly with the system being developed. The best way to accomplish this is to have the vendor demonstrate the tool on the application being tested. However, this usually is not possible, since the system under test is often not yet available during the tool-evaluation phase.

As an alternative, the development staff can create a system prototype for evaluating a set of testing tools. During the evaluation, it may be discovered that some tools function abnormally with the selected development technologies—for example, that the test team cannot properly operate the capture/playback or other important features of the tool.

Early in the development of the system, the prototype must be based on certain key areas of the system that will provide a somewhat representative sample of the technologies to be used in the system. The performance of tool-compatibility tests is especially important for GUI-test tools, because such tools may have difficulty recognizing custom controls in the application's user interface. Problems are often encountered with the calendars, grids, and spin controls that are incorporated into many applications, especially on Windows platforms. These controls, or widgets, were once called VBXs, then OCXs, and now are referred to as Active-X Controls in the Windows and Web interface worlds. They are usually written by third parties, and few test tool manufacturers can keep up with the hundreds of controls created by the various companies.

As an example of this problem, a test tool may be compatible with all releases of Visual Basic and PowerBuilder used to build the application under test; but if an incompatible third-party custom control is introduced into the application, the tool might not recognize the object on the screen. It may even be that most of the application uses a third-party grid that the test tool does not recognize. The test engineer must decide whether to automate testing of the unrecognized parts via a work-around solution, or to test these controls manually.

Such incompatibility issues may be circumvented if the test engineer evaluates and chooses the appropriate tool to match the project's needs right from the start.

The staff member evaluating test tools must also have the appropriate background. Usually, a technical background is required to ensure that the engineer can use the

tool against the application, or a prototype, in such a way as to guarantee that it functions properly with the required technologies. Moreover, ease-of-use, flexibility, and the other characteristics discussed in [Item 34](#) are best evaluated by a technical staff member.

There is no substitute for seeing a tool in action against a prototype of the proposed application. Any vendor can promise tool compatibility with many Web application technologies; but it is best to verify that claim. With the pace of technology "churn" in the software industry, no vendor exists that can claim compatibility with all technologies. It's up to each organization to ensure that the selected tool or tools meet its needs.

Chapter 8. Automated Testing: Selected Best Practices

To be most effective, automated software testing should be treated as a software development project. Like software application development, the test development effort requires careful analysis and design based on detailed requirements. This section covers using selected automated testing best practices—for example, how best to apply capture/playback technologies in a testing effort, and the pitfalls associated with using this type of automation technique. Automating regression tests, automated software builds, and smoke tests are additional automated testing practices discussed in this section.

The automated testing tools currently on the market are not always compatible with all environments or all testing needs. Some testing efforts may require custom test harnesses, either as a substitute for an off-the-shelf testing solution or to complement it. This section discusses when and why such tools should be developed, and the issues associated with them.

No single automation framework solution is best for all automation requirements. Automated test frameworks must be tailored to the organization's testing environment and to the task at hand. It is important that the appropriate automation test frameworks be chosen when developing an automated testing program.

Item 36: Do Not Rely Solely on Capture/Playback^[1]

^[1] Elfriede Dustin et al., "Reusable Test Procedures," Section 8.2.2 in *Automated Software Testing* (Reading, Mass: Addison-Wesley, 1999).

Functional testing tools (also called capture/playback tools) comprise just one of the numerous types of testing tools available. Capture/playback mechanisms can enhance the testing effort, but should not be the sole method used in automated testing. They have limitations, even when the best capture/playback automation techniques have been applied.

Capture/playback scripts must be modified after initial recording. Modification for functional testing focuses mostly on verifying tests through the GUI, also called **black-box** testing. To be most effective, black-box testing should be applied in combination with **gray-box** (internal-component based) and **white-box** (code-based) testing. Therefore, the most effective test automation involves additional automated testing tools and techniques beyond capture/playback.

Capture/playback is a feature of most automated test tools that enables the user to record keystrokes and mouse movements while interacting with an application. The keystrokes and mouse movements are recorded as a script, and can then be "played back" during test execution. While this method can be beneficial in specific situations, test scripts created using only capture/playback have significant limitations and drawbacks:

- *Hard-coded data values.* Capture/playback tools generate scripts based on user interactions including any data entered or received from the user interface.

Having data values "hard coded" right in the script can create maintenance problems later. The hard-coded values may render the script invalid if the user interface or other aspects of the application change. For example, input values, window coordinates, window captions, and other values can be captured within the script code generated during recording. These fixed values can become problematic during test execution if any of these values has changed in the application: The test script may interact improperly with the application or fail altogether. Another problematic hard-coded value is the date stamp. When the test engineer captures the current date in a test procedure, and then activates the script on a later date, it will fail, because the hard-coded date value contained in the script no longer matches the current date.

- *Non-modular, hard-to-maintain scripts.* Capture/playback scripts generated by the testing tool are usually not modular, and as a result are hard to maintain. For example, a particular URL in a Web application may be referenced in a number of test procedures, so it follows that a single change to this URL can render a significant number of scripts unusable, if those scripts are using the hard-coded URL. A modular development approach references, or **wraps up**, the URL in only one function. That function is then called by the various scripts that use it, so any changes to the URL need to be made in only one place.
- *Lack of standards for reusability.* One of the most important issues in test-procedure development is reusability. A specific test-creation standard that addresses the development of reusable automated test procedures can vastly improve test-team efficiency. Encapsulating the user-interface portion of the tests into modular, reusable script files that can be called from other scripts can substantially reduce the effort of maintaining scripts for a constantly changing user interface.

When creating a library of reusable functions, it is best to keep functionality such as data reading, writing, and validation, navigation, logic, and error checking separate, possibly in different script files. Automated-test development guidelines should be based on many of the same principles used for efficient software development. It is a good practice to follow the development-language guidelines that are closest to the development language of the scripts generated by the testing tool. For example, if the tool uses a C-like language, follow C development standards; if the tool uses a BASIC-like language, follow BASIC development standards.

It is evident that test scripts created exclusively using the capture/playback method for automated test development are difficult to maintain and reuse. In a few situations, basic scripts are useful; but in most cases, unless they modify the scripts after recording, test engineers would need to rerecord them many times during the test-execution process to account for any changes in the application under test. The potential advantages of using the capture/playback tool would thus be negated by the need to continually recreate the test scripts. This can lead to a high level of frustration among test personnel and general dissatisfaction with the automated testing tool.

To avoid the problems associated with unmodified capture/playback scripts, development guidelines for reusable test scripts should be created. Unmodified capture/playback scripting does not represent efficient test automation, and the exclusive use of capture/playback should be minimized.

Following these practices will improve the quality of capture/playback and automated testing. Still, it is important to realize that capture/playback tools, even when implemented using the best practices described in this chapter, are not the sole answer to test automation. Additional tools and techniques are required for effective testing to allow for more complete coverage and depth.

Item 37: Develop a Test Harness When Necessary

A *test harness* can be a tool that performs automated testing of the core components of a program or system. As used here, the term refers to code developed in-house that tests the underlying logic (back end) of an application. Off-the-shelf automated testing tools have their limitations, some of which are described in [Item 32](#). Additionally, automation through the user interface is often too slow to be practical with thousands of test cases, and can be hampered by constant changes to the user interface.

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.

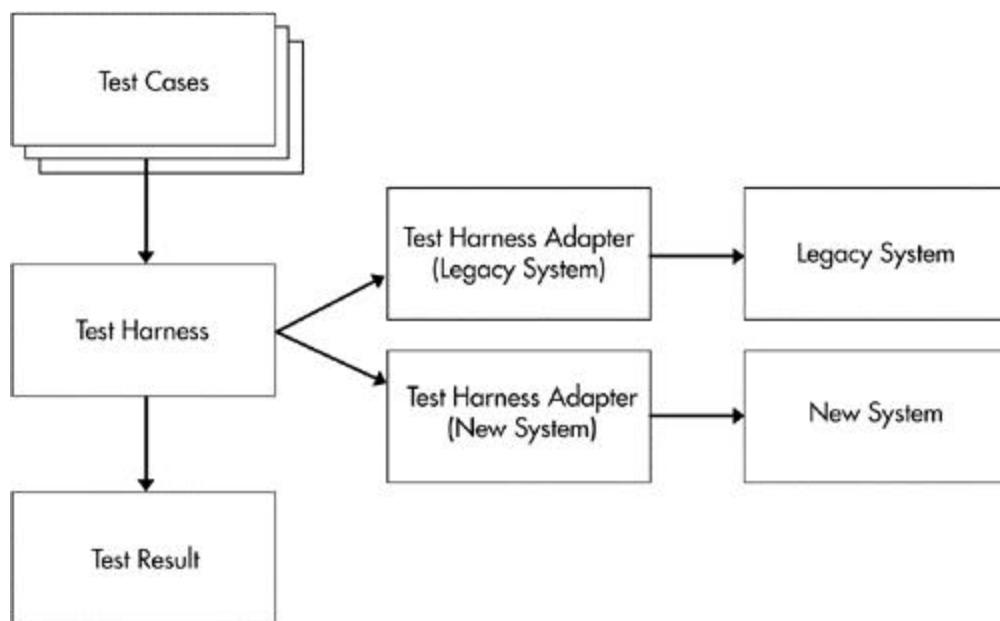
It is very time-consuming to develop and verify thousands of computational test cases by hand. In most cases, it would be far too slow to execute a large volume of test cases through the user interface. A more effective alternative may be to develop a test harness that executes test cases against the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

- *Data-driven framework.* **Data driven** means that data values are read from either spreadsheets or tool-provided data pools, rather than being hard coded into the test-procedure script. As mentioned in [Item 37](#), capture/playback tools create scripts that contain hard-coded values. Generally it is recommended that the test engineer edit these scripts, because hard-coded data values often prevent test procedures from being reused. Hard-coded values should be replaced with variables, and whenever possible read data from external sources, such as an ASCII text file, a spreadsheet, or a database.

The test team identifies the data scenarios, data elements, data value names, or data variable names to be used by the various test procedures. The key pieces of data added after a test-procedure script has been created via capture/playback are variable definitions and definitions of data sources (names and locations of files, spreadsheets, databases, and so on) from which the data values are to be read.

Externalizing input data also reduces data maintenance and increases flexibility. Whenever there is a need to execute a test procedure with a different set of data, the data source can be updated or changed as necessary.

- *Modular script development.* A common-software development technique used to increase maintainability and readability of source code is to divide the code into logical **modules**, each of which performs a separate part of the job. These modules can then be called from multiple points in a script, so their code does not have to be rewritten each time, or modified in each script. This is especially important when a change is made to one of the modules—the change needs to be made in only one location, which reduces maintenance costs.
- *Modular user-interface navigation.* An especially important function of test scripts that should be modularized is the navigation of the application's user interface. To increase the reusability of test scripts, a test engineer should have the script use a navigation method that is least likely to be affected by changes in the application's user interface. Navigation functions should be kept in a separate module within the test library. The test requirement should specify whether test procedures developed using a test-tool recorder navigate through the application through the use of tabs, keyboard accelerator keys (hot keys), or mouse clicks, and whether they capture the x-y coordinates (not recommended) or the object names of particular objects, such as window names or names of HTML objects on a Web page.

All of these choices will potentially be affected by design changes to the application. For example, if test-procedure scripts are developed using tab-key navigation to move between objects, the resulting test scripts are tab-order dependent. If the tab order changes, all of the scripts potentially must be corrected, which can be a monumental task. If accelerator keys are used in the script, but subsequently changed in the application, the scripts must likewise be changed or replaced. Mouse playback can be disrupted if a user-interface control is moved, or is replaced by a different type of control.

Introducing a new type of control affects almost all scripts negatively—no matter what navigation method is used.^[1] Modularizing the interaction with the control can help. Another way to avoid the problem is to navigate through an application by identifying fields by their object names or other unique identifiers assigned by the particular development tool. That way, a field can be moved to any position on the screen and the script will not be affected, because it navigates to the field by its name, not position. In this way, a recorded script can be made less susceptible to code changes and more reusable.

^[1] However, tools are becoming "smarter." For example, take Rational Software's RobotJ (see <http://www.rational.com/products/tstudio/robotj.jsp>, retrieved Sept. 8, 2002). Rational's new technology uses pattern matching. To validate changing data in interactive applications, the tester can set a range of acceptable data and replay tests with fewer problems than would occur with less sophisticated software tools.

Similarly, whenever possible, the test procedure should make use of a window's object name, which is most likely to remain static, instead of identifying the window by caption. Depending on how scripts are designed, window or object information can be maintained in a spreadsheet.

Navigation can be executed by using a driver that retrieves the various navigation information pieces from this spreadsheet. Some tools keep object information in a special GUI Map or Frame File. In such a case, it's not necessary to also put the information into a spreadsheet.

- *Libraries of reusable functions.* Test scripts typically perform many similar actions when acting upon the same product, and even with different products of the same type, such as Web applications. Separating these common

actions into shared script libraries usable by all test engineers in the organization can greatly enhance the efficiency of the testing effort.

- *Pre-built libraries.* Pre-built libraries of functions for certain testing tools are available on the Internet. For example, a group of test engineers using the WinRunner test tool created a library of reusable scripts and made them available to all test engineers via the Web.^[2] The library contains templates for test scripts, extensions of the test-script languages, and standardized test procedures for GUI testing using WinRunner, a test tool from Mercury Interactive. [Table 38.1](#) provides examples of reusable functions available for WinRunner.

^[2] Tilo Linz and Matthias Daigl, "How to Automate Testing of Graphical User Interfaces" (Mö hrendorf, Germany: imbus GmbH, n.d., retrieved Sept. 5, 2002 from http://www.imbus.de/engl/forschung/pie24306/gui/aquis-full_paper-1.3.html [text] and http://www.imbus.de/engl/forschung/pie24306/gui/gui_folien.html [accompanying slides]).

The library contains three basic sections: script templates, functions, and GUI checks. All three are designed to simplify the work of the test developer and to standardize the test results. The script templates provide frameworks for test engineers writing their own scripts. The script functions can be copied, then developed into complete test scripts or function sets. GUI checks are integrated into the WinRunner environment by starting the corresponding installation script; they can then be used just like the GUI checks built into WinRunner.

Table 38.1. Reusable Functions of WinRunner Scripts

Module	Description
Array	Functions for associative arrays
File	Functions for file access
General	Miscellaneous universal functions
Geo	Functions for checking the geometric alignment of objects
List	Functions for simplified access to list objects
Menu	Functions for handling menus
Standard	Standard tests
StdClass	Standard tests for object classes, such as radio buttons

String	Functions for handling character strings
Tab	Functions for accessing tab-card controls
Tooltips	Functions for accessing tool tips and toolbars

- *Version control.* Although not a script-development technique per se, version control is a vital part of any software project. All test scripts should be stored in a version-control tool, such as Microsoft Visual SourceSafe, to ensure that they are protected from simultaneous changes by different test engineers, and that a history of changes to each file is maintained. Without a version-control tool, it is all too easy for different script developers to change the same file, putting the developers in the error-prone position of needing to manually merge their changes. In addition, tracking which scripts work with which software builds is much easier when using a version-control tool, because sets of scripts can be labeled into groups that can be retrieved at any time in the future.

Item 39: Automate Regression Tests When Feasible

Regression testing determines whether any errors were introduced into previously working functionality while fixing an earlier error or adding new functionality to an application. Providing valued customers with lots of new and exciting features is of no benefit if the upgrade process manages to also break features they have been using successfully for the last several years. Regression testing should catch these newly introduced defects. However, despite its recognized importance, regression testing remains the testing phase often given the least attention in the planning of test activities.

An unplanned and manual approach can lead to inefficient and inadequate regression testing and is an inefficient use of resources. In developing a well-planned, effective regression test program, a number of questions must be answered:

- *When should regression tests be performed?* Regression testing is conducted for each change that may affect the functioning of previously baselined software. It includes all other previously run test procedures. Regression tests are also conducted when previously defective software that was corrected.
- *What should be included in the regression test?* Initially, regression testing should focus on high-risk functionality, and most-often executed paths. After these elements are tested, the more detailed functionality may be examined.

Regression testing can entail a specific selection of automated tests that exercise high-risk and areas of code that are potentially affected by repair of defects, or it can involve rerunning an entire suite of tests.

In the ideal test environment, all necessary test procedures eventually are automated as part of the regression test suite as new features become available for testing (see below).

- *How can the regression test suite be optimized and improved?* A test engineer can optimize a regression test suite by performing the following steps.
 1. Run the regression test set.
 2. If cases are discovered for which the regression test set ran correctly but errors surfaced later, include the test procedures that identified those bugs and other related scenarios in the regression test set. (Defects can be uncovered after the regression testing phase, for example during further system testing or beta testing, or through tech support calls).
 3. Repeat these steps, continually optimizing the regression test suite of scripts using quality measurements. (In this case, the relevant metric would be the number and types of test-procedure errors.)

In addition to following these steps, developers should provide information about the application areas directly affected by the implementation of new features and bug fixes and by other code changes. The information provided by the developers should be documented in a regression impacts matrix.

Testers can use such a matrix to evaluate the regression test suite, determining whether the planned tests provide adequate coverage of the affected areas, and adding or modifying test procedures accordingly.

The regression impact matrix should allow for coverage of other areas in addition to the section of software that has been changed. Errors may appear where they are least expected. In the classic example of a regression error, one module is changed, so it is tested during system test; but the change made in this module also affects another module that uses the data created by the changed module. If regression tests cover only the directly changed areas, many regression issues are missed.

One important point to remember is that in the iterative and incremental development model, regression tests are also developed incrementally and

must build on one another. Additionally, some regression tests must be changed or removed as parts of the applications change. Regression tests cannot be static: Modification of regression tests is necessary.

As discussed in [Item 22](#), the basic set of test procedures is based on requirements. Therefore, there must be a process to ensure that any requirement changes are directly communicated to the testing team. Modifications to existing specifications require corresponding modifications of the regression test procedures to account for the application's new functionality. The testing team must have a direct line of communication to the requirements team. No changes to the requirements should take place without the testing team's awareness and sign-off.

Regression test cases should be traceable to the original requirement. This way, when requirements are changed or removed, the related test cases that need updating will be known. Regression test phases are typically conducted by simply rerunning existing test suites, hoping these will catch all problems. However, it is important that new test cases be added as the system under test evolves, to provide depth and coverage that encompasses the software in its amended state.

- *Why automate regression tests?* When dealing with a large, complex system, it is possible to end up with thousands upon thousands of regression tests, as feature test suites from previous releases are added to the regression program for the next release. It is necessary, therefore, that regression test suites be automated, and then executed in a stable test environment that allows running all of the tests quickly without using a lot of resources. If those tests were not automated, regression testing could turn the testing effort into a long, tedious process. Worse, some regression tests might never be executed, leaving big gaps in the testing effort.

Additionally, manual execution of regression tests can be tedious, error prone, and lacking in coverage. Often, a test engineer feels that because a test was executed on a previous build or a fix was put in place in another area of the software, not the area currently under scrutiny, it is safe not to repeat that test. Other times, the production schedule might not allow for running an entire suite of manual regression tests. Because it is usually lengthy and tedious, and therefore prone to human error, regression testing should be done through an automated tool, a test harness, or a combination of the two.

Some tools allow groups of test procedures to be created. For example, in Robot a **test shell** groups several test procedures and plays them back in a specific, predefined sequence. Executing a test shell or **wrapper** containing a complete set of system test scripts is an efficient use of automated scripts during regression testing. Such a procedure allows the test engineer to create and execute a comprehensive test suite, and then store the results of the test in a single output log.

It is in the area of regression testing that automated test tools provide the largest return on investment. All of the scripts that have been developed previously can be executed in succession to verify that no new errors are introduced when changes are made to fix defects. Because the scripts can be run with no manual intervention, they can easily be run as many times as necessary to detect errors. Automated testing allows for simple repeatability of tests.

A significant amount of testing is conducted on the basic user-interface operations of an application. When the application is sufficiently operable, test engineers can proceed to test business logic in the application and behavior in other areas. Portions of the regression suite can be used to make a simple **smoke test**—a condensed version of a regression test suite, as described in [Item 40](#). With both manual and automated regression testing, the test teams repeatedly perform the same basic operability tests. For example, with each new release, the test team would verify that everything that previously worked still does.

Besides delaying the focus on other tests, the tedium of running regression tests manually exacts a very high toll on the testers. Manual testing can become delayed by the repetition of these tests, at the expense of progress on other required tests. Automated testing presents the opportunity to move more quickly and to perform more comprehensive testing within a given schedule. Automated regression testing frees up test resources, allowing test teams to turn their creativity and effort to more advanced test problems and concerns.

Automating regression testing also allows for after-hours testing, since most automated test tools provide the option for scripts to be executed at preset times without any user interaction. The test engineer can set up a test-script program in the morning, for example, to be executed by the automated test

tool at 11 p.m. that night. The next day, the team can review the test-script output log and analyze the results.

- *How does one analyze the results of a regression test?* When errors are observed for a system functionality that previously worked, the test team must identify other functional areas most likely to have an effect on the function in which the error occurred. Based upon the results of such analysis, greater emphasis can be placed on regression testing for the affected areas. Once fixes have been implemented, regression testing is again performed on the problem area, to verify closure of the open defects and that no new defects have been introduced.

The test team must also identify particular components or functions that produce a relatively greater number of problem reports. As a result of this analysis, additional test procedures and effort may need to be assigned to those components. If developers indicate that a particular functional area is now *fixed*, yet regression testing continues to uncover problems for the software, the test engineer must ascertain whether an environment issue is at play, or poor implementation of the software correction is at fault.

Test results can also be analyzed to determine whether particular test procedures are worthwhile in terms of identifying errors. If a regression test case has not caught any errors yet, there is no guarantee it will not catch an error at the 50th regression run. However, it is important to evaluate the effectiveness and validity of each test as new and changed functionality is introduced. Test-results analysis is also another way to identify the functions where the most defects have been uncovered, and therefore where further test and repair efforts should be concentrated.

The preceding points can help the testing team create an effective regression test program. As with all testing techniques and approaches, it is important to make sure that automated regression testing makes sense for the project at hand. If the system is constantly changing, for example, the benefit of automating regression testing can be low for the extensive maintenance cost of keeping the automated tests current for the ever-evolving system. Therefore, before automating a regression test suite, it is important to determine that the system is stable, that its functionality, underlying technology, and implementation are not constantly changing.

Item 40: Implement Automated Builds and Smoke Tests

Automated builds are typically executed once or twice per day (perhaps overnight), using the latest set of stable code. Developers can pull the components from the build machine each day, or rebuild locally themselves if they so desire.

A **smoke test** is a condensed version of a regression test suite. It is focused on automated testing of the critical high-level functionality of the application. Instead of having to repeatedly retest everything manually whenever a new software build is received, the smoke test is played against the software, verifying that the major functionality still operates properly.

Using an automated test tool, the test engineer records the test steps that would otherwise be manually performed in software-build verification. Typically, this verification is performed after the system is compiled, unit tested, and integration tested. It should be regarded as the "entrance criterion" for starting the testing phase for that build.

Implementing automated builds can greatly streamline the efforts of the development and configuration-management teams. Large software systems are typically built after-hours, so that a new build can be ready for the next day.^[1] Any problems encountered by the build process during its unattended operation must be resolved as soon as possible, however, so that development progress is not delayed. Some organizations use automated notification technologies that send emails or text pages to contact the appropriate personnel if a software build or regression test fails. Depending on the length of the build or regression test, this can be critical, allowing the problem to be corrected prior to the start of the next business day.

^[1] On large projects, sophisticated configuration-management or **process-management** tools are usually required to perform frequent builds. So-called **enterprise-level** tools go far beyond the simple file-based version control of simpler, less-costly tools. They allow a working version of the software to be built at almost any time, even when several developers are making widespread changes.

In addition to automating the software build, an automated smoke test can further optimize the development and testing environments. Software is always ready to use after the build, in a known good condition, because the smoke test will have automatically verified the build. If the unit-testing practices outlined in [Chapter 6](#)

are also part of the build process, the development and testing teams can be confident that the number of defects in the build has been significantly reduced by the time the software is ready to be tested.

If all of the aforementioned practices are followed, the typical software build sequence is as follows:

1. Software build (compilation), possibly including unit tests as described in [Chapter 6](#)
2. Smoke test
3. Regression test

If an error occurs at any point in the process, the entire cycle halts. After the problem is corrected, the build sequence is restarted at the appropriate point.

To build a smoke test, the test team first determines which parts of the application make up the high-level functionality, and then develops automated procedures for testing the major parts of the system. In this context, **major** refers to the basic operations that are used most frequently and can be exercised to determine if there are any large flaws in the software. Examples of major functions include logging on, adding records, deleting records, and generating reports.

Smoke tests may also comprise a series of tests verifying that the database points to the correct environment, the database is the correct version, sessions can be launched, all screens and menu selections are accessible, and data can be entered, selected and edited. When testing the first release of an application, the test team may wish to perform a smoke test on each available part of the system in order to begin test development as soon as possible without having to wait for the entire system to stabilize.

If the results meet expectations, meaning that the application under test has passed the smoke test, the software is formally moved into the test environment. Otherwise, the build should not be considered ready for testing.

Chapter 9. Nonfunctional Testing^[1]

^[1] Nonfunctional requirements do not endow the system with additional functions, but rather constrain or further define how the system will perform any given function.

The nonfunctional aspects of an application or system, such as performance, security, compatibility, and usability, can require considerable effort to test and perfect. Meeting nonfunctional requirements can make the difference between an application that merely performs its functions, and one that is well received by its end-users and technical support personnel, and is readily maintainable by system administrators.

It is a mistake to ignore nonfunctional system requirements until late in the application's development life cycle. The potential impact on the system architecture and implementation, which may need to change in order to satisfy these needs, necessitates consideration of nonfunctional requirements early in the development life cycle. Ideally, nonfunctional requirements are treated as important characteristics of the application. Therefore, proper documentation of nonfunctional aspects in the requirements can be vital to the development of a well-received application.

In [Chapter 1](#), it is recommended that nonfunctional requirements are associated with their corresponding functional requirements and defined during the requirements phase. If nonfunctional requirements are not defined, the requirements as a whole should be considered incomplete; unless defined early on, these important aspects of the system will probably not receive the appropriate attention until late in the development life cycle.

Item 41: Do Not Make Nonfunctional Testing an Afterthought^[1]

^[1] For detailed discussion of nonfunctional implementation, see Elfriede Dustin et al., *Quality Web Systems* (Boston, Mass.: Addison-Wesley, 2002).

Typically, the requirements, development, and testing efforts of a software project are so focused on the functionality of the system that the nonfunctional aspects of the software are overlooked until the very end. Certainly, if an application or system does not have the appropriate functionality, then it cannot be considered a

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-

party controls, such as ActiveX controls, for parts of the user interface. However, these controls may not be compatible with all platforms or installations. If an application is released without verifying the compatibility of its components with all user environments and an incompatibility is later identified, there may be no choice but to remove the third-party control and provide an alternate or custom implementation, requiring extra development and testing time. Meanwhile, the organization is unable to serve some users until the new version of the application is ready.

- *Concurrency in a multiuser client-server application.* Multiuser access to an application is one of the primary features of the client-server architecture. In client-server systems, concurrency, or simultaneous access to application data, is a major concern because unexpected behavior can occur if, for example, two users attempt to modify the same item of data. A strategy for handling data isolation is necessary to ensure that the data remains consistent in a multiuser environment. This strategy is typically implemented in the lower levels of the application, or in some cases in the user interface. If concurrency issues are not considered in the architecture and design of the application, components must later be modified to accommodate this requirement.

The risks of ignoring the nonfunctional aspects of an application until late in the development cycle must be considered by any organization, regardless of the type of software product being developed. Although it may be decided that some risks cannot be addressed immediately, it is better to know and plan for them than to be surprised when problems surface.

When planning a software project, consider the following nonfunctional risks:

- *Poor performance.* Poor application performance may be merely an inconvenience, or it may render the application worthless to the end user. When evaluating the performance risk of an application, consider the need for the application to handle large quantities of users and data. ([Item 42](#) discusses large data sets.) In server-based systems, such as Web applications, failure to evaluate performance can leave scaling capabilities unknown, and lead to inaccurate cost estimates, or lack of cost estimates, for system growth.
- *Incompatibility.* The proper functioning of an application on multiple end-user system configurations is of primary concern in most software-development activities. Incompatibilities can result in many technical-support calls and product returns. As with poor performance, compatibility

problems may be mere nuisances, or they may prevent the application from functioning at all.

- *Inadequate security.* Although security should be considered for all applications, Web-based software projects must be particularly concerned with this aspect. Improper attention to security can result in compromised customer data, and can even lead to legal action. If the application is responsible for sensitive customer information, security lapses can cause a severe loss of reputation, and possibly legal action.
- *Insufficient usability.* Inadequate attention to the usability aspects of an application can lead to a poor acceptance rate among end users, based on the perception that the application is difficult to use, or that it doesn't perform the necessary functions. This can trigger an increase in technical support calls, and can negatively affect user acceptance and application sales.

To avert such problems, it is very important to assess the nonfunctional aspects of an application as early as possible in the development life cycle.

It is equally important, however, not to overcompensate with too much attention to nonfunctional concerns. When addressing nonfunctional areas, there are typically tradeoffs. For example, consider application performance: Some software projects place so much emphasis on having well-performing applications that they underemphasize good design—two concepts that are often at odds. This can lead to code that is difficult to maintain or expand at a later point. Security provides another example of trade-offs: Architecting an application for maximum security can lead to poor application performance. It is critical that the risks of focusing on such nonfunctional issues as security or performance be weighed against the corresponding sacrifices to other aspects of the application.

It is most beneficial to the testing team if the nonfunctional requirements of an application are considered along with the functional aspects during the requirements phase of development. Requirements documents can specify performance or security constraints for each user interaction. This makes it possible not only for developers to ensure that the nonfunctional requirements are satisfied in the application's implementation, but also for the testing team to devise test cases for these concerns. Test procedures should include areas for each nonfunctional test for a given requirement, as described in [Item 21](#).

In addition to specific requirements-level nonfunctional information, it is also useful to have a set of **global** nonfunctional constraints that apply to all

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.

Often, a generic set of sample data will be created early in the project, and—because of time constraints—that data will be used throughout much, if not all, of the development and functional testing. Unfortunately, data sets such as these tend to be unrealistic, with insufficient size and variation, which may result in performance surprises when an actual customer, using much more data than contained in the sample data set, attempts to use the application. For this reason, it is recommended that testing teams, and perhaps development teams, use production-sized databases that include a wide range of possible data combinations and scenarios while the application is under development. Although it will take some extra time up-front to create these databases, it is worth the effort to avoid a crisis when the application goes into production.

Working with databases of realistic size also has secondary benefits to the project. Large databases can be difficult to handle and maintain. For example, building a large data set may require significant disk space and processor power. Storing data sets of this size may require additional resources, such as large disk drives or tapes. Depending on the application, large data sets may also lead to backup and recoverability concerns. If data is transferred across a network, or the Internet, bandwidth may also be a consideration for both the application and any additional processing performed on the data by support personnel. Working with realistic data sets as early as possible will help bring these issues to the surface while there is time, and when it is more cost-effective, to do something about them, rather than after the application or system is already in a production environment, when such issues can become very costly to address in terms of both budget and schedule.

There are a few ways to determine and create the appropriate data for a large data set, many of which are discussed in [Item 10](#). It is usually necessary to consult the development, product-management, and requirements teams to determine which parts of the data are the most critical, as well as which are most variable. For example, the database for an order-processing application may contain many tables full of information, but the most variable of these tables are those that store the details of orders and the customer information.

Once the necessary data elements are known, it may be possible to randomly generate (through a script or other program) a large number of records to increase the size of the database. Be aware, however, that randomly generated data may not be useful for *functional* testing, because it may not reflect real data scenarios that would be entered by a user. It is usually necessary to employ separate special-purpose data sets for functional testing and performance testing.

One way to gather realistic data is to poll potential or existing customers to learn about the data they use or plan to use in the application. This should be done during the requirements phase, while investigating how large the largest data set could be, what the average database size is, and so on. To use the previous example, it may be determined that the potential customers of the order-processing system have data sets that range from 1,000 to 100,000 orders. This information should be placed into the business case for the application, as well as the global nonfunctional requirements specifications. Once the decision is made to focus on supporting this range of order quantities, test databases can be created that reflect the smallest and largest anticipated sets of orders, as well as increments in between.

If the application being developed is a Web or client-server application, the production-class hardware that will ultimately run the software is typically more powerful than the hardware available for testing and development. A production-class machine will most likely have faster and multiple processors, faster disks, and more RAM. Due to cost constraints, most organizations will not provide production-class hardware for the development and testing environments.

This makes it necessary to extrapolate performance based on the differences between testing and production hardware platforms. Such estimations usually require baselining to determine a performance ratio between the platforms. Once an estimated performance multiple is determined, performance tests on the less-robust hardware can be used to verify the application's ability to handle large data with a reasonable amount of confidence. It should be kept in mind, however, that extrapolations are only rough estimates, since many factors can cause performance differences between two systems.

Item 43: Tailor Usability Tests to the Intended Audience

Usability testing is a difficult, but necessary, step toward delivering an application that satisfies the needs of its users. The primary goal of usability testing is to verify that the intended users of the application are able to interact properly with the application while having a positive and convenient experience. This requires examination of the layout of the application's interface, including navigation paths, dialog controls, text, and other elements, such as localization and accessibility. Supporting components, such as the installation program, documentation, and help system, must also be investigated.

Proper development and testing of an application for usability requires understanding the target audience and its needs. This requirement should be prominently featured in the application's business case and other high-level documents.

There are several ways to determine the needs of the target audience from a usability perspective:

- *Subject-matter experts.* Having staff members who are also experts in the domain area is a necessity in the development of a complex application. It can be a great asset to have such staff members counseling the requirements, development, and testing teams on a continual basis. Most projects, in fact, require several subject-matter experts, as opinions on domain-related rules and processes can differ.
- *Focus groups.* An excellent way to get end-user input on a proposed user interface is to hold focus-group meetings to get potential customers' comments on what they would like to see in an interface. Prototypes and screenshots are useful tools in focus-group discussions. To ensure adequate coverage, it is important to select focus groups that are representative of all types of end users of the product.
- *Surveys.* Although not as effective as subject-matter experts and focus groups, surveys can yield valuable information about how potential customers would use a software product to accomplish their tasks.
- *Study of similar products.* Investigating similar products can provide information on how the problem has been solved by other groups, in the same and different problem domains. Although user interfaces should not be blatantly copied from other products, it is useful to study how other groups or competitors have chosen to approach the user interface.
- *Observation of users in action.* Monitoring a user's interaction with an application's interface can provide a wealth of information about its usability. This can be done by simply taking notes while a user works with the application, or by videotaping a session for later analysis. Observing users in action enables the usability tester to see where the users stumble with the user interface and where they find it intuitively easy.

As with most nonfunctional requirements, early attention to usability issues can produce much better results than attempting to retrofit the application at a later time. For example, some application designs and architectures may not be suitable for the required user interface; it would be exceptionally difficult to recover if it were determined late in the process that the application architecture could not

support good usability. In addition, much time and effort goes into crafting the application's user interface, so it is wise to specify the correct interface as early as possible in the process.

An effective tool in the development of a usable application is the user-interface prototype. This kind of prototype allows interaction between potential users, requirements personnel, and developers to determine the best approach to the application's interface. Although this can be done on paper, prototypes are far superior because they are interactive and provide a more-realistic preview of what the application will look like. In conjunction with requirements documents, prototypes can also provide an early basis for developing test procedures, as described in [Item 24](#). During the prototyping phase, usability changes can be implemented without much impact on the development schedule.

Later in the development cycle, end-user representatives or subject-matter experts should participate in usability tests. If the application is targeted at several types of end users, at least one representative from each group should take part in the tests. Participants can use the software at the site of the development organization, or a pre-release version can be sent to the end user's site along with usability-evaluation instructions. End-user testers should note areas where they don't find the interface understandable or usable, and should provide suggestions for improvement. Since, at this stage in the development life cycle, large-scale changes to the application's user interface are typically not practical, requests for feedback should be targeted toward smaller refinements.

A similar approach can be taken for an application that is already in production. Feedback and survey forms are useful tools in determining usability improvements for the next version of the application. Such feedback can be extremely valuable, coming from paying customers who have a vested interest in seeing the application improved to meet their needs.

Item 44: Consider All Aspects of Security, for Specific Requirements and System-Wide

Security requirements, like other nonfunctional issues, should be associated with each functional requirement. Each functional requirement likely has a specific set of related security issues to be addressed in the software implementation, in addition to security requirements that apply system-wide. For example, the log-on requirement in a client-server system must specify the number of retries allowed, the action to take if the log on fails, and so on. Other functional requirements have

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

does not write confidential information to a log file that can be read by an intruder if the server is compromised.

It is important to also keep up with and install the latest vendor supplied patches for third-party products. This includes patches for Web servers, operating systems, and databases. Microsoft, for example, provides frequent security updates for its Internet Information Server (IIS) as security problems are found. Although it is usually not necessary to verify that the supplied security patches correct the advertised problem, it is important to have personnel monitoring the status of security patches for the third-party products used in the system. As with all updates, the system should be regression tested after the installation of any patch to verify that new problems have not been introduced.

If the security risk associated with an application has been determined to be substantial, it is worth investigating options for outsourcing security-related testing. E-commerce sites, online banking, and other sites that deal in sensitive customer data should regard security as a high priority, because a break-in could mean disaster for the site and the organization. There are many third-party security-testing firms that will investigate the site, and possibly its implementation, for security flaws using the latest tools and techniques. Outsourcing in combination with in-house testing of security-related nonfunctional requirements is an excellent way to deliver a secure application.

Item 45: Investigate the System's Implementation To Plan for Concurrency Tests

In a multiuser system or application, **concurrency** is a major issue that the development team must address. Concurrency, in the context of a software application, is the handling of multiple users attempting to access the same data at the same time.

For example, consider a multiuser order processing system that allows users to add and edit orders for customers. Adding orders is not a problem—since each new order generates a discrete record, several users can simultaneously add orders to the database without interfering with one another.

Editing orders, on the other hand, can result in concurrency problems. When a user opens an order to edit it, a dialog is displayed on the local machine with information about that order. To accomplish this, the system retrieves the data from the database and stores it temporarily in the local machine's memory, so the

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.

- *No concurrency protection: "last one wins."* The simplest model of all, this approach does not attempt to protect users from editing the same data. If two users open a record and make changes, the second user's changes will overwrite the first's in a "last-one-wins" situation. The first user's changes will be lost. Also, depending on how the system is implemented, this approach could lead to data corruption if two users attempt to save at the same moment.

The way a software application deals with concurrency can affect the performance, usability, and data integrity of the system. Therefore, it is important to design tests to verify that the application properly handles concurrency, following the concurrency model that has been selected for the project.

Testing application concurrency can be difficult. To properly exercise the system's concurrency handling techniques, timing is an issue. Testers must simulate two attempts to read or write data at the same instant to properly determine whether the system properly guards the data, whether by the establishment of locks in the case of pessimistic locking, or by checking for an update prior to saving data, as in the case of optimistic locking.

A combination of manual and automated techniques can be used for concurrency testing. Two test engineers can attempt to access the data through the user interface on separate machines, coordinating verbally to ensure that they access the data simultaneously. Automation can be used to exercise the system with higher volume, and more-reliable instantaneous access, possibly below the user-interface layer, to uncover any concurrency related issues. Knowledge of the system's implementation is key to designing and executing the appropriate tests.

Concurrency testing becomes more complicated when the same data can be updated via different interfaces or functions in the system. For example, if one user is editing a record, and another user deletes the record, the system must enforce the appropriate concurrency control. If pessimistic locking is in place, the system should not allow the record to be deleted while it is being edited. In fact, all functions that could potentially access the record should be tested to ensure that they cannot modify it while it is being edited.

The system-wide nonfunctional specifications should spell out how the system is to handle concurrency issues in general. If there are any deviations for individual system functions, these should be noted in the requirement documentation for those particular functions. For example, the system-wide specification may call for

optimistic concurrency, while one particularly sensitive area of the system requires pessimistic concurrency. That fact must be noted in the documentation for the requirements in that specific system area.

Test procedures necessarily vary depending upon the concurrency approach taken by the system, and, as described in the preceding example, within parts of the system. It is essential that the testing team understand the concurrency handling approaches used in the system, because it is difficult to implement appropriate tests without this knowledge. Each concurrency model has nuances that must be accounted for in the test procedures.

The following list details some test-procedure concerns for each of the different concurrency models:

- *Pessimistic*. Because the application forcibly restricts users from accessing data while that data is being edited, the main concern in testing a pessimistic concurrency scheme is to verify that locks on records are properly acquired, released, and enforced in all areas of the application that may attempt to update the record. There are three primary areas of concern here:
 - *Lock acquisition*. Because only one user at a time can enter an update state with a particular record or item of data, it is critical that the system properly assign the lock to the user who requests it first. Improper implementation of lock-acquisition code could lead multiple users to believe they have the lock, in which case they will probably encounter errors or unexpected behavior when they attempt to save the data. Lock acquisition can be tested by having two users attempt to enter an edit state at the same instant, or with heavy volume. For the latter, a script may be employed to generate, say, 1,000 or more simultaneous requests to edit data and verify that only one is allowed to proceed.
 - *Lock enforcement*. Once a user has a lock, the system must ensure that the data cannot be modified in any way, including updates and deletions, by any other user. This is typically accomplished by having one user hold a record open—enter edit mode and stay there—while other users attempt to edit, delete, or otherwise update the data in all areas of the application. The system should reject all other users' attempts at updating the data.
 - *Lock release*. Another tricky area within pessimistic concurrency is lock release. Testers must verify that the system successfully makes the record available to other users once it has been released by the

editing user, whether that user has chosen to go through with the update or cancel it. An important aspect of lock release is error handling—what happens if the user who holds the lock encounters an error, such as a client system crash. Is the lock orphaned (left in place with no way to release it)? The ability of the system to recover from failure to release a lock is an important concern.

- *Optimistic.* The optimistic concurrency model is somewhat less complex to implement. Because all users are allowed to retrieve and edit the data, the point of update is the only concern. As mentioned previously, a combination of manual and automated techniques offers the best approach to testing optimistic concurrency. In the manual approach, two testers edit the data and then attempt to save it at precisely the same time. The first user should get a successful update, while the second is sent a message that another user has updated the record, and to reload the record and make the changes again.

In addition to manual testing, an automated, volume-oriented approach should be used. Using a script that attempts hundreds or thousands of updates at the same instant can ensure that the optimistic locking mechanism allows only one user at a time to update the record. The rest should receive error messages indicating that the record cannot be saved because another user has updated it.

As with pessimistic concurrency, testers must verify that optimistic concurrency is enforced everywhere the data can be modified, including any possibilities for modification through different areas of the user interface.

- *No concurrency control.* The most basic, and error-prone, approach is a system with no concurrency control. Testing such a system is similar to testing optimistic concurrency, with the exception that all users should achieve successful updates regardless of the order in which they request the updates. Again, a combination of manual and automated testing techniques should be used, but with an emphasis on data integrity, an area for concern when there is no concurrency control. Testers should also ensure that update errors, such as when a record is deleted while another user is attempting to update it, are properly handled.

Investigating the system's implementation plan will help the test team determine the correct type of concurrency testing to perform. Because concurrency is a common problem area in most multiuser systems, proper testing of this area is essential.

Item 46: Set Up an Efficient Environment for Compatibility Testing

Testing an application for compatibility can be a complex job. With so many different operating system, hardware, and software configurations available, the number of possible combinations can be high. Unfortunately, there really isn't any way around the compatibility test problem—software either runs properly on any given system or it doesn't. An appropriate environment for compatibility testing can, however, make this process more efficient.

The business case for the system being developed should state all of the targeted end-user operating systems (and server operating systems, if applicable) on which the application must run. Other compatibility requirements must be stated as well. For example, the product under test may need to operate with certain versions of Web browsers, with hardware devices such as printers, or with other software, such as virus scanners or word processors.

Compatibility may also extend to upgrades from previous versions of the software. Not only must the system itself properly upgrade from a previous version, but the data and other information from the previous version must be considered as well. Is it necessary for the application to be backward compatible with data produced by a previous version? Should the user's preferences and other settings be retained after the upgrade? These questions and others must be answered in comprehensively assessing the compatibility situation.

With all the possible configurations and potential compatibility concerns, it probably is impossible to explicitly test every permutation. Software testers should consider ranking the possible configurations in order, from most to least common for the target application. For example, if most users run Windows 2000 with MS Internet Explorer 6.0 on dial-up connections, then that configuration should be at the top of list, and must be emphasized in the compatibility tests. For some of the less-common configurations, time and cost constraints may limit the testing to quick smoke tests.

In addition to selecting the most-important configurations, testers must identify the appropriate test cases and data for compatibility testing. Unless the application is small, it is usually not feasible to run the entire set of possible test cases on every possible configuration, because this would take too much time. Therefore, it is necessary to select the most representative set of test cases that confirms the application's proper functioning on a particular platform. In addition, testers also

need appropriate test data to support those test cases. If performance is a compatibility concern for the application—for example, if it works with an open-ended amount of data—then testers will also need data sets of sufficient size.

It is important to continually update the compatibility test cases and data as issues are discovered with the application—not only while it is in development, but also once it is released. Tech-support calls can be a good source of information when selecting tests to add to the compatibility test suite. The calls may also provide an indication whether the most appropriate configurations were selected for compatibility testing, possibly leading to a revised set of configurations with different priority rankings.

A beta-test program is another potential source of information on end-user configurations and compatibility issues. A widely distributed beta test can provide a large set of data regarding real end-user configurations and compatibility issues prior to the final release of the application.

Managing the compatibility test environment can be a major undertaking. To properly test the application, it is best to precisely recreate the end-user environment. This is accomplished by installing actual operating system, hardware, and software configurations and executing the selected set of test procedures on each installation. As discussed previously, however, users may have many combinations of configurations. It is usually not cost effective to purchase and install all the machines needed to represent every potential test configuration; moreover, reinstalling even one or a few machines from the ground up for each new test is very costly. Therefore, a different approach must be taken.

A practical and cost-effective approach involves the use of removable hard drives in combination with a partition-management tool. This allows a small number of machines to potentially run a large number of configurations. The tester simply places the correct drive in the bay, reboots, and selects the desired configuration. There are many partition management tools on the market that can be used in this context, such as Partition Magic.

An alternative approach would be to use a drive-imaging program, such as Symantec Ghost, to create image files of the necessary configurations. These images can be used later to recreate the configuration on a target machine. The downside with this approach is that it can take a significant amount of time to recreate the configuration from the image, depending on the size of the installation.

In addition, the image files can be quite large, so a way of managing them must be established.

Regardless of the technique used to manage the installations, once the proper configuration environment has been booted on the target machine, configuration tests can then be executed and analyzed to determine the application's compatibility on that platform.

In compatibility testing, the manner in which the application is installed on the target configuration is important. It is critical that the application be installed in the exact way the end user will install it, using the same versions of the components and the same installation procedures. This is best accomplished when an installation program is made available early in the development effort, and the software is provided to the testing team in an installable form, rather than a specialized package created by the development team. If a specialized or manual method of installation is used, the test environment may not mirror the end user's environment, and compatibility-test results may be inaccurate. It is also critical that no development tools be installed on the compatibility-test platform, because they may "taint" the environment so that it is not an exact reflection of the real operating conditions.

Chapter 10. Managing Test Execution

Test execution is the phase that follows after everything discussed to this point. With test strategies, test planning, test procedures designed and developed, and the test environment operational, it is time to execute the tests created in the preceding phases.

Once development of the system is underway and software builds become ready for testing, the testing team must have a precisely defined workflow for executing tests, tracking defects found, and providing information, or metrics, on the progress of the testing effort.

The items that follow discuss the participation of many teams throughout the organization, including testing, development, product management, and others, in test execution. It is the job of all of these teams to ensure that defects are found, prioritized, and corrected.

Item 47: Clearly Define the Beginning and End of the Test-Execution Cycle

Regardless of the testing phase, it is important to define the **entrance criteria** (when testing can begin) and the **exit criteria** (when testing is complete) for a software test execution cycle.

The entrance criteria describe when a testing team is ready to start testing a specific build. In order to accept a software build during system testing, various criteria should be met. Some examples:

- All unit and integration tests have been executed successfully.
- The software builds (compiles) without any issues.
- The build passes a smoke test, as discussed in [Item 40](#).
- The build has accompanying documentation (**release notes**) describing what's new in the build and what has been changed.
- Defects have been repaired and are ready for retesting. (See [Item 49](#) for a discussion of the defect-tracking life cycle.)
- The source code is stored in a version-control system.

Only after the entrance (or acceptance) criteria have been met is the testing team ready to accept the software build and begin the testing cycle.

Just as it is important to define the entrance criteria for a testing phase, exit criteria must also be developed. Exit criteria describe when the software has been adequately tested. Like entrance criteria, they depend on the testing phase. Because testing resources are finite, the test budget and number of test engineers are limited, and deadlines approach quickly, the scope of the test effort must have its limits. The test plan must indicate clearly when testing is complete: If exit criteria are stated in ambiguous terms, the test team cannot determine the point at which the test effort is complete.

For example, a test-completion criterion might be that all defined test procedures, which are based on requirements, have been executed successfully without any significant problems, meaning that all high-priority defects have been fixed by the development staff and verified through regression testing by a member of the test team. Meeting such a criterion, in the context of the other practices discussed throughout this book, provides a high level of confidence that the system meets all requirements without major flaws.

Following are example statements that might be considered as part of exit criteria for an application:

- Test procedures have been executed to determine that the system meets the specified functional and nonfunctional requirements.
- All level 1, level 2, and level 3 (showstopper, urgent, and high-priority) software problems documented as a result of testing have been resolved.
- All level 1 and level 2 (showstopper, urgent) software problems reported have been resolved.
- All level 1 and level 2 (showstopper, urgent) software problems documented as a result of testing have been resolved, and 90 percent of reported level-3 problems have been resolved.
- Software may be shipped with known low-priority defects (and certainly with some unknown defects).

Notwithstanding the exit criteria that have been met, software will be only as successful as it is useful to the customers. Therefore, it is important that user-acceptance testing is factored into the testing plan.

Developers must be made aware of system-acceptance criteria. The test team must communicate the list of entrance and exit criteria to the development staff early on, prior to submitting the test plan for approval. Entrance and exit testing criteria for

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.

Without a separate test environment, the testing effort is likely to encounter several of the following problems:

- *Changes to the environment.* A developer may apply a fix or other change to the development configuration, or add new data, without informing the testing team. If a tester has just written up a defect based on the environment, that defect may now not be reproducible. The defect report probably must be marked "closed—cannot be reproduced," resulting in wasted development and testing resources. Additionally, unless the developers inform the testers of any changes to existing code made in the course of the day, new defects will be missed. The testers will not know to retest the changed code.
- *Version management.* It's difficult to manage versions when developers and testers share the same environment. Developers may need to integrate new features into the software, while testers need a stable version to complete testing. Conversely, developers usually need somewhere to run the latest build of the software other than their own desktop machines. If testers are sharing the development environment, its resources are less available for use by developers.
- *Changes to the operating environment.* Testing may require reconfiguration of the environment to test the application under various conditions. This creates a large amount of turbulence in the environment, which may impinge upon developer activities. For example, machines may be rebooted, or taken off-line for several days so changes can be made to the installed software or hardware.

While budget constraints often do not allow for a separate testing environment, it is possible that, with certain cost-saving measures, a separate test environment may be created affordably. Some examples:

- *Reduced performance and capacity.* In a testing environment, it may not be necessary to have high-performance machines with massive disk and memory capacity. Smaller machines can be used for most test activities, with results being extrapolated when necessary to estimate performance on larger hardware platforms. This is recommended only if the budget doesn't allow for production-sized test machines, however, as extrapolation is an imprecise way to gauge performance, producing only a ballpark estimate of actual performance in the production environment. When exact numbers are required, performance figures should be derived on production-class hardware.

- *Removable disks.* Testing several configurations does not necessarily require several machines or a complete reinstall for each configuration. A removable disk with multiple partitions allows a single machine to host a multitude of software configurations, saving money and time. See [Item 46](#) for more information on using removable disks for compatibility testing.
- *A shared test lab.* Share the costs of the testing environment by using a test lab for several software projects. The lab should be designed to be easily reconfigured for use with various testing engagements, not so specifically that it works for only one project.

Item 49: Implement a Defect-Tracking Life Cycle

The defect tracking life cycle is a critical aspect of the testing program. It is important to evaluate and select an adequate defect-tracking tool for the system environment. Once the tool has been acquired, or developed in-house, a defect-tracking life cycle should be instituted, documented, and communicated. All stakeholders must understand the flow of a defect, from the time it has been identified until it has been resolved.

After a defect is fixed, it must be retested. In the defect-tracking system, the element would be labeled as being in **retest status**. The development manager must be aware of and comply with the process for tracking the status of a defect. If the manager did not follow the tracking procedures, how would the test team know which defects to retest?

Test engineers must document the details of a defect and the steps necessary to recreate it, or reference a test procedure if its steps expose the problem, to help the development team pursue its defect-correction activities. Defects are commonly classified based on priority in order that higher-priority defects may be resolved first. Test engineers must participate in change-control boards when those boards are involved in reviewing outstanding defect reports. Once the identified defects have been corrected in a new software build, test engineers are informed via the defect-tracking tool. Testers can then refer to the tracking tool to see which defects are identified as being ready for retesting. Ideally, the record for the fixed defect also contains a description of the fix and what other areas of the system it may affect, helping testers determine which potentially affected areas to also retest.

Each test team must perform defect reporting using a defined process that includes the following steps.

1 Analysis and Defect Record Entry

The process should describe how to evaluate unexpected system behavior. First, false results must be ruled out. Sometimes a test can produce a false negative, with the system under test behaving as intended but the test incorrectly reporting an error. Or the test (especially when using a testing tool) can produce a false positive, where the system is reported to pass the test when in fact there is an error. The tester must be equipped with specific diagnostic capabilities to determine the accuracy of the test procedure's output.

Assuming the tester's diagnosis determines that the unexpected system behavior is an actual defect (not a false positive, false negative, duplicate of a defect already reported, etc.) a software problem or defect report typically is entered into the defect-tracking tool by a member of the testing team, or by a member of another team who is tasked to report defects.

Following is an example of attributes to be included in documentation of a defect:

- *Subject heading.* The subject heading should start with the name of the system area in which the defect was found—for example, Reports, Security, User Interface, or Database.
- *Version of build under test.* The version number of the build that is being tested. The version should differentiate between production and test builds of the software. Examples: Build #3184, Test_Build_001.
- *Operating system and configuration.* List the operating system and other configuration elements (UNIX, Win98, Win95, WinXP, WinME, Win2000; I.E. 5.5, I.E. 6.0, Netscape 6, and so on) of the test in which the defect was discovered.
- *Attachments.* Examples of attachments are screenshots of the error and printouts of logs. Attachments should be placed in a central repository accessible to all stakeholders.
- *Reproducibility.* Sometimes a defect only occurs intermittently. Indicate whether it's always reproducible, appears inconsistently, or is not reproducible at all.
- *Step-by-step instructions for reproducing the error.* If the error is reproducible, include any information necessary for a developer to reproduce the defect, such as specific data used during the test. The error itself should be clearly described. Avoid vague language such as, "The program displayed incorrect information." Instead, specify the incorrect

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

many priority levels makes it difficult to classify defects properly, as the levels begin to lose their distinctiveness and blend together.

3 Reoccurrence

The process should define how to handle reoccurring issues. If a defect has been resolved before but continues to reappear, how should it be handled? Should the earlier defect report be reopened, or should a new defect be entered? It is generally advisable to review the earlier defect report to study the implemented fix, but open a new defect report with a cross-reference to the "closed, but related" defect. This style of referencing provides an indication of how often defects are reintroduced after they have been fixed, perhaps pointing to a configuration problem or an issue on the development side.

4 Closure

Once a defect has been corrected, determined to be a duplicate, or deemed not a defect, the report can be closed. However, it should remain in the defect database. No one should delete defect reports from the tracking system. This ensures that all defects and their histories are properly tracked. In addition to the loss of historical information that could prove valuable in the future, deleting records from the defect-tracking system can destroy the continuity of the numbering system. Furthermore, if people have made hard copies of the defect reports, they will not agree with the tracking system file. It is much better practice to simply close the defect by marking it "closed—works as expected," "closed—not a defect," or "closed—duplicate."

It may be useful to have a rule regarding partially corrected defects, such as, "If a defect is only partially fixed, the defect report cannot be closed as fixed." Usually, this should not be an issue. If defect reports are sufficiently detailed and granular, each report will contain only one issue, so there will be no "partially fixed" condition. If multiple problems are encountered, each should be entered in its own defect report, regardless of how similar the problems may be. This way, instead of listing one complex issue as "partially fixed," the elements that are fixed and the elements that are outstanding will each have their own records.

Once a defect has been corrected and unit tested to the satisfaction of the software-development team, the corrected software code is stored in the software configuration management system. At some point, the conclusion is made that an acceptable number of defects has been corrected, or in some cases a single critical

defect has been fixed, and a new software build is created and given to the testing team.

Defects are discovered throughout the software development life cycle. It is recommended that the test team generate software problem reports and classify each defect according to the life-cycle phase in which it was found. [Table 49.1](#) provides example categories for software problem reports.

Table 49.1. Categories of Defects and Associated Testing Life Cycle Elements

Category	Applies if a problem has been found in:	System	Software	Hardware
A	System development plan	✓		
B	Operational concept	✓		
C	System or software requirements		✓	✓
D	Design of the system or software		✓	✓
E	The coded software (application under test)		✓	
F	Test plans, cases and procedures report		✓	✓
G	User or support manuals		✓	✓
H	The process being followed on the project		✓	✓
I	Hardware, firmware, communications equipment			✓
J	Any other aspect of the project	✓	✓	✓

When using a defect-tracking tool, the test team must define and document the defect life-cycle model, also called the **defect workflow**. In some organizations, the configuration management group or process engineering group is responsible for the defect workflow, while in other organizations, the test team is responsible. Following is an example of a defect workflow.

Defect Workflow

1. When a defect report is initially generated, its status is set to "New."

2. Various teams (test, documentation, product management, development, training, or requirements team) have been designated as being able to open an issue. They select the type of issue:
 - Defect
 - Change Request
 - Enhancement Request
3. The originator then selects the priority of the defect, subject to modification and approved by the change-control board (CCB).
4. A designated party, such as the software manager or the CCB, evaluates the defect, assigns a status, and may modify the issue type and priority.

The status "*Open*" is assigned to valid defects. The status "*Closed*" is assigned to duplicate defects or user errors. Only the testing team should be allowed to close a defect. The reason for "*closing*" the defect must be documented:

- Closed: Not Reproducible
- Closed: Duplicate
- Closed: Works as expected

The status "*Enhancement*" is assigned if it's been determined that the reported defect is in fact an enhancement request.

5. If the status is determined to be "*Open*," the software manager (or other designated person) assigns the defect to the responsible stakeholder, and sets the status to "*Development*."
6. Once the developer begins working on the defect, the status can be set to "*In Development*."
7. Once the developer believes the defect is fixed, the developer documents the fix in the defect-tracking tool and sets the status to "*Fixed*." The status can also be set to "*Works As Expected*" or "*Cannot Be Reproduced*" if the software is functioning properly. At the same time, the developer reassigns the defect to the originator.
8. Once a new build is created, the development manager moves all fixed defects contained in the current build to "*re-test*" status.
9. The test engineer re-tests those fixes and other affected areas. If the defect has been corrected with the fix, the test engineer sets the status to "*Closed-Fixed*." If the defect has not been corrected with the fix, the test engineer

sets the status to "*Re-Test Failed*." If the defect is fixed, but the fix breaks something else in the software a new defect is generated. The cycle repeats.

Item 50: Track the Execution of the Testing Program

Everyone involved in a software project wants to know when testing is completed. To be able to answer that question, it is imperative that test execution be tracked effectively. This is accomplished by collecting data, or metrics, showing the progress of testing. The metrics can help identify when corrections must be made in order to assure success. Additionally, using these metrics the test team can predict a release date for the application. If a release date has been predetermined, the metrics can be used to measure progress toward completion.

Progress metrics are collected iteratively during the stages of the test-execution cycle. Sample progress metrics include the following:

- *Test Procedure Execution Status (%) = executed number of test procedures / total number of test procedures.* This execution status measurement is derived by dividing the number of test procedures already executed by the total number of test procedures planned. By reviewing this metric value, the test team can ascertain the number of percentage of test procedures remaining to be executed. This metric, by itself, does not provide an indication of the quality of the application. It provides information only about the progress of the test effort, without any indication of the success of the testing.

It is important to measure the test procedure *steps* executed, not just the number of test *procedures* executed. For example, one test procedure may contain 25 steps. If the tester successfully executes Steps 1 through 23, and then encounters a showstopper at Step 24, it is not very informative to report only that the entire test procedure failed; a more useful measurement of progress includes the number of steps executed. Measuring test-procedure execution status at the step level results in a highly granular progress metric.

The best way to track test procedure execution is by developing a matrix that contains the identifier of the build under test, a list of all test procedure names, the tester assigned to each test procedure, and the percentage complete, updated daily and measured by the number of test-procedure steps executed successfully versus total number of test-procedure steps planned.

Many test-management or requirements-management tools can help automate this process.

- *Defect Aging* = *Span from date defect was opened to date defect was closed.* Another important metric in determining progress status is the turnaround time for a defect to be corrected, also called **defect aging**. Defect aging is a high-level metric that verifies whether defects are being addressed in a timely manner. Using defect-aging data, the test team can conduct a trend analysis. For example, suppose 100 defects are recorded for a project. If documented past experience indicates that the development team can fix as many as 20 defects per day, the turnaround time for 100 problem reports may be estimated at one work-week. The defect-aging statistic, in this case, is an average of five days. If the defect-aging measure grows to 10–15 days, the slower response time of the developers making the corrections may affect the ability of the test team to meet scheduled deadlines.

Standard defect-aging measurement is not always applicable. It sometimes must be modified, depending on the complexity of the specific fix being implemented, among other criteria.

If developers don't fix defects in time, it can have a ripple effect throughout the project. Testers will run into related defects in another area, creating duplication when one timely fix might have prevented subsequent instances. In addition, the older a defect becomes, the more difficult it may be to correct it, since additional code may be built on top of it. Correcting the defect at a later point may have much larger implications on the software than if it had been corrected when originally discovered.

- *Defect Fix Time to Retest* = *Span from date defect was fixed and released in new build to date defect was retested.* The defect fix retest metric provides a measure of whether the test team is retesting corrections at an adequate rate. If defects that have been fixed are not retested adequately and in a timely manner, this can hold up progress, since the developer cannot be assured that a fix has not introduced a new defect, or that the original problem has been properly corrected. This last point is especially important: Code that is being developed with the assumption that earlier code has been fixed will have to be reworked if that assumption proves incorrect. If defects are not being retested quickly enough, the testing team must be reminded of the importance of retesting fixes so developers can move forward, knowing their fixes have passed the test.

- *Defect Trend Analysis = Trend in number of defects found as the testing life cycle progresses.* Defect trend analysis can help determine the trend in the identification of defects. Is the trend improving (i.e., are fewer defects being found over time) as the system testing phase is nearing completion, or is the trend worsening? This metric is closely related to "Newly opened defects." The number of newly opened defects should decline as the system-testing phase nears the end. If this is not the case, it may be an indicator of a severely flawed system.

If the number of defects found increases with each subsequent test release, assuming no new functionality is being delivered and the same code is being tested (changed only by the incorporation of code fixes), several possible causes may be indicated, such as:

- improper code fixes for previous defects
- incomplete testing coverage for earlier builds (new testing coverage discovers new defects)
- inability to execute some tests until some of the defects have been fixed, allowing execution of those tests, which then find new defects
- *Quality of Fixes = Number of errors (newly introduced or reoccurring errors in previously working functionality) remaining per fix.* This metric is also referred to as the **recurrence ratio**. It measures the percentage of fixes that introduce new errors into previously working functionality or break previously working functionality. The value obtained from this calculation provides a measure of the quality of the software corrections implemented in response to problem reports.

This metric helps the test team determine the degree to which previously working functionality is being adversely affected by software corrections. When this value is high, the test team must make developers aware of the problem.

- *Defect Density = Total number of defects found for a requirement / number of test procedures executed for that requirement.* The defect-density metric is the average number of defects found per test procedure in a specific functional area or requirement. If there is a high defect density in a specific functional area, it is important to conduct a causal analysis using the following types of questions:
 - Is this functionality very complex, and therefore expected to exhibit high defect density?

- Is there a problem with the design or implementation of the functionality?
- Were inadequate resources assigned to the functionality, perhaps because its difficulty was underestimated?

Additionally, when evaluating defect density, the relative priority of the defect must be considered. For example, one application requirement may have as many as 50 low-priority defects and still satisfy acceptance criteria. Another requirement might have one open high-priority defect that prevents the acceptance criteria from being satisfied.

These are just few of the metrics that must be gathered to measure test-program execution; many others are available. The metrics described in this Item are a core set to track in order to identify any need for corrective activity, to highlight risk areas, and ultimately, to enable successful execution of the testing program.

The Value of Mobile Application Testing

White Paper

National Software Testing Labs



www.nstl.com

Table of Contents

I.	Executive Summary	3
II.	Challenges for Mobile Application Developers	5
A.	Time to Market	6
B.	Different Operators, Different Requirements	7
C.	The Role of Testing.....	8
D.	Handset Inventory and Distribution.....	9
III.	Challenges for Operators.....	9
A.	Escalating Support Costs	10
B.	Application Quality Control	10
C.	Testing Across Distinct Networks.....	11
IV.	Challenges for Consumers	11
A.	Understanding the Mobile Language	12
B.	Handset Compatibility.....	12
C.	Where to Turn for Help	12
V.	The Value of Testing.....	13
A.	The Need for Standardization in Testing	14
B.	Developer Benefits.....	14
C.	Operator Benefits.....	15
D.	Handset Manufacturer Benefits.....	15
E.	Consumer Benefits.....	16
F.	From Market-specific to Global Benefits	16
VI.	Conclusion.....	17
A.	Testing Programs	17
B.	Developer Forums	17

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.

To begin, the challenges:

For **developers** of mobile applications, it's all about time to market, time to market, and time to market. Fragmented testing methodologies can force developers to submit an application for multiple rounds of testing. A cost-effective and multi-tiered testing and QA process is a major key to accelerating time to market. Further, the short-lived shelf life of mobile applications demand a constant need to stay abreast of the latest platform, handset and operator requirements.

For **operators**, challenges include the need to minimize costs while maximizing customer satisfaction. When there is a lack of application accountability, the operators not only lose revenue, they also run the risk of losing consumer confidence.

For **consumers**, challenges include a lack of understanding of the mobile language and not necessarily knowing where to turn when they need help with their mobile application. Further complicating the matter are operators and handset manufacturers who try to trademark and brand general industry acronyms, heightening the mobile language barrier for consumers.

Now, the benefits of standardized testing:

With less fragmentation across operator and within platform testing requirements and standards, **developers** will not have to navigate through a complicated and lengthy path to market. Industry wide operator agreement in testing standards, will result in quality applications that are simultaneously available across all major operator networks. The goal of the emerging movement toward one-submission testing within platform testing and QA requirements is that applications see a more timely release to market across a greater number of handset models and devices. An application that is certified and secure is an application with greater financial value to all participants in the mobile revenue stream.

Handset manufacturers/vendors benefit from standardized test platforms that are built to match standardized development and execution platforms. The goal of these test platforms is to create increased device interoperability and more efficient and cost-effective testing requirements.

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.

The market for developing applications for mobile devices is not an old market, but its growth and the introduction of new technologies is moving at a rapid pace. Mobile devices and networks have progressed quickly in the recent past (though not quickly enough for many users). Today's devices offer far more capabilities than those of just two years ago, adding screen resolution, color depth, cpu speed, camera, and other capabilities at an extreme pace. Plus today's networks can now handle the many features that are being built into the devices and applications. Handsets that enable text messaging and the downloading of ring tones may have been the next big thing two years ago, whereas today, these devices are quickly being upgraded to handsets that have richer graphic capabilities through more complex color screens. The latest handsets take pictures and record video; they are capable of representing both 2D and 3D animations, email and video mail and more. For developers to succeed in a market where technology continues to advance, they must familiarize themselves across all of these platform and handset capabilities.

Following are four crucial and common challenges developers encounter when navigating the maze to market:

A. Time to Market

In the mobile world, an application that doesn't make it to market on time is an application that will miss large revenue opportunities. Developers that consistently miss these market opportunities probably won't be considered by the major operators or for the choice content licenses. Developing application software for the PC typically involves large teams of developers, working together for nearly a year or longer to create and take the final product to market, a process that can cost hundreds of thousands of dollars (or more). Within the mobile application industry, development teams are much smaller (often the entire development is completed by one individual), development budgets are closer to tens of thousands of dollars and development cycles typically occur over a few months.

Handsets, often considered to be trendy fashion statements, are the final destination for mobile applications and have a significantly shorter shelf live than PCs and game consoles. It benefits the developer to have an application created and released for the widest assortment of handsets. A refreshing market of opportunity, operators release handsets at various times of the year, and each handset launch creates an opportunity for developers to include their application during the launch of the handset (when sales are at the highest level). This also creates opportunity to bring existing titles to market on a new handset. When an application being designed for these new handset models does not coincide with the operator cycles,

developers risk missing the opportunity to take their application to market during the best and most profitable conditions. Couple this with the practice of cross-marketing intellectual property for coincidental release, and it is easy to see that for mobile application developers, timing is everything.

Timely market launch is a challenge that requires developers to overcome not one, but many hurdles. Timely submission of an application for operator testing and QA requirements is one challenge. However, the larger hurdle is that operators do not share a common set of standards for quality and assurance testing. For an application to be approved across multiple operator networks, it must be developed in time for multiple submissions across each operator's different testing requirements.

B. Different Operators, Different Requirements

Successful launch of an application with one operator does not equate to successful market launch across all operators. Currently, there exist few application interoperability requirements between operators (although several industry initiatives have already been spearheaded to address this issue). For developers, the wide variety of operator requirements has added a level of complexity previously not found. This poses several challenges; instead of only having to submit a mobile application for one set of functionality, security and quality assurance testing, each application must pass many sets of requirements. Functionality, security and quality assurance tests often differ with each operator.

If an application is not submitted with every potential operator, the consumer audience is smaller than it could be and profitability may be less than anticipated. Applications may need to be written specifically to meet an operator's network requirements adding to the development effort. Many larger publishers releasing an application on a global basis report needing hundreds of SKUs. And when operator requirements differ, applications need to be re-written so they are compatible with each operator's requirements. This creates longer development cycles and less profitability for developers. For developers who understand the importance of testing as asset and not a liability, the reward is the timely launch of an application across multiple operator networks.

C. The Role of Testing

The role of testing in maintaining a profitable and growing mobile marketplace cannot be stressed enough. For developers, testing provides assurances that the application is ready for successful market deployment. With testing processes that consider the complete path to market, many of the challenges to market are eased.

Like any application, a mobile application must be tested for standard functionality. But in the mobile universe, applications must also be submitted for device-specific testing as well as language and security requirements. While the mobile testing process is extensive, the end result is a positive one for developers as a signed and certified application is a trusted application. In the mobile industry, the term certification typically refers to a process by which applications are reviewed against a set of criteria and they either pass or fail based on that check. Platform programs serve as a “seal of approval” for developers. Among others, platform programs include Java Verified™, Symbian Signed, and Palm Powered Mobile World. When an application is tested within a platform, handset manufacturers, operators and consumers know it is an application that is trustworthy.

For developers, the need for testing is becoming more prevalent than ever. Operators are currently considering blocking downloads from third-party domains that do not require any testing process; however none of them currently block downloading. In another approach, operators are examining methods that warn the consumer that the application could be problematic, but ultimately leave the final decision to download and use the application to the consumer. It is also likely that applications that pass operator and handset manufacturer testing will receive better placement on storefronts, and better opportunities for marketing partnerships. Another challenge with handsets is the pace of growth in device capabilities.

D. Handset Inventory and Distribution

Developers are currently required to constantly refresh their handset learning to keep up with the ever-evolving handset inventory across the world. It's not an easy task. Newer, more capable and more functional handsets and mobile devices are vying for consumer attention every year. Each of the many handsets on the market implements a development platform. Major platforms include Microsoft Mobile, Java™, Symbian™, and Blackberry® and each of these platforms require application submission and testing unique to that platform. An application may even require separate code for different handset models within one manufacturer's inventory, on the same platform. The difference in standards across handsets even within a given development platform makes it necessary for developers to maintain an understanding of new handset developments. Testing programs that are industry wide provide the opportunity for developers to submit their application across an ever-evolving handset inventory, and even across application environments without having to navigate each individual program.

Operators also face significant challenges when bringing mobile products to market. In the next segment, we will focus on the challenges they face.

III. Challenges for Operators

The good news is; developers aren't alone in this mobile industry challenge. Operators must also overcome several critical challenges. When consumers have a question about why an application isn't working on their phone, they often turn to their operator for support. As consumer comfort with and demand for the latest mobile handset technology increases, so do the costs of customer service and support. Further, applications that are susceptible to viruses or those that otherwise interrupt a device's functionality can lead consumers to lose trust in their operator. These challenges and others have prompted operator demand for more rigid application testing and quality control, and more industry wide standards to help ensure that applications are trustworthy before allowing them to reach the marketplace.

Three of the most common challenges that a lack in testing standardization poses for operators, who are proactively driving the movement towards standardization, are presented below. For operators, it's all about profitability, quality, and accountability:

A. Escalating Support Costs

Due, in part, to security threats that have internationally penetrated PC networks, mobile operators have made it a priority to counter security threats before they occur. This aggressive approach has helped operators to minimize security disruption. While the benefits are clear, this approach also comes at a significant cost to operators. But security is only one challenge. With no industry wide testing and QA requirements, operators are more accountable for applications that don't function properly, resulting in higher customer service costs. Or, when an application disables a phone and renders it useless, operators lose all-important usage revenue. These are just a few reasons why operators are moving toward more industry standards; where accountability can be better shared between operators, device manufacturers and application developers. A second, less direct threat to operator revenue is a loss of customer trust. To lower the costs of customer support, operators want to raise quality control requirements.

B. Application Quality Control

Operators need to know that an application is bug-free before they allow it to be downloaded within their network. Simply requiring that an application be signed is only the first step for operators. When considering quality control from an operator perspective, application testing means that there is an assurance that the application is a trustworthy one. While layers of testing are multi-tiered and testing requirements are different between operators and handset manufacturers, there are a few types of testing that are of particular importance to operators: One is functional testing. This assures that an application performs as expected and thus the customer is provided with a positive experience.

The second is platform testing. This provides assurances for the operators, handset manufacturers and consumers, that the application will not disrupt day-to-day operation of their device (such as phone calls, SMS, and handset stability).

A third set of tests of particular importance to operators is security testing. This satisfies that their network is protected against the threat of mobile viruses, and malware / spyware.

The current reality of testing fragmentation makes it more of a challenge for operators to depend upon function, platform and security testing. A final operator concern is one that can have a direct effect on their bottom line.

C. Testing Across Distinct Networks

Operators share many of the same time-to-market and profitability concerns with the developer community. Fragmented testing methodologies and requirements can significantly hinder an operator's handset release schedule and add to the difficulty of releasing a quality product on the market. Fragmented testing methodology reduces the effectiveness of testing and also reduces consistency across the market.

As the success of the worldwide mobile marketplace is ultimately in the hands of the consumer, the last and final set of challenges will focus on them.

IV. Challenges for Consumers

For most consumers, the mobile marketplace is still a relatively new one. Excluding the early adopters of mobile technology, most consumers are not yet comfortable with mobile technology and vocabulary or with mobile limitations and possibilities. From not understanding the difference between an SMS and a GPRS-enabled email, most consumers are still searching for their mobile comfort zone. Consumers are not only being asked to embrace new and rapidly changing capabilities, but also a new language filled with acronyms like GSM and CDMA. Further, consumers don't know whether they should call their operator or their handset manufacturer, if for example, the new game they just downloaded seems to have disappeared from their phone. Lastly, PC viruses have burned consumers in recent years and so Operators and handset vendors are understandably concerned when it comes to mobile security. Following are three of the biggest roadblocks for the mobile industry's end users:

A. Understanding the Mobile Language

With consumers, a lack of understanding equates to a lack of trust. Just when most people got comfortable with the acronym PDA, along come terms like WAP, GSM, GPS, GPRS, and CDMA, to name a few. Outside of technology-savvy youths and early adopters, not many consumers can explain what a WAP server is and what it means to them. Beyond learning the capabilities of these new devices, they are being asked to learn an entirely different language.

Market fragmentation and different standards and technologies slow consumer adoption of these new application and handset capabilities. Worst of all, in an attempt to battle the alphabet soup, operators and handset manufacturers have both attempted to brand some of the acronym standards by calling them by trademarked names. So even a savvy consumer has to double check the names of the features on their phone and see if that feature is something truly new, or just a standard feature by another name. As demonstrated in the sections exploring developer and operator challenges, the rollout of new handsets forces to the consumers to not only learn the mobile language, but also the new mobile hardware along with hardware capabilities.

B. Handset Compatibility

Consumers don't understand why the application they downloaded for their old handset won't play on their new handset. With the rapid pace of change throughout the mobile industry, consumers are being asked to adopt new devices, new applications and new capabilities without enough explanation of why they should do so. Most consumers have only moderate to low interest in paying to download applications onto their phone and for those who do pay, they may have a difficult time understanding why the application doesn't work on devices from the same operator and even the same handset manufacturer. When this occurs, consumers need to know who to hold accountable and how these issues will be remedied.

C. Where to Turn for Help

According to Mobinet, a project from A.T. Kearney and Cambridge University Business School, the Judge School of Management, 75% of current mobile users use some form of mobile data services. But where to turn for usability and customer support help with these services is a question that needs to be better enunciated for consumers. They need to know

the right source to turn to when they encounter problems with charges on their mobile plan versus where to turn when there exists damage to their device or phone, or for a loss of personal data, or problems with mobile functionality. Fragmented delegation of roles and responsibilities within the mobile industry translates to consumer frustration when they need help.

How these challenges are turned into benefits for the entire mobile community is a problem many feel can be addressed through the establishment of standardization in testing.

V. The Value of Testing

It is important that developers understand and utilize the need for multi-tiered testing methodologies. It's the only way to ensure the quality of their product and their ability to release their application. In many cases the environments mobile applications have to run in are quite complex, and often have an impact on customer experience with the application. To help defray the costs of testing and decrease time-to-market, developers can turn to organizations that offer one-stop testing for their applications – for different platforms, different networks, different handsets and across multiple languages. By covering all phases of QA at the same time, developers ensure fast and cost-effective deployment of their application. Mobile application development teams are typically smaller than software development teams, so partnering with an organization that handles all phases of testing and QA on their behalf allows these teams to focus their efforts on the development process alone.

Why do some developers and publishers consistently put out a high quality product while others have difficulty with the quality equation? Some of this can be attributed to the structure and maturity of the development effort – good programming practices, and good testing. Other developers have more haphazard and less structured efforts. An important element is having an understanding of the effort needed – requirements for devices, operators, languages, and platforms.

A. The Need for Standardization in Testing

Testing results in quality. Testing results in assurance. As operators begin to require testing standards, the cost of testing is reduced while the eventual market for an application is increased for the testing effort. The wireless industry has begun to adopt comprehensive testing and quality assurance standards in an effort to address the challenges of application development and deployment. In establishing more rigorous testing methodologies, and leveraging the best efforts of platform vendors and providers, the mobile industry is able to maintain quality and sustain growth by creating a system of checks and balances within the rapidly growing mobile application environment. By doing so, a degree of quality assurance is being added to each and every step of the application development and sales process. From the challenges raised by a lack of testing, to defining the role of testing, the next steps are to explore the benefits of standardization in testing.

Quite simply, standardization in testing requirements will benefit the entire mobile marketplace, from consumers to developers and operators and every stage in between. Testing and standardized programs provide a comfort zone of quality and security for all applications and the more the mobile industry streamlines the definitions of quality and security, the better the process and the end result for all constituents. The benefits are clear for each segment of the mobile marketplace:

B. Developer Benefits

Mobile applications are not only tested for standard functionality, they are also tested for device, operator, language, and security specific compatibility. With less fragmentation across application requirements and standards, developers are not required to learn so many different rules and standards. Common testing requirements allow developers to submit their application once – for a comprehensive battery of tests – as opposed to multiple submissions for varying programs and requirements.

There are more than one hundred mobile handsets and devices on the international market that are equipped with application functionality. Further, there are several different platforms that enable these applications to run on one device versus another. Organizations dedicated to testing can manage the entire testing process through one submission and allow developers to deploy games across multiple handsets and networks at a fraction of the time

and cost. When a developer's application is viewed to be bug-free and secure, the result is an increased value of the application.

C. Operator Benefits

For operators, they gain assurances that the quality of applications being accepted adheres to a higher standard of quality and security, thereby minimizing their customer support demands. Application interoperability between operator networks equates to an industry-wide reduction in QA costs. Better quality and accountability means more profitability through less customer service demand. Customer activation and retention is an undeniable operator concern, as with credit cards, consumer loyalties can quickly shift when they believe something better has come along. With better application quality comes higher customer satisfaction and increased customer loyalty.

The benefits to both operators and developers from the standardization of the testing and QA processes across multiple platforms share many similarities. The current state of fragmentation existing in the testing methodologies in the mobile industry reduces the effectiveness of testing, by increasing both costs and time to market, as well as reducing consistency across the market in general. By adopting standard testing processes, operators and developers are cooperating to increase the benefits and enhance, yet simplify the testing process.

D. Handset Manufacturer Benefits

The standardization of platform requirements is perhaps the most obvious need and the process that will meet the most resistance. By standardizing platform requirements, applications can be deployed more swiftly across different models. Conversely, it should be noted that there is a fear of losing competitive advantage by making it more difficult for manufacturers to claim device superiority. It is unlikely that this will ever be perfected, as a balance needs to be maintained between the compatibility across handsets and the innovation that follows from new handset introductions.

In the past, it was up to each handset manufacturer to determine what testing and QA was necessary for platform functionality. As handset platforms became hosts to standardized environments such as Java and Symbian, the handset manufacturer testing procedures began to have significant overlap, creating duplication and inefficiency in the testing process.

This has led to the creation of standardized test specifications matching the standardized development and execution platforms.

E. Consumer Benefits

Consumers are unaware of testing. They become aware of the lack of testing when their application provides them with a poor user experience or when an application has a negative impact on their mobile device. One of the hurdles to changing the consumer mindset about purchasing applications for their mobile devices can be overcome by providing them with high quality applications.

F. From Market-specific to Global Benefits

The mobile application industry is one of global reach, truly making it an industry without boundaries. However, disparate markets equal disparate requirements. Because of this, there are challenges to testing consolidation. To help address these challenges, there is an industry-wide movement to create more uniform standards to help ensure quality control. In doing so, the entire industry is able to benefit from best practices all over the world. Operators such as Orange are taking a leadership role to bring this vision to pass.

Industry-wide standardization in testing frees developers to focus on fewer testing methodologies. This results in a more thorough understanding of the core methodologies and enables them to determine whether their application meets all of the necessary requirements to go global. Also, developers can always be assured that they meet or exceed the industry requirements for application quality and security.

VI. Conclusion

Today, operators such as Orange have begun to fully support industry-wide testing and QA standardization. They understand how and why doing so will lessen the barrier to marketplace entry for quality mobile products and applications. This is a significant step in the consolidation of testing efforts. Within the current structure of the industry, it is the operator who has the most direct relationship with and influence over the mobile customer. Because of this direct relationship, the operator must be very concerned with the quality of the product that their customers are purchasing. The operator also has many of the same revenue and time-to-market concerns that the developer faces. Both of those concerns are helped by the industry's adoption of standardized processes.

With more clearly defined rules and standards for testing and QA, the mobile industry can truly become one globally networked industry where operators, manufacturers, developers and end users all benefit from overlapping touch points of control. Developers benefit through feedback on how their application is performing in the market, they gain marketing support from operators, and lastly from simplified testing requirements. Manufacturers gain streamlined device and testing requirements and better channels of distribution. Consumers gain a global outlet to voice their opinions and better service and support. And operators gain the ability to stay in the center of all activity.

For more information on standardized programs, developers are encouraged to access the following links:

A. Testing Programs

- [Symbian Signed](#)
- [JavaVerified](#)
- [QUALCOMM TRUE BREW](#)
- [Microsoft Mobile2Market](#)
- [Palm Powered Mobile World](#)

B. Developer Forums

- [Orange Partner](#)
- [Forum Nokia](#)
- [Siemens Mobile Developer Program](#)
- [Sony Ericsson Developer World](#)
- [Motorola iDEN Developer Community](#)
- [Blackberry Developer Zone](#)

BUG TERMINOLOGY

Software bug

A problem that causes a program to produce invalid output or to crash (lock up). The problem is either insufficient logic or erroneous logic. For example, a program can crash if there are not enough validity checks performed on the input or on the calculations themselves, and the computer attempts to divide by zero. Bad instruction logic misdirects the computer to a place in the program where an instruction does not exist, and it crashes.

A program with bad logic may produce bad output without crashing, which is the reason extensive testing is required. For example, if the program is supposed to add an amount, but subtracts it instead, bad output results, although the computer keeps running.

ABEND

(AB)normal END) Also called a "crash" or "lock-up," an abend occurs when the computer is presented with instructions or data it cannot recognize or the program is reaching beyond its protective boundary. It is generally the result of erroneous software logic in the application or operating system. However, a hardware failure can also cause the computer to stop. A short circuit on the motherboard can simply halt the operation, but a failing memory cell can cause an instruction to point to an erroneous location, making it look like a software failure.

It Depends on the OS

If the abend occurs due to a bug in the software and the operating system is not crashproof, the computer locks up and has to be rebooted. Operating systems with memory protection attempt to halt the offending program and allow the remaining programs to continue. For example, as Windows evolved through the years, it became more resilient to application bugs (GPFs). There is less rebooting of the computer after a crash with Windows NT/2000/XP than with Windows 95/98/ME or DOS/Windows 3.x.

Abending

Abending, or crashing, often occurs when the program points outside of its address space. This diagram depicts the anatomy of a program. "The Data" refers to constants used within the program and the input/output areas that hold the data while it is being processed. "The Processing" refers to the program's logic embodied in the flow chart and physically implemented as thousands of machine instructions (the columns).

Bug

A persistent error in software or hardware. If the bug is in software, it can be corrected by changing the program. If the bug is in hardware, new circuits have to be designed.

Broken

Not working properly. The term applies to software as well as hardware. If software is

"broken," it means there is a bug in it. It may mean that people are having difficulty using it, because of poor design or cryptic error messages, in which case some parts of all software are "broken."

Bug fix

A revised program file or patch that corrects a software bug

Hot fix

To make a repair during normal operation. It often refers to marking sectors in poor condition as bad and remapping the data to spare sectors. Some SCSI drives can automatically move the data in sectors that are becoming hard to read to spare sectors without the user, operating system or even the SCSI host adapter being aware of it.

Patch

A fix to a program. In the past, a patch used to mean changing actual executable, machine instructions, but today more often than not, it means replacing an executable module in its entirety such as an .EXE or .DLL file. A profusion of patches to an application implies that its logic was poorly designed in the first place. It also implies that the program's internal logic is becoming spaghetti code and more difficult to maintain.

Buggy

Refers to software that contains many flaws. Many in the software industry swear that bugs are inevitable, and perhaps they are right. As long as we work in the competitive, pressure-cooker environment of our high-tech world, products will more often than not be developed too hastily and released too early.

Bugrade

(**BUg upGRADE**) A software upgrade that fixes bugs more than it adds functionality. Although new features are touted, the upgrade is purchased to eliminate headaches in the prior version.

Glitch

A temporary or random hardware malfunction. It is possible that a bug in a program may cause the hardware to appear as if it had a glitch in it and vice versa. At times it can be extremely difficult to determine whether a problem lies within the hardware or the software.

Web bug

Also called a "Web beacon," "pixel tag," "clear GIF" and "invisible GIF," it is a method

for passing information from the user's computer to a third party Web site. Used in conjunction with cookies, Web bugs enable information to be gathered and tracked in the stateless environment of the Internet. The Web bug is typically a one-pixel, transparent GIF image, although it can be a visible image as well. As the HTML code for the Web bug points to a site to retrieve the image, it can pass along information at the same time.

Web bugs can be placed into an HTML page used for e-mail messages as most mail programs support the display of HTML pages.

The Bug Life Cycle

Table of Contents

What happens to a bug from start to finish. 3

What is a bug? 3

Who can report a bug? 3

When do you report a bug? 3

Bugs are tracked in a database 3

A good bug reports include the following items 4

Things to remember... 4

Rating Bugs 4

Severity 5

Likelihood 5

Severity * Likelihood = Rating 5

Other useful information on a bug report 5

An example of a bug report 6

Examples of poorly written bugs 6

Now we have a bug... 7

The Developer works on the bug... 7

Fixed 7

Duplicate 7

Resolved 7

Need More Information 8

Working as Designed 8

Enhancement 8

Defer 8

Not to be Fixed 9

Tested 9

Pending 9

Can't Duplicate 9

The product has shipped, what happens next? 9

Advanced defect database techniques 9

Reports 9

Examples of Reports 10

Customized views of the database 10

The Bug Life Cycle 11

What happens to a bug from start to finish.

While attending testing seminars, I noticed that there was a gap in what was being taught. There's a lot of theory presented, a lot of 'why test' classes and a lot of classes on specific techniques but nothing on a couple of practices that will go a long way towards improving the testing process in a company, specifically setting up a defect tracking system and enforcing policies and procedures to resolve those defects. Setting up these two things, more than anything else, will put a company on the road to organizing its testing and QA effort. To fill that gap, I've come up with the 'Bug Life Cycle' presentation. While I can't claim it as my own, it is what I've learned over the years as a tester; many of you will find it familiar.

What is a bug?

In computer technology, a bug is a coding error in a computer program. Myers defined it by saying that "A software error is present when the program does not do what its end user reasonably expects it to do." (Myers, 1976.). I tell my testers if you don't like it, it's a bug.

Over the years, my colleagues and I have decided that there are as many definitions for the term "bug" as there are testers. "There can never be an absolute definition for bugs, nor an absolute determination of their existence. The extent to which a program has bugs is measured by the extent to which it fails to be useful. This is a fundamentally human measure." (Beizer, 1984.). For a more definitive list of many types of bugs refer to **Software Testing** by Cem Kaner, et. al., pages 363-432.

Who can report a bug?

Anyone who can figure out that the software isn't working properly can report a bug. The more people who critique a product, the better it's going to be. However, here's a short list of people expected to report bugs:

Testers / QA personnel

Developers

Technical Support

Beta sites

End users

Sales and marketing staff
(especially when interacting
with customers).

When do you report a bug?

When you find it! When in doubt, write it up. Waiting means that you'll forget to write it altogether or important details about the bug will be forgotten. Writing it now also gives you a 'scratch pad' to make notes on as you do more investigation and work on the bug.

Also, writing the bug when you find it makes that information instantly available to everyone. You don't have to run around the building telling everyone about the bug; a simple phone call or email will alert everyone that the bug exists. Additionally, the information about the bug doesn't change or get forgotten with every telling of the story.

Bugs are tracked in a database

The easiest way to keep track of defect reports is in a database. Paper is an ok way to record defect reports on but pieces of paper can get lost or destroyed; a database is more reliable and can be backed up on a regular basis.

You can purchase many commercially available defect tracking databases or you can build your own. It's up to you. I've always built my own with something small like Microsoft Access or SQL Server. The decision then was that it was cheaper to build and maintain it on site than it was to purchase it. You'll have to run the numbers for your situation when you make that decision.

The rule of thumb is one and only one defect per report (or record) when writing a bug report. If more than one defect is put into a report, the human tendency is to deal with the first problem and forget

the rest of them. Also, defects are not always fixed at the same time. With one defect per report, as the defects get fixed, they will be tested individually instead of in a group where the chance that a defect is overlooked or forgotten is greater.

You may hear the term "bugfile" used by people when referring to a defect database. The name bugfile is a slang term from the old WordPerfect Corporation. "Bugs" were first logged into a flat file database called DataPerfect; a file of bugs, hence the word "bugfile".

A good bug reports include the following items:

Put the *Reporter's Name* on the bug. If there are questions we need to know who originated this report.

Specify the *Build or Version number* of the code being worked on. Is this the shipping version or a build done in-house for testing and development? Some bugs may only occur in the shipping version; if this is the case, the version number is a crucial piece of information.

Specify the *Feature or Specification or part of the code*. This facilitates assigning the bug to a developer assigned to that part of the product.

Include a *Brief Description* of what the problem is. For example, "Fatal error when printing landscape." is a good description; short and to the point.

List *Details* including how to duplicate the bug and any other relevant data or clues about the bug. Start with how the computer and software is setup. List each and every step (don't leave any out) to produce the bug. Sometimes a minor detail can make all the difference in duplicating or not duplicating a bug. For example, using the keyboard versus using the mouse may product very different results when duplicating a bug.

If the status isn't '*Submitted*' by default, change it to Submitted. This is a flag to the bug verifier that a new bug has been created and needs to be verified and assigned.

Things to remember...

Keep the text of the bug impersonal. Bug reports will be read by a variety of people including those outside the department and even the company. Please don't insult people's ancestors or the company they work for or the state they live in or make any other impulsive or insensitive comment. Be careful with humorous remarks; one person's humor is another person's insult. Keep the writing professional.

Be as specific as possible in describing the current state of the bug along with the steps to get into that state. Don't make assumptions that the reader of the bug will be in the same frame of mind as you are. Please don't make people guess where you are or how you got into that situation. Not everyone is thinking along the same lines as you are.

Rating Bugs

While it is important to know how many bugs are in a product, it is even more useful to know how many of those bugs are severe, ship stopping bugs compared to the number of inconvenient bugs. To aid in assessing the state of the product and to prioritize bug fixes, bugs are ranked. The easiest way to rank or rate bugs is to assign each bug a severity rating and a likelihood rating. This assignment is done by the bug reporter when the bug is created. The bug's rating is a combination of the severity and likelihood ratings.

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:

Examples of poorly written bugs:

Please keep in mind that I didn't make any of these up!

<input type="button" value="submit"/>	<input type="button" value="Jump To Bug"/>	<input type="button" value="Navigation"/>	BugNumber	Submitted	Updated	Reported By	Build	Rating	Shipping																				
			1205	<< < > >>	1205	3/2/99	3/3/99	David Butler	205	6	No																		
<table border="1"><tr><td>Status:</td><td>In Progress</td></tr><tr><td>Severity:</td><td>Loss without a workaround</td></tr><tr><td>Likelihood:</td><td>The user will sometimes see this bug</td></tr><tr><td>Assigned To:</td><td>Lisa Anderson</td></tr><tr><td>Feature:</td><td>Voice Messaging</td></tr><tr><td>Problem Description:</td><td>Submitting a bug with no Build Number causes dk.</td></tr><tr><td>Details</td><td>Notes</td></tr><tr><td colspan="2">When a new bug is submitted without a build number, a blank form is returned with an error at the top stating that build number is required, and suggesting that the back button be used. Do as directed. Note that when you go back and enter a build number, a second bug is</td></tr><tr><td colspan="2">LISAAN 3/3/99 changed status to In Progress. Until this is fixed, remember to enter a build number. Thanks.</td></tr></table>												Status:	In Progress	Severity:	Loss without a workaround	Likelihood:	The user will sometimes see this bug	Assigned To:	Lisa Anderson	Feature:	Voice Messaging	Problem Description:	Submitting a bug with no Build Number causes dk.	Details	Notes	When a new bug is submitted without a build number, a blank form is returned with an error at the top stating that build number is required, and suggesting that the back button be used. Do as directed. Note that when you go back and enter a build number, a second bug is		LISAAN 3/3/99 changed status to In Progress. Until this is fixed, remember to enter a build number. Thanks.	
Status:	In Progress																												
Severity:	Loss without a workaround																												
Likelihood:	The user will sometimes see this bug																												
Assigned To:	Lisa Anderson																												
Feature:	Voice Messaging																												
Problem Description:	Submitting a bug with no Build Number causes dk.																												
Details	Notes																												
When a new bug is submitted without a build number, a blank form is returned with an error at the top stating that build number is required, and suggesting that the back button be used. Do as directed. Note that when you go back and enter a build number, a second bug is																													
LISAAN 3/3/99 changed status to In Progress. Until this is fixed, remember to enter a build number. Thanks.																													

Figure 1 The status tells us the state of the bug. The severity tells us how bad the bug is. The likelihood tells us how often the average user will see the bug. The Assigned To field tells us who is responsible for resolving the bug. The Feature tells us what part of the product the bug is in. The Problem Description gives a brief (very brief) summation of the problem. This description is used when compiling reports or lists of bugs. The details tell us the current setup or situation, the steps to duplicate the problem and any other essential information that will allow someone else to duplicate the bug. Once the bug is submitted, this field cannot be changed. Any additional information will go in the notes field. The notes field contains any discussions about the bug. For example, when and why the status was changed and by whom; additional information about how to duplicate the bug that will aid in resolving the bug; and opinions about the bug. This is a "free-forum" area where people should feel free to express their opinions without censure or criticism. Once comments are placed on a bug, they cannot be changed or deleted by anyone else or even the author. The comments, like the details, stand. Anyone reading the bug after the fact should be able to understand not only what the bug was and how bad a bug it was but also how it was resolved and why it was resolved that way.

"My computer crashed." We are sorry for your loss, can you give us more information?

"It's kinda "sucky".' This one violates all sorts of rules. What's kind of "sucky". For that matter, define "sucky.". Better yet, don't use the work "sucky" it's not in the dictionary and most certainly not in good taste.

"It don't." This bug doesn't provide enough information. What don't? Don't what?

"Product needs a "speel" checker." It goes without saying that "spelling counts"!

Now we have a bug...

The first step is *Verification*. A bug verifier searches the database looking for all 'Submitted' bugs assigned to him. He then duplicates the bug by following the steps listed in the details section of the bug. If the bug is reproduced and has all the proper information, the assigned to field is changed to the appropriate person who will be fixing the bug. If the bug is not written clearly, is missing some steps or can't be reproduced, it will be sent back to the bug reporter for additional work.

The *Assigned To* field contains the name of the person responsible for that area of the product or code. It is important to note that from this point onward, the developer's name stays on the bug. Why? There are usually more developers than there are testers. Developers have a set of features to work on. Developers look at bugs from a stand point of "what is assigned to me?". Testers have multiple sets of features to test. Testers look at bugs from a stand point of "what needs to be tested?"; testers may also change what features they are assigned to test. Because of the different way testers and developers work, developers sort bugs by the Assigned To field and testers sort bugs by the Status field. Leaving the developer's name on the bug also makes it easier to send the bug back to the developer for more work. The tester simply changes the status field to Verified and it automatically goes back to the developer.

The Developer works on the bug...

The first thing the developer does is give the bug a '*In Progress*' status indicating that he has seen the bug and is aware that it his responsibility to resolve. The developer works on the bug and based on his conclusions assigns a status to the bug indicating what the next step should be.

Remember, the developer does NOT change the *Assigned To* field. His name stays on the bug

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 getting 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.

This bug can go several directions after being assigned this status. If the tester agrees with the status, then the status stands and the bug is finished. The bug may be sent to documentation for inclusion in help files and the manual. If the tester disagrees with the status then the bug can be appealed by putting a Submitted status on it to send the bug back through the process again. The tester should include in the notes a reason why, although it is Working as Designed, it should be changed now. The bug may also be sent back to the design committee so that the design can be improved.

This is a dangerous status. It's an easy way to sweep bugs under the carpet by giving them this status. It's up to the bug reporter to make sure the bug doesn't get forgotten in this manner. Product managers may also review lists of bugs recently assigned Working as Designed.

Enhancement

Enhancement means that while the suggested change is great idea because of technical reasons, time constraints or other factors, it won't be incorporated into the code until the next version of the product.

This status may also be appealed by changing the status to Submitted and adding a note specifying why it should be fixed now.

Defer

Defer is almost the same status as Enhancement. This status implies that the cost of fixing the bug is too great given the benefits that it would produce. If the fix is a one liner to one file that doesn't influence other files, it might be ok to fix the bug. On the other hand, if the fix will force the rebuild of many files which would force the re-testing of the product and there's no time left to test the fix before shipping the product, then the fix would be unacceptable and the bug would get a Defer status. To appeal the status, send it back through the process again by putting a Submitted status on it and add a note saying why it should be fixed now.

Not to be Fixed

You may see the *Not to be Fixed* status although I don't recommend making this status available for use. There may be extenuating circumstances where a bug will not be fixed because of technology, time constraints, a risk of destabilizing the code or other factors. A better status to use is Not to be Fixed. To appeal the status, send it back through the process again by putting a Submitted status on it and add a note saying why it should be fixed now.

This is similar to the Working as Designed status in that its use can be dangerous. Be on the watch for this one. Sometimes developers call this status "You can't make me".

Tested

The *Tested* status is used only by testers on Fixed, Resolved and Duplicate bugs. This status is a "end of the road" status indicating that the bug has reached the end of its life cycle.

Pending

The *Pending* status is used only by testers on Fixed bugs when a bug cannot be immediately tested. The tester may be waiting on hardware, device drivers, a build or additional information necessary to test the bug. When the necessary items have been obtained, the bug status is changed back to Fixed and then it is tested. Make sure the testing part isn't skipped.

Can't Duplicate

This status is used only by the bug reporter; developers or managers cannot use this status. If a bug isn't reproducible by the assigned developer or bug verifier the bug reporter needs a chance to clarify or add to the bug. There may be a hardware setup or situation or particular way of producing a bug that is peculiar to only this computer or bug reporter and he needs a chance to explain what the circumstances are. Limiting this status to bug reporters only prevents bugs from slipping between the cracks and not getting fixed.

The product has shipped, what happens next?

First of all, to ship a product the number of bugs rated 5 or less, bugs with a Fixed, Resolved, Pending or Need More Information status must be zero. We'll assume that the product has shipped and

all these bugs have been taken care of. This means the bugfile is full of bugs that have reached the end of their life cycle. Proper database maintenance takes place at this point. Archiving or hiding all these bugs will make the database easier to use and read.

All bugs with a Tested or Can't Duplicate bugs are archived. This means that the records are either removed and placed in an archive database or flagged to be hidden from the current view of the database. Never delete any bug records; it may be necessary to do some historical research in the bugfile ('What did we ship when?' or 'Why did we ship with this bug?').

Enhancement and Defer bugs are either moved to the new bugfile or retained in the current bugfile. The status of these bugs is then changed back to Verified.

Advanced defect database techniques

There are things you can do with your database to make it more than just a to-do list for developers and testers. The bugfile is basically raw data, sorting and filtering it makes the data information that is useful in the decision making process done by management. A couple of these things to do with the data is to create Reports and Customized views.

Reports

The data in the defect database is not very useful until it is sorted and presented in a organized fashion, then it becomes information. For example, sorting by developer, the information becomes a 'to-do' list sorted by rating. Sorting by status lets the reader know how many bugs are submitted or in progress; i.e. how many bugs are currently being worked on? By feature – how many open bugs are there for a particular feature? What feature needs more work and what feature is stable? Sorting by product is useful when more than one product is being worked on simultaneously.

Be aware that there are certain metrics or reports that should not be used. If you use these reports you will destroy the credibility of your bug file and it will be reduced to a "laundry-list" for developers. One of these reports is "how many bugs did a tester report" and the other is "how many bugs did a developer fix". Neither one of these has any useful purpose except to beat up people uselessly. See **Software Testing** by Cem Kaner, et. al.

Examples of Reports

The Product Manager wants to see a list of bugs to be fixed before shipping that is sorted by rating to determine how ready the product is for shipping. He needs to know what work is left to be done so that an appropriate schedule can be set. The Test Lead needs a list of bugs to be tested before shipping, sorted by tester to make sure all the bugs get tested in the allotted amount of time. The Development Lead has a list of bugs to be resolved before shipping, sorted by developer so that adjustments in work load can be made to ensure that everything is taken care of by the deadline. Technical Support likes a list of bugs that were fixed and not fixed in the product before shipping so they can adequately prepare to support the product. Technical support can resolve customer problems faster by knowing what has been fixed in a product release (the new release or upgrade will fix that problem) and what bugs are still in the product.

A defect database that has all these fields built into it is able to sort defect data and make it useful information.

Customized views of the database

If your defect tracking database supports it, you can limit the information seen based on who the person logging in is. For example, the database that I built with SQL Server 7 was web browser based. Each person was assigned a job category such as 'Tester', 'Developer' or 'Manager'. The web page that displayed was then the web page designed for that job category. The Developer would see Submitted and In Progress bugs assigned to him while Testers would see all Fixed, NMI, Pending, Resolved, and Duplicate bugs regardless of who is assigned to the bug. The Product Manager view sees all bugs assigned to him, all newly reported bugs, and newly updated bugs.

The reason for this customized view is that each job views the defect database in a different way based on their needs and the work that has to be accomplished. Testers don't care who the bug is assigned to, just what the status is. Testers want to see all the bugs, not just bugs written by them, with a 'Fixed' status so the bugs can be tested with the latest build. Developers only care about what current and active bugs are assigned to them; developers aren't concerned about any other person's bugs.

**BUG LIFE CYCLES – BUG STATES – CONTENTS OF A BUG
– REPORT / LOG A BUG – WHITE BOX TESTING STRATEGY
– BLACK BOX TESTING STRATEGY**

Error: programmatically mistake leads to error.

Bug: Deviation from the expected result.

Defect: Problem in algorithm leads to failure.

Failure: Result of any of the above.

Bug Life Cycles

The duration or time span between the first time bug is found ('New') and closed successfully (status: 'Closed'), rejected, postponed or deferred is called as 'Bug/Error Life Cycle'.

(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. For more information about various statuses used for a bug during a bug life cycle, you can refer to article 'Software Testing – Bug & Statuses Used During A Bug Life Cycle')

There are seven different life cycles that a bug can pass through:

< I > Cycle I:

- 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) Test lead finds that the bug is not valid and the bug is 'Rejected'.

< II > Cycle II:

- 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 development leader and team verify if it is a valid bug. The bug is invalid and is marked with a status of 'Pending Reject' before passing it back to the testing team.
- 5) After getting a satisfactory reply from the development side, the test leader marks the bug as 'Rejected'.

< III > Cycle III:

- 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 development leader and team verify if it is a valid bug. The bug is valid and the development leader assigns a developer to it marking the status as 'Assigned'.
- 5) The developer solves the problem and marks the bug as 'Fixed' and passes it back to the Development leader.
- 6) The development leader changes the status of the bug to 'Pending Retest' and passes on to the testing team for retest.
- 7) The test leader changes the status of the bug to 'Retest' and passes it to a tester for retest.
- 8) The tester retests the bug and it is working fine, so the tester closes the bug and marks it as 'Closed'.

< IV > Cycle IV:

- 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 development leader and team verify if it is a valid bug. The bug is valid and the development leader assigns a developer to it marking the status as 'Assigned'.
- 5) The developer solves the problem and marks the bug as 'Fixed' and passes it back to the Development leader.
- 6) The development leader changes the status of the bug to 'Pending Retest' and passes on to the testing team for retest.
- 7) The test leader changes the status of the bug to 'Retest' and passes it to a tester for retest.

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 make it secure 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/document 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.

(Please note that there are various ways to communicate the bug to the developer and track the bug status)

Statuses associated with a bug:

New:

When a bug is found/revealed for the first time, the software tester communicates it to his/her team leader (Test Leader) in order to confirm if that is a valid bug. After getting confirmation from the Test Lead, the software tester logs the bug and the status of 'New' is assigned to the bug.

Assigned:

After the bug is reported as 'New', it comes to the Development Team. The development team verifies if the bug is valid. If the bug is valid, development leader assigns it to a developer to fix it and a status of 'Assigned' is assigned to it.

Open:

Once the developer starts working on the bug, he/she changes the status of the bug to 'Open' to indicate that he/she is working on it to find a solution.

Fixed:

Once the developer makes necessary changes in the code and verifies the code, he/she marks the bug as 'Fixed' and passes it over to the Development Lead in order to pass it to the Testing team.

Pending Retest:

After the bug is fixed, it is passed back to the testing team to get retested and the status of 'Pending Retest' is assigned to it.

Retest:

The testing team leader changes the status of the bug, which is previously marked with 'Pending Retest' to 'Retest' and assigns it to a tester for retesting.

Closed:

After the bug is assigned a status of 'Retest', it is again tested. If the problem is solved, the tester closes it and marks it with 'Closed' status.

Reopen:

If after retesting the software for the bug opened, if the system behaves in the same way or same bug arises once again, then the tester reopens the bug and again sends it back to the developer marking its status as 'Reopen'.

Pending Rejected:

If the developers think that a particular behavior of the system, which the tester reports as a bug has to be same and the bug is invalid, in that case, the bug is rejected and marked as 'Pending Reject'.

Rejected:

If the Testing Leader finds that the system is working according to the specifications or the bug is invalid as per the explanation from the development, he/she rejects the bug and marks its status as 'Rejected'.

Postponed:

Sometimes, testing of a particular bug has to be postponed for an indefinite period. This situation may occur because of many reasons, such as unavailability of Test data, unavailability of particular functionality etc. That time, the bug is marked with 'Postponed' status.

Deferred:

In some cases a particular bug stands no importance and is needed to be/can be avoided, that time it is marked with 'Deferred' status.

Contents of a Bug

When a tester finds a defect, he/she needs to report a bug and enter certain fields, which helps in uniquely identifying the bug reported by the tester. The contents of a bug are as given below:

Project: Name of the project under which the testing is being carried out.

Subject: Description of the bug in short which will help in identifying the bug. This generally starts with the project identifier number/string. This string should be clear enough to help the reader in anticipate the problem/defect for which the bug has been reported.

Description: Detailed description of the bug. This generally includes the steps that are involved in the test case and the actual results. At the end of the summary, the step at which the test case fails is described along with the actual result obtained and expected result.

Summary: This field contains some keyword information about the bug, which can help in minimizing the number of records to be searched.

Detected By: Name of the tester who detected/reported the bug.

Assigned To: Name of the developer who is supposed to fix the bug. Generally this field contains the name of developer group leader, who then delegates the task to member of his team, and changes the name accordingly.

Test Lead: Name of leader of testing team, under whom the tester reports the bug.

Detected in Version: This field contains the version information of the software application in which the bug was detected.

Closed in Version: This field contains the version information of the software application in which the bug was fixed.

Date Detected: Date at which the bug was detected and reported.

Expected Date of Closure: Date at which the bug is expected to be closed. This depends on the severity of the bug.

Actual Date of Closure: As the name suggests, actual date of closure of the bug i.e. date at which the bug was fixed and retested successfully.

Priority: Priority of the bug fixing. This specifically depends upon the functionality that it is hindering. Generally Medium, Low, High, Urgent are the type of severity that are used.

Severity: This is typically a numerical field, which displays the severity of the bug. It can range from 1 to 5, where 1 is high severity and 5 is the lowest.

Status: This field displays current status of the bug. A status of 'New' is automatically assigned to a bug when it is first time reported by the tester, further the status is changed to Assigned, Open, Retest, Pending Retest, Pending Reject, Rejected, Closed, Postponed, Deferred etc. as per the progress of bug fixing process.

Bug ID: This is a unique ID i.e. number created for the bug at the time of reporting, which identifies the bug uniquely.

Attachment: Sometimes it is necessary to attach screen-shots for the tested functionality that can help tester in explaining the testing he had done and it also helps developers in re-creating the similar testing condition.

Test Case Failed: This field contains the test case that is failed for the bug.

Any of above given fields can be made mandatory, in which the tester has to enter a valid data at the time of reporting a bug. Making a field mandatory or optional depends on the company requirements and can take place at any point of time in a Software Testing project.

(Please Note: All the contents enlisted above are generally present for any bug reported in a bug-reporting tool. In some cases (for the customized bug-reporting tools) the number of fields and their meaning may change as per the company requirements.)

How to Report/Log a bug?

It's a good practice to take screen shots of execution of every step during software testing. If any test case fails during execution, it needs to be failed in the bug-reporting tool and a bug has to be reported/logged for the same. The tester can choose to first report a bug and then fail the test case in the bug-reporting tool or fail a test case and report a bug. In any case, the Bug ID that is generated for the reported bug should be attached to the test case that is failed.

At the time of reporting a bug, all the mandatory fields from the contents of bug (such as Project, Summary, Description, Status, Detected By, Assigned To, Date Detected, Test Lead, Detected in Version, Closed in Version, Expected Date of Closure, Actual Date of Closure, Severity, Priority and Bug ID etc.) are filled and detailed description of the bug is given along with the expected and actual results. The screen-shots taken at the time of execution of test case are attached to the bug for reference by the developer.

After reporting a bug, a unique Bug ID is generated by the bug-reporting tool, which is then associated with the failed test case. This Bug ID helps in associating the bug with the failed test case.

After the bug is reported, it is assigned a status of 'New', which goes on changing as the bug fixing process progresses.

If more than one tester are testing the software application, it becomes a possibility that some other tester might already have reported a bug for the same defect found in the application. In such situation, it becomes very important for the tester to find out if any bug has been reported for similar type of defect. If yes, then the test case has to be blocked with the previously raised bug (in this case, the test case has to be executed once the bug is fixed). And if there is no such bug reported previously, the tester can report a new bug and fail the test case for the newly raised bug.

If no bug-reporting tool is used, then in that case, the test case is written in a tabular manner in a file with four columns containing Test Step No, Test Step Description, Expected Result and Actual Result. The expected and actual results are written for each step and the test case is failed for the step at which the test case fails.

This file containing test case and the screen shots taken are sent to the developers for reference. As the tracking process is not automated, it becomes important keep updated information of the bug that was raised till the time it is closed.

(Please Note: The above given procedure of reporting a bug is general and not based on any particular project. Most of the times, the bug reporting procedures, values used for the various fields used at the time

of reporting a bug and bug tracking system etc. may change as per the software testing project and company requirements.)

White Box Testing Strategy

White box testing strategy deals with the internal logic and structure of the code. White box testing is also called as glass, structural, open box or clear box testing. The tests written based on the white box testing strategy incorporate coverage of the code written, branches, paths, statements and internal logic of the code etc.

In order to implement white box testing, the tester has to deal with the code and hence is needed to possess knowledge of coding and logic i.e. internal working of the code. White box test also needs the tester to look into the code and find out which unit/statement/chunk of the code is malfunctioning.

Advantages of White box testing are:

- i) As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.
- ii) The other advantage of white box testing is that it helps in optimizing the code
- iii) It helps in removing the extra lines of code, which can bring in hidden defects.

Disadvantages of white box testing are:

- i) As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost.
- ii) And it is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.

Types of testing under White/Glass Box Testing Strategy:

Unit Testing:

The developer carries out unit testing in order to check if the particular module or unit of code is working fine. The Unit Testing comes at the very basic level as it is carried out as and when the unit of the code is developed or a particular functionality is built.

Static and dynamic Analysis:

Static analysis involves going through the code in order to find out any possible defect in the code. Dynamic analysis involves executing the code and analyzing the output.

Statement Coverage:

In this type of testing the code is executed in such a manner that every statement of the application is executed at least once. It helps in assuring that all the statements execute without any side effect.

Branch Coverage:

No software application can be written in a continuous mode of coding, at some point we need to branch out the code in order to perform a particular functionality. Branch coverage testing helps in validating of all the branches in the code and making sure that no branching leads to abnormal behavior of the application.

Security Testing:

Security Testing is carried out in order to find out how well the system can protect itself from unauthorized access, hacking – cracking, any code damage etc. which deals with the code of application. This type of testing needs sophisticated testing techniques.

Mutation Testing:

A kind of testing in which, the application is tested for the code that was modified after fixing a particular bug/defect. It also helps in finding out which code and which strategy of coding can help in developing the functionality effectively.

Black Box Testing Strategy

Black Box Testing is not a type of testing; it instead is a testing strategy, which does not need any knowledge of internal design or code etc. As the name "black box" suggests, no knowledge of internal logic or code structure is required. The types of testing under this strategy are totally based/focused on the testing for requirements and functionality of the work product/software application. Black box testing is sometimes also called as "Opaque Testing", "Functional/Behavioral Testing" and "Closed Box Testing".

The base of the Black box testing strategy lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system. Now a days, it is becoming common to route the Testing work to a third party as the developer of the system knows too much of the internal logic and coding of the system, which makes it unfit to test the application by the developer.

In order to implement Black Box Testing Strategy, the tester is needed to be thorough with the requirement specifications of the system and as a user, should know, how the system should behave in response to the particular action.

Various testing types that fall under the Black Box Testing strategy are: functional testing, stress testing, recovery testing, volume testing, User Acceptance Testing (also known as UAT), system testing, Sanity or Smoke testing, load testing, Usability testing, Exploratory testing, ad-hoc testing, alpha testing, beta testing etc.

These testing types are again divided in two groups: a) Testing in which user plays a role of tester and b) User is not required.

Testing method where user is not required:**Functional Testing:**

In this type of testing, the software is tested for the functional requirements. The tests are written in order to check if the application behaves as expected.

Stress Testing:

The application is tested against heavy load such as complex numerical values, large number of inputs, large number of queries etc. which checks for the stress/load the applications can withstand.

Load Testing:

The application is tested against heavy loads or inputs such as testing of web sites in order to find out at what point the web-site/application fails or at what point its performance degrades.

Ad-hoc Testing:

This type of testing is done without any formal Test Plan or Test Case creation. Ad-hoc testing helps in deciding the scope and duration of the various other testing and it also helps testers in learning the application prior starting with any other testing.

Exploratory Testing:

This testing is similar to the ad-hoc testing and is done in order to learn/explore the application.

Usability Testing:

This testing is also called as ‘Testing for User-Friendliness’. This testing is done if User Interface of the application stands an important consideration and needs to be specific for the specific type of user.

Smoke Testing:

This type of testing is also called sanity testing and is done in order to check if the application is ready for further major testing and is working properly without failing up to least expected level.

Recovery Testing:

Recovery testing is basically done in order to check how fast and better the application can recover against any type of crash or hardware failure etc. Type or extent of recovery is specified in the requirement specifications.

Volume Testing:

Volume testing is done against the efficiency of the application. Huge amount of data is processed through the application (which is being tested) in order to check the extreme limitations of the system.

Testing where user plays a role/user is required:

User Acceptance Testing:

In this type of testing, the software is handed over to the user in order to find out if the software meets the user expectations and works as it is expected to.

Alpha Testing:

In this type of testing, the users are invited at the development center where they use the application and the developers note every particular input or action carried out by the user. Any type of abnormal behavior of the system is noted and rectified by the developers.

Beta Testing:

In this type of testing, the software is distributed as a beta version to the users and users test the application at their sites. As the users explore the software, in case if any exception/defect occurs that is reported to the developers.

What IS a Good Test Case?

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology
Department of Computer Sciences
kaner@kaner.com

STAR East, May 2003

Abstract

Designing good test cases is a complex art. The complexity comes from three sources:

- Test cases help us discover information. Different types of tests are more effective for different classes of information.
- Test cases can be “good” in a variety of ways. No test case will be good in all of them.
- People tend to create test cases according to certain testing styles, such as domain testing or risk-based testing. Good domain tests are different from good risk-based tests.

What's a Test Case?

Let's start with the basics. What's a test case?

IEEE Standard 610 (1990) defines *test case* as follows:

- “(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.”
- “(2) (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.”

According to Ron Patton (2001, p. 65),

“Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software.”

Boris Beizer (1995, p. 3) defines a test as

“A sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. The word ‘test’ is used to include subtests, tests proper, and test suites.

Bob Binder (1999, p. 47) defines test case:

“A test case specifies the pretest state of the IUT and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases

may also specify initial and resulting conditions for other objects that constitute the IUT and its environment.”

In practice, many things are referred to as test cases even though they are far from being fully documented.

Brian Marick uses a related term to describe the lightly documented test case, the *test idea*:

“A test idea is a brief statement of something that should be tested. For example, if you’re testing a square root function, one idea for a test would be ‘test a number less than zero’. The idea is to check if the code handles an error case.”

In my view, a test case is a question that you ask of the program. The point of running the test is to gain information, for example whether the program will pass or fail the test.

It may or may not be specified in great procedural detail, as long as it is clear what is the idea of the test and how to apply that idea to some specific aspect (feature, for example) of the product. If the documentation is an essential aspect of a test case, in your vocabulary, please substitute the term “test idea” for “test case” in everything that follows.

An important implication of defining a test case as a question is that *a test case must be reasonably capable of revealing information*.

- Under this definition, the *scope* of test cases changes as the program gets more stable. Early in testing, when anything in the program can be broken, trying the largest “legal” value in a numeric input field is a sensible test. But weeks later, after the program has passed this test several times over several builds, *a standalone test of this one field is no longer a test case because there is only a minuscule probability of failure*. A more appropriate test case at this point might combine boundaries of ten different variables at the same time or place the boundary in the context of a long-sequence test or a scenario.
- Also, under this definition, the metrics that report the number of test cases are meaningless. What do you do with a set of 20 single-variable tests that were interesting a few weeks ago but now should be retired or merged into a combination? Suppose you create a combination test that includes the 20 tests. Should the metric report this one test, 20 tests, or 21? What about the tests that you run only once? What about the tests that you design and implement but never run because the program design changes in ways that make these tests uninteresting?

Another implication of the definition is that a test is not necessarily designed to expose a defect. The goal is information. Very often, the information sought involves defects, but not always. (I owe this insight to Marick, 1997.) To assess the value of a test, we should ask how well it provides the information we’re looking for.

Information Objectives

So what are we trying to learn or achieve when we run tests? Here are some examples:

- **Find defects.** This is the classic objective of testing. A test is run in order to trigger failures that expose defects. Generally, we look for defects in all interesting parts of the product.

- **Maximize bug count.** The distinction between this and “find defects” is that total number of bugs is more important than coverage. We might focus narrowly, on only a few high-risk features, if this is the way to find the most bugs in the time available.
- **Block premature product releases.** This tester stops premature shipment by finding bugs so serious that no one would ship the product until they are fixed. For every release-decision meeting, the tester’s goal is to have new showstopper bugs.
- **Help managers make ship / no-ship decisions.** Managers are typically concerned with risk in the field. They want to know about coverage (maybe not the simplistic code coverage statistics, but some indicators of how much of the product has been addressed and how much is left), and how important the known problems are. Problems that appear significant on paper but will not lead to customer dissatisfaction are probably not relevant to the ship decision.
- **Minimize technical support costs.** Working in conjunction with a technical support or help desk group, the test team identifies the issues that lead to calls for support. These are often peripherally related to the product under test--for example, getting the product to work with a specific printer or to import data successfully from a third party database might prevent more calls than a low-frequency, data-corrupting crash.
- **Assess conformance to specification.** Any claim made in the specification is checked. Program characteristics not addressed in the specification are not (as part of *this objective*) checked.
- **Conform to regulations.** If a regulation specifies a certain type of coverage (such as, at least one test for every claim made about the product), the test group creates the appropriate tests. If the regulation specifies a style for the specifications or other documentation, the test group probably checks the style. In general, the test group is focusing on anything covered by regulation and (in the context of *this objective*) nothing that is not covered by regulation.
- **Minimize safety-related lawsuit risk.** Any error that could lead to an accident or injury is of primary interest. Errors that lead to loss of time or data or corrupt data, but that don’t carry a risk of injury or damage to physical things are out of scope.
- **Find safe scenarios for use of the product (*find ways to get it to work, in spite of the bugs*).** Sometimes, all that you’re looking for is one way to do a task that will consistently work--one set of instructions that someone else can follow that will reliably deliver the benefit they are supposed to lead to. In this case, the tester is not looking for bugs. He is trying out, empirically refining and documenting, a way to do a task.
- **Assess quality.** This is a tricky objective because quality is multi-dimensional. The nature of quality depends on the nature of the product. For example, a computer game that is rock solid but not entertaining is a lousy game. To *assess quality -- to measure and report back* on the level of quality -- you probably need a clear definition of the most important quality criteria for this product, and then you need a theory that relates test results to the definition. For example, *reliability* is not just about the number of bugs in the product. It is (or is often defined as being) about the number of reliability-related failures that can be expected in a period of time or a period of use. (*Reliability-related?* In measuring reliability, an organization might not care, for example, about misspellings in error messages.) To make this prediction, you need a mathematically and empirically sound

model that links test results to reliability. Testing involves gathering the data needed by the model. This might involve extensive work in areas of the product believed to be stable as well as some work in weaker areas. Imagine a reliability model based on counting bugs found (perhaps weighted by some type of severity) per N lines of code or per K hours of testing. Finding the bugs is important. Eliminating duplicates is important. Troubleshooting to make the bug report easier to understand and more likely to fix is (*in the context of assessment*) out of scope.

- **Verify correctness of the product.** It is impossible to do this by testing. You can prove that the product is *not* correct or you can demonstrate that you didn't find any errors in a given period of time using a given testing strategy. However, you can't test exhaustively, and the product might fail under conditions that you did not test. The best you can do (if you have a solid, credible model) is assessment--test-based estimation of the probability of errors. (See the discussion of reliability, above).
- **Assure quality.** Despite the common title, *quality assurance*, you can't assure quality by testing. You can't assure quality by gathering metrics. You can't assure quality by setting standards. Quality assurance involves building a high quality product and for that, you need skilled people throughout development who have time and motivation and an appropriate balance of direction and creative freedom. This is out of scope for a test organization. It is within scope for the project manager and associated executives. The test organization can certainly help in this process by performing a wide range of technical investigations, but those investigations are not quality assurance.

Given a testing objective, the good test series provides information directly relevant to that objective.

Tests Intended to Expose Defects

Let's narrow our focus to the test group that has two primary objectives:

- Find bugs that the rest of the development group will consider relevant (worth reporting) and
- Get these bugs fixed.

Even within these objectives, tests can be *good* in many different ways. For example, we might say that one test is better than another if it is:

- **More powerful.** I define power in the usual statistical sense as more likely to expose a bug if it the bug is there. Note that Test 1 can be more powerful than Test 2 for one type of bug and less powerful than Test 2 for a different type of bug.
- **More likely to yield significant (more motivating, more persuasive) results.** A problem is *significant* if a stakeholder with influence would protest if the problem is not fixed. (A stakeholder is a person who is affected by the product. A stakeholder with influence is someone whose preference or opinion might result in change to the product.)
- **More credible.** A credible test is more likely to be taken as a realistic (or reasonable) set of operations by the programmer or another stakeholder with influence. "Corner case" is an example of a phrase used by programmers to say that a test or bug is non-credible: "No one would do that." A test case is credible if some (or all) stakeholders agree that it is realistic.

- **Representative of events more likely to be encountered by the customer.** A population of tests can be designed to be highly credible. Set up your population to reflect actual usage probabilities. The more frequent clusters of activities are more likely to be covered or covered more thoroughly. (I say *cluster of activities* to suggest that many features are used together and so we might track which combinations of features are used and in what order, and reflect this more specific information in our analysis.) For more details, read Musa's (1998) work on software reliability engineering.
- **Easier to evaluate.** The question is, did the program pass or fail the test? Ease of Evaluation. The tester should be able to determine, quickly and easily, whether the program passed or failed the test. It is not enough that it is *possible* to tell whether the program passed or failed. The harder evaluation is, or the longer it takes, the more likely it is that failures will slip through unnoticed. Faced with time-consuming evaluation, the tester will take shortcuts and find ways to less expensively guess whether the program is OK or not. These shortcuts will typically be imperfectly accurate (that is, they may miss obvious bugs or they may flag correct code as erroneous.)
- **More useful for troubleshooting.** For example, high volume automated tests will often crash the system under test without providing much information about the relevant test conditions needed to reproduce the problem. They are not useful for troubleshooting. Tests that are harder to repeat are less useful for troubleshooting. Tests that are harder to perform are less likely to be performed correctly the next time, when you are troubleshooting a failure that was exposed by this test.
- **More informative.** A test provides value to the extent that we learn from it. In most cases, you learn more from the test that the program passes than the one the program fails, but the informative test will teach you something (reduce your uncertainty) whether the program passes it or fails.
 - For example, if we have already run a test in several builds, and the program reliably passed it each time, we will expect the program to pass this test again. Another "pass" result from the reused test doesn't contribute anything to our mental model of the program.
 - The notion of equivalence classes provides another example of information value. Behind any test is a set of tests that are sufficiently similar to it that we think of the other tests as essentially redundant with this one. In traditional jargon, this is the "equivalence class" or the "equivalence partition." If the tests are sufficiently similar, there is little added information to be obtained by running the second one after running the first.
 - This criterion is closely related to Karl Popper's theory of value of experiments (See Popper 1992). Good experiments involve risky predictions. The theory predicts something that many people would expect not to be true. Either your favorite theory is false or lots of people are surprised. Popper's analysis of what makes for good experiments (good tests) is a core belief in a mainstream approach to the philosophy of science.
 - Perhaps the essential consideration here is that the expected value of what you will learn from this test has to be balanced against the opportunity cost of

designing and running the test. The time you spend on this test is time you don't have available for some other test or other activity.

- **Appropriately complex.** A complex test involves many features, or variables, or other attributes of the software under test. Complexity is less desirable when the program has changed in many ways, or when you're testing many new features at once. If the program has many bugs, a complex test might fail so quickly that you don't get to run much of it. Test groups that rely primarily on complex tests complain of *blocking bugs*. A blocking bug causes many tests to fail, preventing the test group from learning the other things about the program that these tests are supposed to expose. Therefore, early in testing, simple tests are desirable. As the program gets more stable, or (as in eXtreme Programming or any evolutionary development lifecycle) as more stable features are incorporated into the program, greater complexity becomes more desirable.
- **More likely to help the tester or the programmer develop insight into some aspect of the product, the customer, or the environment.** Sometimes, we test to understand the product, to learn how it works or where its risks might be. Later, we might design tests to expose faults, but especially early in testing we are interested in learning *what it is* and *how to test it*. Many tests like this are never reused. However, in a *test-first design* environment, code changes are often made experimentally, with the expectation that the (typically, unit) test suite will alert the programmer to side effects. In such an environment, a test might be designed to flag a performance change, a difference in rounding error, or some other change that is not a defect. An unexpected change in program behavior might alert the programmer that her model of the code or of the impact of her code change is incomplete or wrong, leading her to additional testing and troubleshooting. (Thanks to Ward Cunningham and Brian Marick for suggesting this example.)

Test Styles/Types and Test Qualities

Within the field of black box testing, Kaner & Bach (see our course notes, Bach 2003b and Kaner, 2002, posted at www.testingeducation.org, and see Kaner, Bach & Pettichord, 2002) have described eleven dominant *styles* of black box testing:

- Function testing
- Domain testing
- Specification-based testing
- Risk-based testing
- Stress testing
- Regression testing
- User testing
- Scenario testing
- State-model based testing
- High volume automated testing
- Exploratory testing

Bach and I call these "paradigms" of testing because we have seen time and again that one or two of them dominate the thinking of a testing group or a talented tester. An analysis we find intriguing goes like this:

If I was a "scenario tester" (a person who defines testing primarily in terms of application of scenario tests), how would I actually test the program? What makes one scenario test better than another? Why types of problems would I tend to miss, what would be difficult for me to find or interpret, and what would be particularly easy?

Here are thumbnail sketches of the styles, with some thoughts on how test cases are "good" within them.

Function Testing

Test each function / feature / variable in isolation.

Most test groups start with fairly simple function testing but then switch to a different style, often involving the interaction of several functions, once the program passes the mainstream function tests.

Within this approach, a good test focuses on a single function and tests it with middle-of-the-road values. We don't expect the program to fail a test like this, but it will if the algorithm is fundamentally wrong, the build is broken, or a change to some other part of the program has fowled this code.

These tests are ***highly credible*** and ***easy to evaluate*** but not particularly powerful.

Some test groups spend most of their effort on function tests. For them, testing is complete when every item has been thoroughly tested on its own. In my experience, the tougher function tests look like domain tests and have their strengths.

Domain Testing

The essence of this type of testing is sampling. We reduce a massive set of possible tests to a small group by dividing (partitioning) the set into subsets (subdomains) and picking one or two representatives from each subset.

In domain testing, we focus on variables, initially one variable at time. To test a given variable, the set includes all the values (including invalid values) that you can imagine being assigned to the variable. Partition the set into subdomains and test at least one representative from each subdomain. Typically, you test with a "best representative", that is, with a value that is at least as likely to expose an error as any other member of the class. If the variable can be mapped to the number line, the best representatives are typically boundary values.

Most discussions of domain testing are about input variables whose values can be mapped to the number line. The best representatives of partitions in these cases are typically boundary cases.

A good set of domain tests for a numeric variable hits every boundary value, including the minimum, the maximum, a value barely below the minimum, and a value barely above the maximum.

- Whittaker (2003) provides an extensive discussion of the many different types of variables we can analyze in software, including input variables, output variables, results

of intermediate calculations, values stored in a file system, and data sent to devices or other programs.

- Kaner, Falk & Nguyen (1993) provided a detailed analysis of testing with a variable (printer type, in configuration testing) that can't be mapped to a number line.

These tests are **higher power** than tests that don't use "best representatives" or that skip some of the subdomains (e.g. people often skip cases that are expected to lead to error messages).

The first time these tests are run, or after significant relevant changes, these tests carry a lot of **information value** because boundary / extreme-value errors are common.

Bugs found with these tests are sometimes dismissed, especially when you test extreme values of several variables at the same time. (These tests are called *corner cases*.) They are not necessarily credible, they don't necessarily represent what customers will do, and thus they are not necessarily very motivating to stakeholders.

Specification-Based Testing

Check the program against every claim made in a reference document, such as a design specification, a requirements list, a user interface description, a published model, or a user manual.

These tests are **highly significant** (motivating) in companies that take their specifications seriously. For example, if the specification is part of a contract, conformance to the spec is very important. Similarly products must conform to their advertisements, and life-critical products must conform to any safety-related specification.

Specification-driven tests are often weak, not particularly powerful representatives of the class of tests that could test a given specification item.

Some groups that do specification-based testing focus narrowly on what is written in the document. To them, a good set of tests includes an unambiguous and relevant test for each claim made in the spec.

Other groups look further, for problems in the specification. They find that the most informative tests in a well-specified product are often the ones that explore ambiguities in the spec or examine aspects of the product that were not well-specified.

Risk-Based Testing

Imagine a way the program could fail and then design one or more tests to check whether the program will actually fail that way.

A "complete" set of risk-based tests would be based on an exhaustive risk list, a list of every way the program could fail.

A good risk-based test is a **powerful representative** of the class of tests that address a given risk.

To the extent that the tests tie back to significant failures in the field or well known failures in a competitor's product, a risk-based failure will be **highly credible** and **highly motivating**. However, many risk-based tests are dismissed as academic (unlikely to occur in real use). Being able to tie the "risk" (potential failure) you test for to a real failure in the field is very valuable, and makes tests more credible.

Risk-based tests tend to carry ***high information value*** because you are testing for a problem that you have some reason to believe might actually exist in the product. We learn a lot whether the program passes the test or fails it.

Stress Testing

There are a few different definition of stress tests.

- Under one common definition, you hit the program with a peak burst of activity and see it fail.
- IEEE Standard 610.12-1990 defines it as "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements with the goal of causing the system to fail."
- A third approach involves driving the program to failure in order to watch *how* the program fails. For example, if the test involves excessive input, you don't just test near the specified limits. You keep increasing the size or rate of input until either the program finally fails or you become convinced that further increases won't yield a failure. The fact that the program eventually fails might not be particularly surprising or motivating. The interesting thinking happens when you see the failure and ask what vulnerabilities have been exposed and which of them might be triggered under less extreme circumstances. Jorgensen (2003) provides a fascinating example of this style of work.

I work from this third definition.

These tests have ***high power***.

Some people dismiss stress test results as not representative of customer use, and therefore not credible and not motivating. Another problem with stress testing is that a failure may not be useful unless the test provides good troubleshooting information, or the lead tester is extremely familiar with the application.

A good stress test pushes the limit you want to push, and includes enough diagnostic support to make it reasonably easy for you to investigate a failure once you see it.

Some testers, such as Alberto Savoia (2000), use stress-like tests to expose failures that are hard to see if the system is not running several tasks concurrently. These failures often show up well within the theoretical limits of the system and so they are more ***credible*** and more ***motivating***. They are not necessarily easy to troubleshoot.

Regression Testing

Design, develop and save tests with the intent of regularly reusing them. Repeat the tests after making changes to the program.

This is a good point (consideration of regression testing) to note that this is not an orthogonal list of test types. You can put domain tests or specification-based tests or any other kinds of tests into your set of regression tests.

So what's the difference between these and the others? I'll answer this by example:

Suppose a tester creates a suite of domain tests and saves them for reuse. Is this domain testing or regression testing?

- I think of it as primarily domain testing if the tester is primarily thinking about partitioning variables and finding good representatives when she creates the tests.
- I think of it as primarily regression testing if the tester is primarily thinking about building a set of reusable tests.

Regression tests may have been powerful, credible, and so on, when they were first designed. However, after a test has been run and passed many times, it's not likely that the program will fail it the next time, unless there have been major changes or changes in part of the code directly involved with this test. Thus, most of the time, regression tests carry little information value.

A good regression test is designed for reuse. It is adequately documented and maintainable. (For suggestions that improve maintainability of GUI-level tests, see Graham & Fewster, 1999; Kaner, 1998; Pettichord, 2002, and the papers at www.pettichord.com in general). A good regression test is designed to be likely to fail if changes induce errors in the function(s) or area(s) of the program addressed by the regression test.

User Testing

User testing is done by users. Not by testers pretending to be users. Not by secretaries or executives pretending to be testers pretending to be users. By users. People who will make use of the finished product.

User tests might be designed by the users or by testers or by other people (sometimes even by lawyers, who included them as acceptance tests in a contract for custom software). The set of user tests might include boundary tests, stress tests, or any other type of test.

Some user tests are designed in such detail that the user merely executes them and reports whether the program passed or failed them. This is a good way to design tests if your goal is to provide a carefully scripted demonstration of the system, without much opportunity for wrong things to show up as wrong.

If your goal is to discover what problems a user will encounter in real use of the system, your task is much more difficult. Beta tests are often described as cheap, effective user tests but in practice they can be quite expensive to administer and they may not yield much information. For some suggestions on beta tests, see Kaner, Falk & Nguyen (1993).

A good user test must allow enough room for cognitive activity by the user while providing enough structure for the user to report the results effectively (in a way that helps readers understand and troubleshoot the problem).

Failures found in user testing are typically ***credible*** and ***motivating***. Few users run particularly powerful tests. However, some users run complex scenarios that put the program through its paces.

Scenario Testing

A scenario is a story that describes a hypothetical situation. In testing, you check how the program copes with this hypothetical situation.

The ideal scenario test is ***credible, motivating, easy to evaluate, and complex***.

In practice, many scenarios will be weak in at least one of these attributes, but people will still call them scenarios. The key message of this pattern is that you should keep these four attributes in mind when you design a scenario test and try hard to achieve them.

An important variation of the scenario test involves a harsher test. The story will often involve a sequence, or data values, that would rarely be used by typical users. They might arise, however, out of user error or in the course of an unusual but plausible situation, or in the behavior of a hostile user. Hans Buwalda (2000a, 2000b) calls these "killer soaps" to distinguish them from normal scenarios, which he calls "soap operas." Such scenarios are common in security testing or other forms of stress testing.

In the Rational Unified Process, scenarios come from use cases. (Jacobson, Booch, & Rumbaugh, 1999). Scenarios specify actors, roles, business processes, the goal(s) of the actor(s), and events that can occur in the course of attempting to achieve the goal. A scenario is an instantiation of a use case. A simple scenario traces through a single use case, specifying the data values and thus the path taken through the case. A more complex use case involves concatenation of several use cases, to track through a given task, end to end. (See also Bittner & Spence, 2003; Cockburn, 2000; Collard, 1999; Constantine & Lockwood, 1999; Wiegers, 1999.) For a cautionary note, see Berger (2001).

However they are derived, good scenario tests have **high power** the first time they're run.

Groups vary in how often they run a given scenario test.

- Some groups create a pool of scenario tests as regression tests.
- Others (like me) run a scenario once or a small number of times and then design another scenario rather than sticking with the ones they've used before.

Testers often develop scenarios to **develop insight** into the product. This is especially true early in testing and again late in testing (when the product has stabilized and the tester is trying to understand advanced uses of the product.)

State-Model-Based Testing

In state-model-based testing, you model the visible behavior of the program as a state machine and drive the program through the state transitions, checking for conformance to predictions from the model. This approach to testing is discussed extensively at www.model-based-testing.org.

In general, comparisons of software behavior to the model are done using automated tests and so the failures that are found are found easily (**easy to evaluate**).

In general, state-model-based tests are **credible, motivating** and **easy to troubleshoot**. However, state-based testing often involves simplifications, looking at transitions between operational modes rather than states, because there are too many states (El-Far 1995). Some abstractions to operational modes are obvious and credible, but others can seem overbroad or otherwise odd to some stakeholders, thereby reducing the value of the tests. Additionally, if the model is oversimplified, failures exposed by the model can be difficult to troubleshoot (Houghtaling, 2001).

Talking about his experiences in creating state models of software, Harry Robinson (2001) reported that much of the bug-finding happens while doing the modeling, well before the automated tests are coded. Elisabeth Hendrickson (2002) trains testers to work with state models

as an exploratory testing tool--her models might never result in automated tests, their value is that they guide the analysis by the tester.

El-Far, Thompson & Mottay (2001) and El-Far (2001) discuss some of the considerations in building a good suite of model-based tests. There are important tradeoffs, involving, for example, the level of detail (more detailed models find more bugs but can be much harder to read and maintain). For much more, see the papers at www.model-based-testing.org.

High-Volume Automated Testing

High-volume automated testing involves massive numbers of tests, comparing the results against one or more partial oracles.

- The simplest partial oracle is running versus crashing. If the program crashes, there must be a bug. See Nyman (1998, 2002) for details and experience reports.
- State-model-based testing can be high volume if the stopping rule is based on the results of the tests rather than on a coverage criterion. For the general notion of stochastic state-based testing, see Whittaker (1997). For discussion of state-model-based testing ended by a coverage stopping rule, see Al-Ghafees & Whittaker (2002).
- Jorgensen (2002) provides another example of high-volume testing. He starts with a file that is valid for the application under test. Then he corrupts it in many ways, in many places, feeding the corrupted files to the application. The application rejects most of the bad files and crashes on some. Sometimes, some applications lose control when handling these files. Buffer overruns or other failures allow the tester to take over the application or the machine running the application. Any program that will read any type of data stream can be subject to this type of attack if the tester can modify the data stream before it reaches the program.
- Kaner (2000) describes several other examples of high-volume automated testing approaches. One classic approach repeatedly feeds random data to the application under test and to another application that serves as a reference for comparison, an oracle. Another approach runs an arbitrarily long random sequence of regression tests, tests that the program has shown it can pass one by one. Memory leaks, stack corruption, wild pointers or other garbage that cumulates over time finally causes failures in these long sequences. Yet another approach attacks the program with long sequences of activity and uses probes (tests built into the program that log warning or failure messages in response to unexpected conditions) to expose problems.

High-volume testing is a diverse grouping. The essence of it is that the *structure* of this type of testing is designed by a person, but the individual test cases are developed, executed, and interpreted by the computer, which flags suspected failures for human review. The almost-complete automation is what makes it possible to run so many tests.

- The individual tests are often weak. They make up for low power with massive numbers.
- Because the tests are not handcrafted, some tests that expose failures may not be particularly credible or motivating. A skilled tester often works with a failure to imagine a broader or more significant range of circumstances under which the failure might arise, and then craft a test to prove it.

- Some high-volume test approaches yield failures that are very hard to troubleshoot. It is easy to see that the failure occurred in a given test, but one of the necessary conditions that led to the failure might have been set up thousands of tests before the one that actually failed. Building troubleshooting support into these tests is a design challenge that some test groups have tackled more effectively than others.

Exploratory Testing

Exploratory testing is “any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests” (Bach 2003a).

Bach points out that tests span a continuum between purely scripted (the tester does precisely what the script specifies and nothing else) to purely exploratory (none of the tester’s activities are pre-specified and the tester is not required to generate any test documentation beyond bug reports). Any given testing effort falls somewhere on this continuum. Even predominantly pre-scripted testing can be exploratory when performed by a skilled tester.

“In the prototypic case (what Bach calls “freestyle exploratory testing”), exploratory testers continually learn about the software they’re testing, the market for the product, the various ways in which the product could fail, the weaknesses of the product (including where problems have been found in the application historically and which developers tend to make which kinds of errors), and the best ways to test the software. At the same time that they’re doing all this learning, exploratory testers also test the software, report the problems they find, advocate for the problems they found to be fixed, and develop new tests based on the information they’ve obtained so far in their learning.” (Tinkham & Kaner, 2003)

An exploratory tester might use any type of test--domain, specification-based, stress, risk-based, any of them. The underlying issue is not what style of testing is best but what is most likely to reveal the information the tester is looking for at the moment.

Exploratory testing is not purely spontaneous. The tester might do extensive research, such as studying competitive products, failure histories of this and analogous products, interviewing programmers and users, reading specifications, and working with the product.

What distinguishes skilled exploratory testing from other approaches and from unskilled exploration, is that in the moments of doing the testing, the person who is doing exploratory testing well is fully engaged in the work, learning and planning as well as running the tests. Test cases are good to the extent that they advance the tester’s knowledge in the direction of his information-seeking goal. Exploratory testing is highly goal-driven, but the goal may change quickly as the tester gains new knowledge.

Concluding Notes

There’s no simple formula or prescription for generating “good” test cases. The space of interesting tests is too complex for this.

There are tests that are *good for your purposes*, for bringing forth the type of information that you’re seeking.

Many test groups, most of the ones that I've seen, stick with a few types of tests. They are primarily scenario testers or primarily domain testers, etc. As they get very good at their preferred style(s) of testing, their tests become, in some ways, excellent. Unfortunately, no style yields tests that are excellent in all of the ways we wish for tests. To achieve the broad range of value from our tests, we have to use a broad range of techniques.

Acknowledgements

This research was partially support by grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers."

Thanks to Andy Tinkham, Al Jorgenson, and Pat McGee for comments on previous drafts.

References

- Al-Ghafes, Mohammed; & Whittaker, James (2002) "Markov Chain-based Test Data Adequacy Criteria: A Complete Family", *Informing Science & IT Education Conference*, Cork, Ireland, <http://ecommerce.lebow.drexel.edu/eli/2002Proceedings/papers/AlGha180Marko.pdf> (accessed March 30, 2003)
- James Bach (2003a), "Exploratory Testing Explained", www.satisfice.com/articles/et-article.pdf (accessed March 30, 2003).
- Bach, James (2003b) Rapid Software Testing (course notes). www.testingeducation.org/coursenotes/bach_james/cm_200204_rapidtesting/ (accessed March 20, 2003).
- Berger, Bernie (2001) "The dangers of use cases employed as test cases," *STAR West* conference, San Jose, CA. www.testassured.com/docs/Dangers.htm. accessed March 30, 2003.
- Bittner, Kurt & Ian Spence (2003) *Use Case Modeling*, Addison-Wesley
- Buwalda, Hans (2000a) "The three holy grails of test development," presented at *EuroSTAR* conference.
- Buwalda, Hans (2000b) "Soap Opera Testing," presented at *International Software Quality Week Europe* conference, Brussels.
- Cockburn, Alistair (2000) *Writing Effective Use Cases*, Addison-Wesley.
- Collard, R. (1999). "Developing test cases from use cases," *Software Testing & Quality Engineering*, July/August, p. 31.
- Constantine, Larry, L & Lucy A.D. Lockwood (1999) *Software for Use*, ACM Press.
- El-Far, Ibrahim K. (1995) *Automated Construction of Software Behavior Models*, M.Sc. Thesis, Florida Tech, www.geocities.com/model_based_testing/elfar_thesis.pdf (accessed March 28, 2003).
- El-Far, Ibrahim K. (2001) "Enjoying the Perks of Model-Based Testing", *STAR West* conference, San Jose, CA. (Available at www.geocities.com/model_based_testing/perks_paper.pdf, accessed March 25, 2003).
- El-Far, Ibrahim, K; Thompson, Herbert; & Mottay, Florence (2001) "Experiences in Testing Pocket PC Applications," *International Software Quality Week Europe*, www.geocities.com/model_based_testing/pocketpc_paper.pdf (accessed March 30, 2003).
- Hendrickson, Elisabeth (2002) Bug Hunting (course, notes unpublished)
- Graham, Dorothy; & Fewster, Mark (1999) *Software Test Automation: Effective Use of Test Execution Tools*, ACM Press / Addison-Wesley.

Houghtaling, Mike (2001) Presentation at the *Workshop on Model-Based Testing*, Melbourne, Florida, February 2001.

Institute of Electrical & Electronics Engineers, Standard 610 (1990), reprinted in *IEEE Standards Collection: Software Engineering 1994 Edition*.

Jacobson, Ivar, Grady Booch & James Rumbaugh (1999) *The Unified Software Development Process*, Addison-Wesley.

Jorgensen, Alan (2003) "Testing With Hostile Data Streams,"
<http://streamer.it.fit.edu:8081/ramgen/cs/Spring03/CSE5500-Sp03-05/trainer.smi>

Kaner, Cem (1998), "Avoiding Shelfware: A Manager's View of Automated GUI Testing," Keynote address, *STAR '98 Conference*, Orlando, FL., www.kaner.com/pdfs/shelfwar.pdf (accessed March 28, 2003).

Kaner, Cem (2000), "Architectures of Test Automation," *STAR West conference*, San Jose, CA, www.kaner.com/testarch.html (accessed March 30, 2003).

Kaner, Cem (2002) *A Course in Black Box Software Testing (Professional Version)*, www.testingeducation.org/coursesnotes/kaner_cem/cm_200204_blackboxtesting/ (accessed March 20, 2003)

Kaner, Cem; Bach, James; & Pettichord, Bret (2002) *Lessons Learned in Software Testing*, Wiley.

Kaner, Cem; Falk, Jack; & Nguyen, Hung Quoc (1993) *Testing Computer Software*, Wiley.

Marick, Brian (1997) "Classic Testing Mistakes", *STAR East conference*, www.testing.com/writings/classic/mistakes.html (accessed March 17, 2003).

Marick, Brian (undated), documentation for the Multi test tool, www.testing.com/tools/multi/README.html (accessed March 17, 2003).

Musa, John (1998) *Software Reliability Engineering*, McGraw-Hill.

Nyman, N. (1998), "Application Testing with Dumb Monkeys", *STAR West conference*, San Jose, CA.

Nyman, N. (2002), "In Defense of Monkey Testing", www.softtest.org/sigs/material/nnyman2.htm (accessed March 30, 2003).

Patton, Ron (2001) *Software Testing*, SAMS.

Pettichord, Bret (2002) "Design for Testability" *Pacific Northwest Software Quality Conference*, October 2002, www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf (accessed March 28, 2003).

Popper, Karl (1992) *Conjectures and Refutations: The Growth of Scientific Knowledge*. 5th Edition. Routledge.

Robinson, Harry (2001) Presentation at the *Workshop on Model-Based Testing*, Melbourne, Florida, February 2001; For more on Robinson's experiences with state-model based testing, see www.geocities.com/model_based_testing/ (accessed March 20, 2003.)

Savoia, Alberto (2000) The Art and Science of Load Testing Internet Applications, *STAR West conference*, available at www.sticky minds.com.

Tinkham, Andy; & Kaner, Cem (2003) “Exploring Exploratory Testing”, STAR East conference, www.testingeducation.org/articles/exploring_exploratory_testing_star_east_2003_paper.pdf (accessed March 30, 2003).

Wiegers, Karl E. (1999) *Software Requirements*, Microsoft Press.

Whittaker, James (1997) “Stochastic Software Testing,” *Annals of Software Engineering* Vol. 4, 115-131.

Whittaker, James (2002) *Why Software Fails*, Addison-Wesley.

Process Models in Software Engineering

Walt Scacchi, Institute for Software Research, University of California, Irvine

February 2001

Revised Version, May 2001, October 2001

Final Version to appear in, J.J. Marciniak (ed.), *Encyclopedia of Software Engineering, 2nd Edition*, John Wiley and Sons, Inc, New York, December 2001.

Introduction

Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. This article categorizes and examines a number of methods for describing or modeling how software systems are developed. It begins with background and definitions of traditional software life cycle models that dominate most textbook discussions and current software development practices. This is followed by a more comprehensive review of the alternative models of software evolution that are of current use as the basis for organizing software engineering projects and technologies.

Background

Explicit models of software evolution date back to the earliest projects developing large software systems in the 1950's and 1960's (Hosier 1961, Royce 1970). Overall, the apparent purpose of these early software life cycle models was to provide a conceptual scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

Since the 1960's, many descriptions of the classic software life cycle have appeared (e.g., Hosier 1961, Royce 1970, Boehm 1976, Distaso 1980, Scacchi 1984, Somerville 1999). Royce (1970) originated the formulation of the software life cycle using the now familiar "waterfall" chart, displayed in Figure 1. The chart summarizes in a single display how developing large software systems is difficult because it involves complex engineering tasks that may require iteration and rework before completion. These charts are often employed during introductory presentations, for people (e.g., customers of custom software) who may be unfamiliar with the various technical problems and strategies that must be addressed when constructing large software systems (Royce 1970).

These classic software life cycle models usually include some version or subset of the following activities:

- *System Initiation/Planning*: where do systems come from? In most situations, new

feasible systems replace or supplement existing information processing mechanisms whether they were previously automated, manual, or informal.

- *Requirement Analysis and Specification*: identifies the problems a new software system is suppose to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance.
- *Functional Specification or Prototyping*: identifies and potentially formalizes the objects of computation, their attributes and relationships, the operations that transform these objects, the constraints that restrict system behavior, and so forth.
- *Partition and Selection* (Build vs. Buy vs. Reuse): given requirements and functional specifications, divide the system into manageable pieces that denote logical subsystems, then determine whether new, existing, or reusable software systems correspond to the needed pieces.
- *Architectural Design and Configuration Specification*: defines the interconnection and resource interfaces between system subsystems, components, and modules in ways suitable for their detailed design and overall configuration management.
- *Detailed Component Design Specification*: defines the procedural methods through which the data resources within the modules of a component are transformed from required inputs into provided outputs.
- *Component Implementation and Debugging*: codifies the preceding specifications into operational source code implementations and validates their basic operation.
- *Software Integration and Testing*: affirms and sustains the overall integrity of the software system architectural configuration through verifying the consistency and completeness of implemented modules, verifying the resource interfaces and interconnections against their specifications, and validating the performance of the system and subsystems against their requirements.
- *Documentation Revision and System Delivery*: packaging and rationalizing recorded system development descriptions into systematic documents and user guides, all in a form suitable for dissemination and system support.
- *Deployment and Installation*: providing directions for installing the delivered software into the local computing environment, configuring operating systems parameters and user access privileges, and running diagnostic test cases to assure the viability of basic system operation.
- *Training and Use*: providing system users with instructional aids and guidance for understanding the system's capabilities and limits in order to effectively use the system.
- *Software Maintenance*: sustaining the useful operation of a system in its host/target environment by providing requested functional enhancements, repairs, performance

improvements, and conversions.

What is a software life cycle model?

A software life cycle model is either a descriptive or prescriptive characterization of how software is or should be developed. A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes, or for building empirically grounded prescriptive models (Curtis, Krasner, Iscoe, 1988). A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. This means that many idiosyncratic details that describe how a software system is built in practice can be ignored, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and techniques for using a given set of software engineering tools or environment during a development project.

Descriptive life cycle models, on the other hand, characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years. Also, descriptive models are specific to the systems observed and only generalizable through systematic comparative analysis. Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

These two characterizations suggest that there are a variety of purposes for articulating software life cycle models. These characterizations serve as a

- Guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
- Prescriptive outline for what documents to produce for delivery to client.
- Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
- Framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle (Boehm 1981)
- Basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

What is a software process model?

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing.

Software process networks can be viewed as representing multiple interconnected task chains (Kling 1982, Garg 1989). Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed as non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. Winograd and others have referred to these units of cooperative work between people and computers as "structured discourses of work" (Winograd 1986), while task chains have become popularized under the name of "workflow" (Bolcer 1998).

Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design.

Task chains join or split into other task chains resulting in an overall production network or web (Kling 1982). The production web represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how a software system is developed, used, and maintained. However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic

foul-ups that can emerge in the real world of software development (Bendifallah 1989, Mi 1990). Thus, any software production web will in some way realize only an approximate or incomplete description of software development.

Articulation work is a kind of unanticipated task that is performed when a planned task chain is inadequate or breaks down. It is work that represents an open-ended non-deterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain (Bendifallah 1987, Grinter 1996, Mi 1990, Mi 1996, Scacchi and Mi 1997). Thus, descriptive task chains are employed to characterize the observed course of events and situations that emerge when people try to follow a planned task sequence. Articulation work in the context of software evolution includes actions people take that entail either their accommodation to the contingent or anomalous behavior of a software system, or negotiation with others who may be able to affect a system modification or otherwise alter current circumstances (Bendifallah 1987, Grinter 1996, Mi 1990, Mi 1996, Scacchi and Mi 1997). This notion of articulation work has also been referred to as software process dynamism.

Traditional Software Life Cycle Models

Traditional models of software evolution have been with us since the earliest days of software engineering. In this section, we identify four. The classic software life cycle (or "waterfall chart") and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering. The incremental release model is closely related to industrial practices where it most often occurs. Military standards based models have also reified certain forms of the classic life cycle model into required practice for government contractors. Each of these four models uses coarse-grain or macroscopic characterizations when describing software evolution. The progressive steps of software evolution are often described as stages, such as requirements specification, preliminary design, and implementation; these usually have little or no further characterization other than a list of attributes that the product of such a stage should possess. Further, these models are independent of any organizational development setting, choice of programming language, software application domain, etc. In short, the traditional models are context-free rather than context-sensitive. But as all of these life cycle models have been in use for some time, we refer to them as the traditional models, and characterize each in turn.

Classic Software Life Cycle

The classic software life cycle is often represented as a simple prescriptive waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in order (Royce 1970). Such models resemble finite state machine descriptions of software evolution. However, these models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purposes (Royce 1970, Boehm 1976). Alternatively, these classic models have been widely characterized as both poor descriptive and prescriptive models of how software development "in-the-small" or "in-the-large" can or should occur. Figure 1 provides a common view of the waterfall model for software development attributed to Royce (1970).

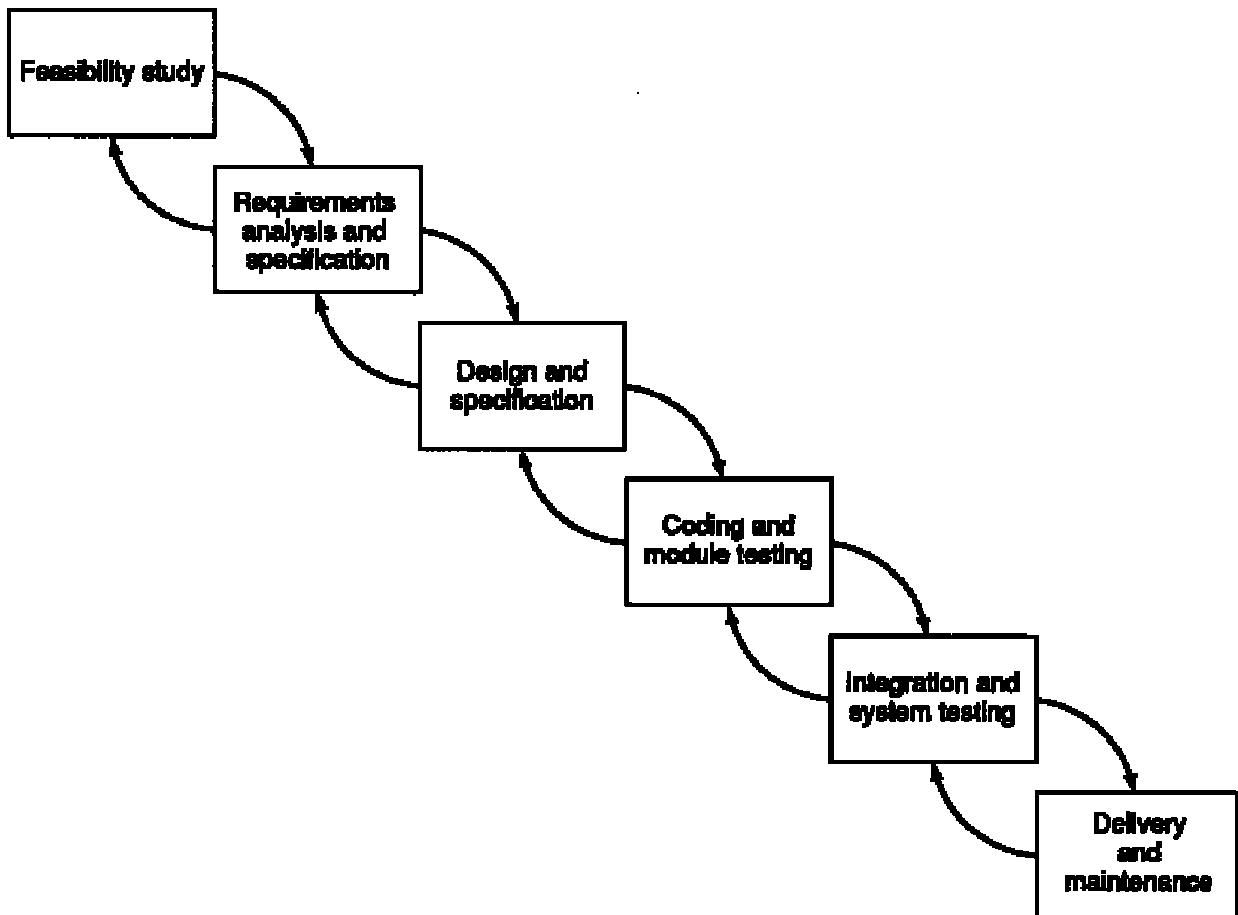


Figure 1. The Waterfall Model of Software Development (Royce 1970)

Stepwise Refinement

In this approach, software systems are developed through the progressive refinement and enhancement of high-level system specifications into source code components (Wirth 1971, Mili 1986). However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has been most effective and widely applied in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle thus subsume this approach within their design and implementations.

Incremental Development and Release

Developing systems through incremental release requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals (Basili 1975). This model combines the classic software life cycle with iterative enhancement at the level of system development organization. It also supports a strategy to

periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software firms and system vendors. This approach has also been extended through the use of software prototyping tools and techniques (described later), which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation (Graham 1989). Figure 2 provides an example view of an incremental development, build, and release model for engineering large Ada-based software systems, developed by Royce (1990) at TRW.

Elsewhere, the Cleanroom software development method at use in IBM and NASA laboratories provides incremental release of software functions and/or subsystems (developed through stepwise refinement) to separate in-house quality assurance teams that apply statistical measures and analyses as the basis for certifying high-quality software systems (Selby 1987, Mills 1987).

Industrial and Military Standards, and Capability Models

Industrial firms often adopt some variation of the classic model as the basis for standardizing their software development practices (Royce 1970, Boehm 1976, Distaso 1980, Humphrey 1985, Scacchi 1984, Somerville 1999). Such standardization is often motivated by needs to simplify or eliminate complications that emerge during large software development or project management.

From the 1970's through the present, many government contractors organized their software development activities according to succession of military software standards such as MIL-STD-2167A, MIL-STD 498, and IEEE-STD-016. ISO12207 (Moore 1997) is now the standard that most such contractors now follow. These standards are an outgrowth of the classic life cycle activities, together with the documents required by clients who procure either software systems or complex platforms with embedded software systems. Military software system are often constrained in ways not found in industrial or academic practice, including: (1) required use of military standard computing equipment (which is often technologically dated and possesses limited processing capabilities); (2) are embedded in larger systems (e.g., airplanes, submarines, missiles, command and control systems) which are mission-critical (i.e., those whose untimely failure could result in military disadvantage and/or life-threatening risks); (3) are developed under contract to private firms through cumbersome procurement and acquisition procedures that can be subject to public scrutiny and legislative intervention; and (4) many embedded software systems for the military are among the largest and most complex systems in the world (Moore 1997). Finally, the development of custom software systems using commercial off-the-shelf (COTS) components or products is a recent direction for government contractors, and thus represents new challenges for how to incorporate a component-based development into the overall software life cycle. Accordingly, new software life cycle models that exploit COTS components will continue to appear in the next few years.

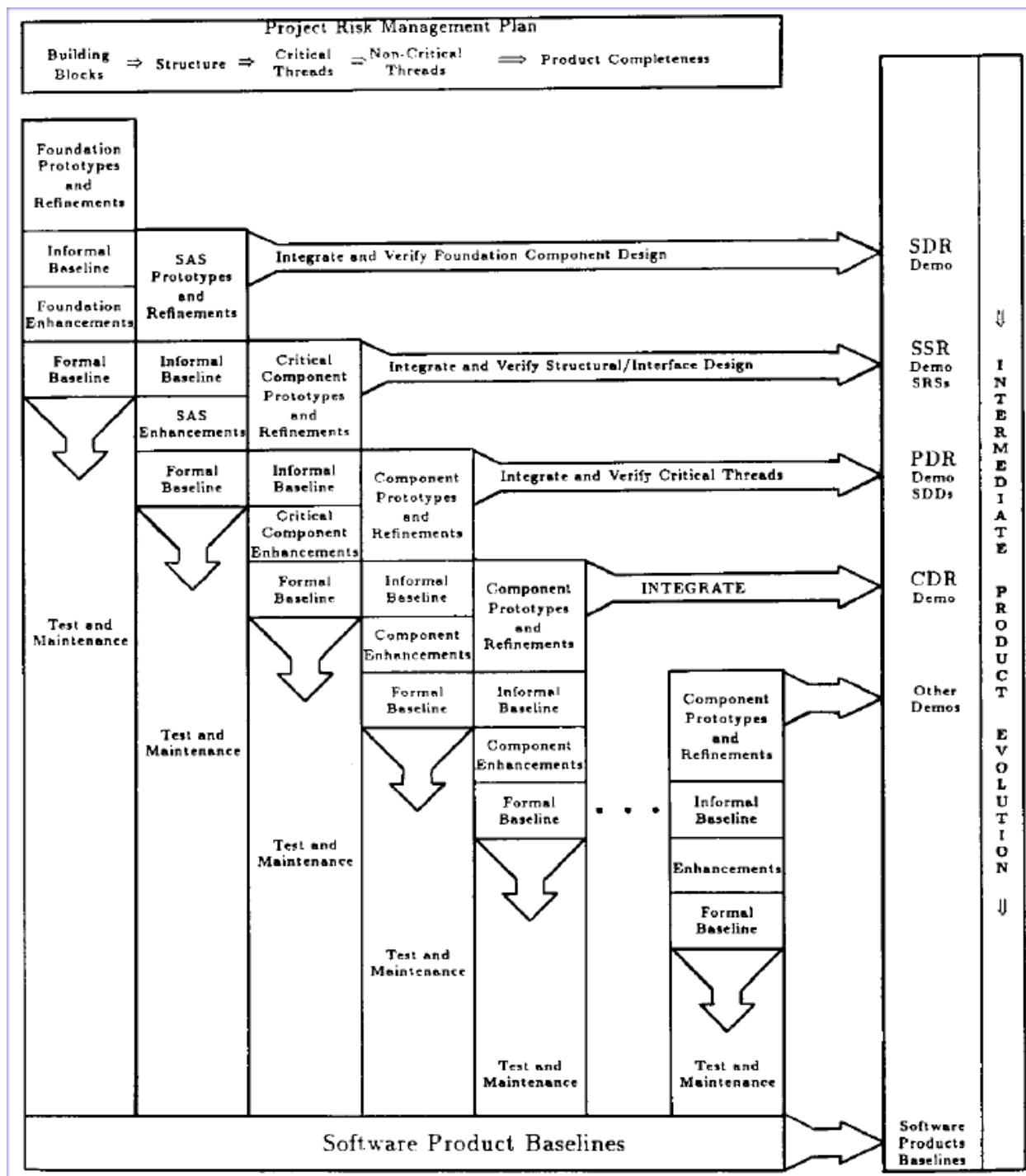


Figure 2. An Incremental Development, Build, and Release Model (Royce 1990)

In industrial settings, standard software development models often provide explicit detailed guidelines for how to deploy, install, customize or tune a new software system release in its operating application environment. In addition, these standards are intended to be compatible with provision of software quality assurance, configuration management, and independent verification and validation services in a multi-contractor development project. Early efforts in monitoring and measuring software process performance found in industrial practice appear in (Humphrey 1985, Radice 1985, Basili 1988). These efforts in turn help pave the way for what many software development organizations now practice, or have been certified to practice, software process capability assessments, following the Capability Maturity Model developed by the Software Engineering Institute (Paulk 1995) (see Capability Maturity Model for Software).

Alternatives to the Traditional Software Life Cycle Models

There are at least three alternative sets of models of software development. These models are alternatives to the traditional software life cycle models. These three sets focus of attention to either the products, production processes, or production settings associated with software development. Collectively, these alternative models are finer-grained, often detailed to the point of computational formalization, more often empirically grounded, and in some cases address the role of new automated technologies in facilitating software development. As these models are not in widespread practice, we examine each set of models in the following sections.

Software Product Development Models

Software products represent the information-intensive artifacts that are incrementally constructed and iteratively revised through a software development effort. Such efforts can be modeled using software product life cycle models. These product development models represent an evolutionary revision to the traditional software life cycle models (MacCormack 2001). The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software development may be implicit in the use of the technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favorable experiences with these technologies substantiate their use. Thus, detailed examination of these models is most appropriate when such technologies are available for use or experimentation.

Rapid Prototyping and Joint Application Development

Prototyping is a technique for providing a reduced functionality or a limited performance version of a software system early in its development (Balzer 1983, Budde 1984, Hekmatpour 1987). In contrast to the classic system life cycle, prototyping is an approach whereby more emphasis, activity, and processing are directed to the early stages of software development (requirements analysis and functional specification). In turn, prototyping can more directly accommodate early

user participation in determining, shaping, or evaluating emerging system functionality. Therefore, these up-front concentrations of effort, together with the use of prototyping technologies, seeks to trade-off or otherwise reduce downstream software design activities and iterations, as well as simplify the software implementation effort. (see Rapid Prototyping)

Software prototypes come in different forms including throwaway prototypes, mock-ups, demonstration systems, quick-and-dirty prototypes, and incremental evolutionary prototypes (Hekmatpour 1987). Increasing functionality and subsequent evolvability is what distinguishes the prototype forms on this list.

Prototyping technologies usually take some form of software functional specifications as their starting point or input, which in turn is simulated, analyzed, or directly executed. These technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. User evaluations can then be incorporated as feedback to refine the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continual revising/refining the input specifications. This has the advantage of always providing a working version of the emerging system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing throwaway or demonstration systems, or for building prototypes by reusing part/all of some existing software systems. Subsequently, it becomes clear why modern models of software development like the Spiral Model (described later) and the ISO 12207 expect that prototyping will be a common activity that facilitates the capture and refinement of software requirements, as well as overall software development.

Joint Application Development (JAD) is a technique for engaging a group or team of software developers, testers, customers, and prospective end-users in a collaborative requirements elicitation and prototyping effort (Wood and Silver 1995). JAD is quintessentially a technique for facilitating group interaction and collaboration. Consultants often employ JAD or external software system vendors who have been engaged to build a custom software system for use in a particular organizational setting. The JAD process is based on four ideas:

1. People who actually work at a job have the best understanding of that job.
2. People who are trained in software development have the best understanding of the possibilities of that technology.
3. Software-based information systems and business processes rarely exist in isolation -- they transcend the confines of any single system or office and effect work in related departments. People working in these related areas have valuable insight on the role of a system within a larger community.
4. The best information systems are designed when all of these groups work together on a project as equal partners.

Following these ideas, it should be possible for JAD to cover the complete development life cycle of a system. The JAD is usually a 3 to 6 month well-defined project, when systems can be

constructed from commercially available software products that do not require extensive coding or complex systems integration. For large-scale projects, it is recommended that the project be organized as an incremental development effort, and that separate JAD's be used for each increment (Wood and Silver 1995). Given this formulation, it is possible to view open source software development projects that rely on group email discussions among globally distributed users and developers, together with Internet-based synchronized version updates (Fogel 1999, Mockus 2000), as an informal variant of JAD.

Assembling Reusable Components

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems (Biggerstaff 1984, Neighbors 1984, Goguen 1986). Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software domain models, system specifications, designs, test case suites, and other software abstractions may themselves be treated as reusable software development components. These components may have a greater potential for favorable impact on reuse and semi-automated system generation or composition (Batory et al., 1994, Neighbors 1984). Therefore, assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.

The basic dilemmas encountered with reusable software componentry include (a) acquiring, analyzing and modeling a software application domain, (b) how to define an appropriate software part naming or classification scheme, (c) collecting or building reusable software components, (d) configuring or composing components into a viable application, and (e) maintaining and searching a components library. In turn, each of these dilemmas is mitigated or resolved in practice through the selection of software component granularity.

The granularity of the components (i.e., size, complexity, and functional capability) varies greatly across different approaches. Most approaches attempt to utilize components similar to common (textbook) data structures with algorithms for their manipulation: small-grain components. However, the use/reuse of small-grain components in and of itself does not constitute a distinct approach to software development. Other approaches attempt to utilize components resembling functionally complete systems or subsystems (e.g., user interface management system): large-grain components. The use/reuse of large-grain components guided by an application domain analysis and subsequent mapping of attributed domain objects and operations onto interrelated components does appear to be an alternative approach to developing software systems (Neighbors 1984), and thus is an area of active research.

There are many ways to utilize reusable software components in evolving software systems. However, the cited studies suggest their initial use during architectural or component design specification as a way to speed implementation. They might also be used for prototyping purposes if a suitable software prototyping technology is available.

Application Generation

Application generation is an approach to software development similar to reuse of

parameterized, large-grain software source code components. Such components are configured and specialized to an application domain via a formalized specification language used as input to the application generator. Common examples provide standardized interfaces to database management system applications, and include generators for reports, graphics, user interfaces, and application-specific editors (Batory, et al. 1994, Horowitz 1985).

Application generators give rise to a model of software development whereby traditional software design activities are either all but eliminated, or reduced to a data base design problem. The software design activities are eliminated or reduced because the application generator embodies or provides a generic software design that should be compatible with the application domain. However, users of application generators are usually expected to provide input specifications and application maintenance services. These capabilities are possible since the generators can usually only produce software systems specific to a small number of similar application domains, and usually those that depend on a data base management system.

Software Documentation Support Environments

Much of the focus on developing software products draws attention to the tangible software artifacts that result. Most often, these products take the form of documents: commented source code listings, structured design diagrams, unit development folders, etc. These documents characterize what the developed system is suppose to do, how it does it, how it was developed, how it was put together and validated, and how to install, use, and maintain it. Thus, a collection of software documents records the passage of a developed software system through a set of life cycle stages.

It seems reasonable that there will be models of software development that focus attention to the systematic production, organization, and management of the software development documents. Further, as documents are tangible products, it is common practice when software systems are developed under contract to a private firm, that the delivery of these documents is a contractual stipulation, as well as the basis for receiving payment for development work already performed. Thus, the need to support and validate conformance of these documents to software development and quality assurance standards emerges. However, software development documents are often a primary medium for communication between developers, users, and maintainers that spans organizational space and time. Thus, each of these groups can benefit from automated mechanisms that allow them to browse, query, retrieve, and selectively print documents (Garg and Scacchi, 1989, 1990). As such, we should not be surprise to see construction and deployment of software environments that provide ever increasing automated support for engineering the software documentation life cycle (e.g., Penedo 1985, Horowitz 1986, Garg and Scacchi, 1989, 1990), or how these capabilities have since become part of the commonly available computer-aided software engineering (CASE) tools suites like Rational Rose, and others based on the use of the Unified Modeling Language (UML).

Rapid Iteration, Incremental Evolution, and Evolutionary Delivery

There are a growing number of technological, social and economic trends that are shaping how a new generation of software systems are being developed that exploit the Internet and World

Wide Web. These include the synchronize and stabilize techniques popularized by Microsoft and Netscape at the height of the fiercely competitive efforts to dominate the Web browser market of the mid 1990's (Cusumano and Yoffie, 1999, MacCormack 2001). They also include the development of open source software systems that rely on a decentralized community of volunteer software developers to collectively develop and test software systems that are incrementally enhanced, released, experienced, and debugged in an overall iterative and cyclic manner (DiBona 1999, Fogel 1999, Mockus 2000). The elapsed time of these incremental development life cycles on some projects may be measured in weeks, days, or hours! The centralized planning, management authority and coordination imposed by the traditional system life cycle model has been discarded in these efforts, replaced instead by a more organic, participatory, reputation-based, and community oriented engineering practice. Software engineering in the style of rapid iteration and incremental evolution is one that focuses on and celebrates the inevitability of constantly shifting system requirements, unanticipated situations of use and functional enhancement, and the need for developers to collaborate with one another, even when they have never met (Truex 1999). As such, we are likely to see more research and commercial development aimed at figuring out whether or how software process models can accommodate rapid iteration, incremental evolution, or synchronize and stabilize techniques whether applied to closed, centrally developed systems, or to open, de-centrally developed systems.

Program Evolution Models

In contrast to the preceding four prescriptive product development models, Lehman and Belady sought to develop a descriptive model of software product evolution. They conducted a series of empirical studies of the evolution of large software systems at IBM during the 1970's (Lehman1985). (see Software Evolution) Based on their investigations, they identify five properties that characterize the evolution of large software systems. These are:

1. *Continuing change*: a large software system undergoes continuing change or becomes progressively less useful
2. *Increasing complexity*: as a software system evolves, its complexity increases unless work is done to maintain or reduce it
3. *Fundamental law of program evolution*: program evolution, the programming process, and global measures of project and system attributes are statistically self-regulating with determinable trends and invariances
4. *Invariant work rate*: the rate of global activity in a large software project is statistically invariant
5. *Incremental growth limit*: during the active life of a large program, the volume of modifications made to successive releases is statistically invariant.

However, it is important to observe that these are global properties of large software systems, not causal mechanisms of software development. More recent advances in the study of program evolution can be found elsewhere in the article by Lehman and Ramil (See Software Evolution

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

interconnected life cycle models that accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models. However, reports indicate that in general most software process models are exploratory, though there is now a growing base of experimental or industrial experience with these models (Basili 1988, Raffo *et al* 1999, Raffo and Scacchi 2000).

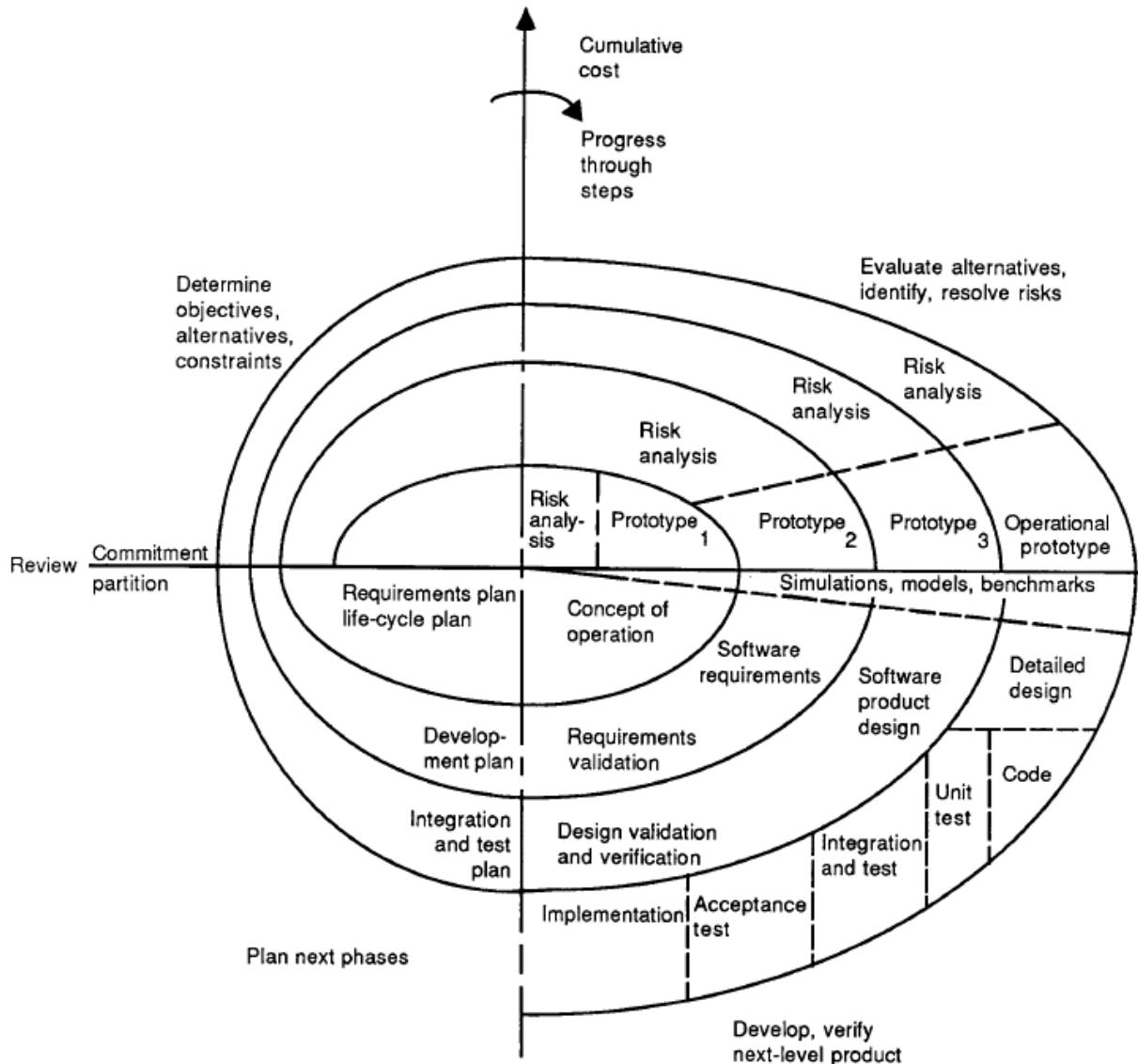


Figure 3.The Spiral Model diagram from (Boehm 1987)

Operational Process Models

In contrast to the preceding non-operational process models, many models are now beginning to appear which codify software engineering processes in computational terms--as programs or

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

development processes (Ould 1988, Wood 1988, Mi 1996). Last, recent research has shown how to combine different life cycle, product, and production process models within a process-driven framework that integrates both conventional and knowledge-based software engineering tools and environments (Garg 1994, Heineman 1994, Scacchi and Mi 1997).

Software process automation and programming. Process automation and programming are concerned with developing formal specifications of how a system or family of software systems should be developed. Such specifications therefore provide an account for the organization and description of various software production task chains, how they interrelate, when then can iterate, etc., as well as what software tools to use to support different tasks, and how these tools should be used (Hoffnagel 1985, Osterweil 1987). Focus then converges on characterizing the constructs incorporated into the language for specifying and programming software processes. Accordingly, discussion then turns to examine the appropriateness of language constructs for expressing rules for backward and forward-chaining, behavior, object type structures, process dynamism, constraints, goals, policies, modes of user interaction, plans, off-line activities, resource commitments, etc. across various levels of granularity (Garg and Scacchi 1989, Kaiser 1988, Mi and Scacchi 1992, Williams 1988, Yu and Mylopoulos 1994). This in turn implies that conventional mechanisms such as operating system shell scripts (e.g., Makefiles on Unix) do not support the kinds of software process automation these constructs portend.

Lehman (1987) and Curtis and associates, (1987) provide provocative critiques of the potential and limitations of current proposals for software process automation and programming. Their criticisms, given our framework, essentially point out that many process programming proposals (as of 1987) were focused almost exclusively to those aspects of software engineering that were amenable to automation, such as tool sequence invocation. They point out how such proposals often fail to address how the production settings and products constrain and interact with how the software production process is defined and performed, as revealed in recent empirical software process studies (Bendifallah 1987, Curtis, et al., 1988, Bendifallah 1989, Grinter 1996).

Beyond these, the dominant trend during the 1990's associated with software process automation was the development of process-centered software engineering environments (Garg 1996). Dozens of research projects and some commercial developments were undertaken to develop, experiment with, and evaluate the potential opportunities and obstacles associated with software environments driven by operational software process models. Many alternative process model formalisms were tried including knowledge-based representations, rule-based schemes, and Petri-net schemes and variations. In the early 1990's, emphasis focused on the development of distributed client-server environments that generally relied on a centralized server. The server might then interpret a process model for how to schedule, coordinate, or reactively synchronize the software engineering activities of developers working with client-side tools (Garg et al 1994, Garg 1996, Heineman 1994, Scacchi and Mi 1997). To no surprise, by the late 1990's emphasis has shifted towards environment architectures that employed decentralized servers for process support, workflow automation, data storage, and tool services (Bolcer 1998, Grundy 1999, Scacchi and Noll 1997). Finally, there was also some effort to expand the scope of operational support bound to process models in terms that recognized their growing importance as a new kind of software (Osterweil 1987). Here we began to see the emergence of process engineering environments that support their own class of life cycle activities and support mechanisms (Garg

and Jazayeri 1996, Garg et al 1994, Heineman 1994, Scacchi and Mi 1997, Scacchi and Noll 1997).

Emerging Trends and New Directions

In addition to the ongoing interest, debate, and assessment of process-centered or process-driven software engineering environments that rely on process models to configure or control their operation (Ambriola 1999, Garg and Jazayeri 1996), there are a number of promising avenues for further research and development with software process models. These opportunities areas and sample direction for further exploration include:

- Software process simulation (Raffo et al, 1999, Raffo and Scacchi 2000) efforts which seek to determine or experimentally evaluate the performance of classic or operational process models using a sample of alternative parameter configurations or empirically derived process data (cf. Cook and Wolf 1998). Simulation of empirically derived models of software evolution or evolutionary processes also offer new avenues for exploration (Chatters, Lehman, *et al.*, 2000, Mockus 2000).
- Web-based software process models and process engineering environments (Bolcer 1998, Grundy 1998, Penedo 2000, Scacchi and Noll 1997) that seek to provide software development workspaces and project support capabilities that are tied to adaptive process models. (see Engineering Web Applications with Java)
- Software process and business process reengineering (Scacchi and Mi 1997, Scacchi and Noll 1997, Scacchi 2000) which focuses attention to opportunities that emerge when the tools, techniques, and concepts for each disciplined are combined to their relative advantage. This in turn is giving rise to new techniques for redesigning, situating, and optimizing software process models for specific organizational and system development settings (Scacchi and Noll 1997, Scacchi 2000). (see Business Reengineering in the Age of the Internet)
- Understanding, capturing, and operationalizing process models that characterize the practices and patterns of globally distributed software development associated with open source software (DiBona 1999, Fogel 1999, Mockus 2000), as well as other emerging software development processes, such as extreme programming (Beck 1999) and Web-based virtual software development enterprises or workspaces (Noll and Scacchi 1999,2001, Penedo 2000).

Conclusions

The central thesis of this chapter is that contemporary models of software development must account for software the interrelationships between software products and production processes, as well as for the roles played by tools, people and their workplaces. Modeling these patterns can utilize features of traditional software life cycle models, as well as those of automatable software process models. Nonetheless, we must also recognize that the death of the traditional system life cycle model may be at hand. New models for software development enabled by the Internet,

group facilitation and distant coordination within open source software communities, and shifting business imperatives in response to these conditions are giving rise to a new generation of software processes and process models. These new models provide a view of software development and evolution that is incremental, iterative, ongoing, interactive, and sensitive to social and organizational circumstances, while at the same time, increasingly amenable to automated support, facilitation, and collaboration over the distances of space and time.

References

- Ambriola, V., R. Conradi and A. Fuggetta, Assessing process-centered software engineering environments, *ACM Trans. Softw. Eng. Methodol.* 6, 3, 283-328, 1997.
- Balzer, R., Transformational Implementation: An Example, *IEEE Trans. Software Engineering*, 7, 1, 3-14, 1981.
- Balzer, R., A 15 Year Perspective on Automatic Programming, *IEEE Trans. Software Engineering*, 11, 11, 1257-1267, 1985.
- Balzer, R., T. Cheatham, and C. Green, Software Technology in the 1990's: Using a New Paradigm, *Computer*, 16, 11, 39-46, 1983.
- Basili, V.R. and H.D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Trans. Soft. Engr.*, 14, 6, 759-773, 1988.
- Basili, V. R., and A. J. Turner, Iterative Enhancement: A Practical Technique for Software Development, *IEEE Trans. Software Engineering*, 1, 4, 390-396, 1975.
- Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin, The GenVoca model of software-system generators, *IEEE Software*, 11(5), 89-94, September 1994.
- Bauer, F. L., Programming as an Evolutionary Process, *Proc. 2nd. Intern. Conf. Software Engineering*, IEEE Computer Society, 223-234, January, 1976.
- Beck, K. *Extreme Programming Explained*, Addison-Wesley, Palo Alto, CA, 1999.
- Bendifallah, S., and W. Scacchi, Understanding Software Maintenance Work, *IEEE Trans. Software Engineering*, 13, 3, 311-323, 1987.
- Bendifallah, S. and W. Scacchi, Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork, *Proc. 11th. Intern. Conf. Software Engineering*, IEEE Computer Society, 260-270, 1989.
- Biggerstaff, T., and A. Perlis (eds.), Special Issues on Software Reusability, *IEEE Trans. Software Engineering*, 10, , 5, 1984.
- Boehm, B., Software Engineering, *IEEE Trans. Computer*, C-25, 12, 1226-1241, 1976.
- Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N. J., 1981

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.

Distaso, 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 Environment*, 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 Technology*, 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

MacCormack, A., Product-Development Practices that Work: How Internet Companies Build Software, *Sloan Management Review*, 75-84, Winter 2001.

Mi, P. and W. Scacchi, A Knowledge Base Environment for Modeling and Simulating Software Engineering Processes, *IEEE Trans. Knowledge and Data Engineering*, 2,3, 283-294, 1990.

Mi, P. and W. Scacchi, Process Integration for CASE Environments, *IEEE Software*, 9,2, March,45-53,1992.

Mi, P. and W. Scacchi., A Meta-Model for Formulating Knowledge-Based Models of Software Development. *Decision Support Systems*, 17(4):313-330, 1996.

Mili, A., J. Desharnais, and J.R. Gagne, Formal Models of Stepwise Refinement of Programs, *ACM Computing Surveys*, 18, 3, 231-276, 1986.

Mills, H.D., M. Dyer and R.C. Linger, Cleanroom Software Engineering, *IEEE Software*, 4, 5, 19-25, 1987.

Mockus, A., R.T. Fielding, and J. Herbsleb, A Case Study of Open Software Development: The Apache Server, *Proc. 22nd. International Conf. Software Engineering*, Limerick, IR, 263-272, 2000.

Moore, J.W., P.R. DeWeese, and D. Rilling, "U. S. Software Life Cycle Process Standards," *Crosstalk: The DoD Journal of Software Engineering*, 10:7, July 1997

Neighbors, J., The Draco Approach to Constructing Software from Reusable Components, *IEEE Trans. Software Engineering*, 10, 5, 564-574, 1984.

Noll, J. and W. Scacchi, Supporting Software Development in Virtual Enterprises, *Journal of Digital Information*, 1(4), February 1999.

Noll, J. and W. Scacchi, Specifying Process-Oriented Hypertext for Organizational Computing, *J. Network and Computer Applications*, 24(1):39-61, 2001.

Osterweil, L., Software Processes are Software Too, *Proc. 9th. Intern. Conf. Software Engineering*, 2-13, IEEE Computer Society, 1987.

Ould, M.A., and C. Roberts, Defining Formal Models of the Software Development Process, *Software Engineering Environments*, P. Brereton (ed.), Ellis Horwood, Chichester, England, 13-26, 1988.

Paulk, M.C., C.V. Weber, B. Curtis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, New York, 1995.

Penedo, M.H., An Active Web-based Virtual Room for Small Team Collaboration, *Software Process --Improvement and Practice*, 5,4,: 251-261, 2000.

Penedo, M.H. and E.D. Stuckle, PMDB--A Project Master Database for Software Engineering Environments, *Proc. 8th. Intern. Conf. Soft. Engr.*, IEEE Computer Society, Los Alamitos, CA,

150-157, 1985.

R. Radice, N.K. Roth, A.C. O'Hara and W.A. Ciarfella, A Programming Process Architecture. *IBM Systems Journal*, 24(2), 79-90, 1985.

Raffo, D. and W. Scacchi, Special Issue on Software Process Simulation and Modeling, *Software Process--Improvement and Practice*, 5(2-3), 87-209, 2000.

Raffo, D., W. Harrison, M.I. Kellner, R. Madachy, R. Martin, W. Scacchi, and P. Wernick, Special Issue on Software Process Simulation Modeling, *Journal of Systems and Software*, 46(2-3), 89-211, 1999.

Royce, W. W., Managing the Development of Large Software Systems, *Proc. 9th. Intern. Conf. Software Engineering*, IEEE Computer Society, 1987 ,328-338 Originally published in Proc. WESCON, 1970.

Royce, W., TRW's Ada Process Model for Incremental Development of Large Software Systems, *Proc. 12th. Intern. Conf. Software Engineering*, Nice, France, 2-11, IEEE Computer Society, 1990.

Sathi, A., M. S. Fox, and M. Greenberg, Representation of Activity Knowledge for Project Management, *IEEE Trans. Patt. Anal. and Mach. Intell.*, 7,5,531-552, 1985.

Scacchi, W., Managing Software Engineering Projects: A Social Analysis, *IEEE Trans. Software Engineering*, SE-10,1, 49-59, January, 1984.

Scacchi, W., Understanding Software Process Redesign using Modeling, Analysis and Simulation. *Software Process --Improvement and Practice* 5(2/3):183-195, 2000.

Scacchi, W. and P. Mi., Process Life Cycle Engineering: A Knowledge-Based Approach and Environment, *Intelligent Systems in Accounting, Finance, and Management*, 6(1):83-107, 1997.

Scacchi, W. and J. Noll, Process-Driven Intranets: Life Cycle Support for Process Reengineering, *IEEE Internet Computing*, 1(5):42-49, 1997.

Selby,R.W., V.R. Basili, and T. Baker, CLEANROOM Software Development: An Empirical Evaluation, *IEEE Trans. Soft. Engr.*, 3, 9, 1027-1037, 1987.

Somerville, I. *Software Engineering* (7th. Edition), Addison-Wesley, Menlo Park, CA, 1999.

Truex, D., R. Baskerville, and H. Klein, Growing Systems in an Emergent Organization, *Communications ACM*, 42(8), 117-123, 1999.

Winograd, T. and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishers, Lexington, MA, 1986.

Williams, L., Software Process Modeling: A Behavioral Approach, *Proc. 10th. Intern. Conf. Software Engineering*, IEEE Computer Society, 174-200, 1988.

Wirth, N., Program Development by Stepwise Refinement, *Communications of the ACM*, 14, 4, 221-227, 1971.

Wood, J. and D. Silver, *Joint Application Development*, Wiley and Sons, Inc. New York, 1995.

Wood, M., and I. Sommerville, A Knowledge-Based Software Components Catalogue, *Software Engineering Environments*, Ellis Horwood, P. Brereton (ed.), Chichester, England, 116-131, 1988.

Yu, E.S.K. and J. Mylopoulos, Understanding "why" in software process modelling, analysis, and design, *Proc. 16th. Intern. Conf. Software Engineering*, 159 -168, 1994.

Zave, P., The Operational Versus the Conventional Approach to Software Development, *Communications of the ACM*, 27, 104-118, 1984.



What is 'Software Quality Assurance'?

Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

What is 'Software Testing'?

Testing involves operation of a system or application under controlled conditions and evaluating the results (eg, 'if the user is in interface A of the application while using hardware B, and does C, then D should happen'). The controlled conditions should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should. It is oriented to 'detection'.

Organizations vary considerably in how they assign responsibility for QA and testing. Sometimes they're the combined responsibility of one group or individual. Also common are project teams that include a mix of testers and developers who work closely together, with overall QA processes monitored by project managers. It will depend on what best fits an organization's size and business structure.

Does every software project need testers?

While all projects will benefit from testing, some projects may not require independent test staff to succeed.

Which projects may not need independent test staff? The answer depends on the size and context of the project, the risks, the development methodology, the skill and experience of the developers, and other factors. For instance, if the project is a short-term, small, low risk project, with highly experienced programmers utilizing thorough unit testing or test-first development, then test engineers may not be required for the project to succeed.

In some cases an IT organization may be too small or new to have a testing staff even if the situation calls for it. In these circumstances it may be appropriate to instead use contractors or outsourcing, or adjust the project management and development approach (by switching to more senior developers and agile test-first development, for example). Inexperienced managers sometimes gamble on the success of a project by skipping thorough testing or having programmers do post-development functional testing of their own work, a decidedly high risk gamble.

For non-trivial-size projects or projects with non-trivial risks, a testing staff is usually necessary. As in any business, the use of personnel with specialized skills enhances an organization's ability to be successful in large, complex, or difficult tasks. It allows for both a) deeper and stronger skills and b) the contribution of differing perspectives. For



example, programmers typically have the perspective of 'what are the technical issues in making this functionality work?'. A test engineer typically has the perspective of 'what might go wrong with this functionality, and how can we ensure it meets expectations?'. Technical people who can be highly effective in approaching tasks from both of those perspectives are rare, which is why, sooner or later, organizations bring in test specialists.

Why does software have bugs?

- miscommunication or no communication - as to specifics of what an application should or shouldn't do (the application's requirements).
- software complexity - the complexity of current software applications can be difficult to comprehend for anyone without experience in modern-day software development. Multi-tiered applications, client-server and distributed applications, data communications, enormous relational databases, and sheer size of applications have all contributed to the exponential growth in software/system complexity.
- programming errors - programmers, like anyone else, can make mistakes.
- changing requirements (whether documented or undocumented) - the end-user may not understand the effects of changes, or may understand and request them anyway - redesign, rescheduling of engineers, effects on other projects, work already completed that may have to be redone or thrown out, hardware requirements that may be affected, etc. If there are many minor changes or any major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems, and the complexity of coordinating changes may result in errors. Enthusiasm of engineering staff may be affected. In some fast-changing business environments, continuously modified requirements may be a fact of life. In this case, management must understand the resulting risks, and QA and test engineers must adapt and plan for continuous extensive testing to keep the inevitable bugs from running out of control
- time pressures - scheduling of software projects is difficult at best, often requiring a lot of guesswork. When deadlines loom and the crunch comes, mistakes will be made.
- egos - people prefer to say things like:

```
'no problem'  
'piece of cake'  
'I can whip that out in a few hours'  
'it should be easy to update that old code'
```

instead of:

```
'that adds a lot of complexity and we could end up  
making a lot of mistakes'  
'we have no idea if we can do that; we'll wing it'  
'I can't estimate how long it will take, until I  
take a close look at it'  
'we can't figure out what that old spaghetti code  
did in the first place'
```



If there are too many unrealistic 'no problem's', the result is bugs.

- poorly documented code - it's tough to maintain and modify code that is badly written or poorly documented; the result is bugs. In many organizations management provides no incentive for programmers to document their code or write clear, understandable, maintainable code. In fact, it's usually the opposite: they get points mostly for quickly turning out code, and there's job security if nobody else can understand it ('if it was hard to write, it should be hard to read').
- software development tools - visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

How can new Software QA processes be introduced in an existing organization?

- A lot depends on the size of the organization and the risks involved. For large organizations with high-risk (in terms of lives or property) projects, serious management buy-in is required and a formalized QA process is necessary.
- Where the risk is lower, management and organizational buy-in and QA implementation may be a slower, step-at-a-time process. QA processes should be balanced with productivity so as to keep bureaucracy from getting out of hand.
- For small groups or projects, a more ad-hoc process may be appropriate, depending on the type of customers and projects. A lot will depend on team leads or managers, feedback to developers, and ensuring adequate communications among customers, managers, developers, and testers.
- The most value for effort will often be in (a) requirements management processes, with a goal of clear, complete, testable requirement specifications embodied in requirements or design documentation, or in 'agile'-type environments extensive continuous coordination with end-users, (b) design inspections and code inspections, and (c) post-mortems/retrospectives.
- Other possibilities include incremental self-managed team approaches such as 'Kaizen' methods of continuous process improvement, the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

What is verification? validation?

Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings. Validation typically involves actual testing and takes place after verifications are completed. The term 'IV & V' refers to Independent Verification and Validation.

What is a 'walkthrough'?

A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required.



What's an 'inspection'?

An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything. Attendees should prepare for this type of meeting by reading thru the document; most problems will be found during this preparation. The result of the inspection meeting should be a written report. Thorough preparation for inspections is difficult, painstaking work, but is one of the most cost effective methods of ensuring quality.

What kinds of testing should be considered?

- Black box testing - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- White box testing - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- unit testing - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.
- integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.
- functional testing - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- system testing - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- end-to-end testing - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.
- sanity testing or smoke testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes,

- bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.
- regression testing - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing tools can be especially useful for this type of testing.
 - acceptance testing - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
 - load testing - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
 - stress testing - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.
 - performance testing - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
 - usability testing - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
 - install/uninstall testing - testing of full, partial, or upgrade install/uninstall processes.
 - recovery testing - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
 - failover testing - typically used interchangeably with 'recovery testing'
 - security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.
 - compatibility testing - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
 - exploratory testing - often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.
 - ad-hoc testing - similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.
 - context-driven testing - 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.
 - user acceptance testing - determining if software is satisfactory to an end-user or customer.
 - comparison testing - comparing software weaknesses and strengths to competing products.

- alpha testing - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.
- beta testing - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.
- mutation testing - a method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources.

What are 5 common problems in the software development process?

- poor requirements - if requirements are unclear, incomplete, too general, and not testable, there will be problems.
- unrealistic schedule - if too much work is crammed in too little time, problems are inevitable.
- inadequate testing - no one will know whether or not the program is any good until the customer complains or systems crash.
- featuritis - requests to pile on new features after development is underway; extremely common.
- miscommunication - if developers don't know what's needed or customer's have erroneous expectations, problems are guaranteed.

What are 5 common solutions to software development problems?

- solid requirements - clear, complete, detailed, cohesive, attainable, testable requirements that are agreed to by all players. Use prototypes to help nail down requirements. In 'agile'-type environments, continuous close coordination with customers/end-users is necessary.
- realistic schedules - allow adequate time for planning, design, testing, bug fixing, re-testing, changes, and documentation; personnel should be able to complete the project without burning out.
- adequate testing - start testing early on, re-test after fixes or changes, plan for adequate time for testing and bug-fixing. 'Early' testing ideally includes unit testing by developers and built-in testing and diagnostic capabilities.
- stick to initial requirements as much as possible - be prepared to defend against excessive changes and additions once development has begun, and be prepared to explain consequences. If changes are necessary, they should be adequately reflected in related schedule changes. If possible, work closely with customers/end-users to manage expectations. This will provide them a higher comfort level with their requirements decisions and minimize excessive changes later on.
- communication - require walkthroughs and inspections when appropriate; make extensive use of group communication tools - groupware, wiki's, bug-tracking tools and change management tools, intranet capabilities, etc.; insure that

information/documentation is available and up-to-date - preferably electronic, not paper; promote teamwork and cooperation; use prototypes 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.

- organize code for readability.
- use whitespace generously - vertically and horizontally
- each line of code should contain 70 characters max.
- one code statement per line.
- coding style should be consistent throughout a program (eg, use of brackets, indentations, naming conventions, etc.)
- in adding comments, err on the side of too many rather than too few comments; a common rule of thumb is that there should be at least as many lines of comments (including header blocks) as lines of code.
- no matter how small, an application should include documentation of the overall program function and flow (even a few paragraphs is better than nothing); or if possible a separate flow chart and detailed program documentation.
- make extensive use of error handling procedures and status and error logging.
- for C++, to minimize complexity and increase maintainability, avoid too many levels of inheritance in class hierarchies (relative to the size and complexity of the application). Minimize use of multiple inheritance, and minimize use of operator overloading (note that the Java programming language eliminates multiple inheritance and operator overloading.)
- for C++, keep class methods small, less than 50 lines of code per method is preferable.
- for C++, make liberal use of exception handlers

What is 'good design'?

'Design' could refer to many things, but often refers to 'functional design' or 'internal design'. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works correctly when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements. For programs that have a user interface, it's often a good idea to assume that the end user will have little computer knowledge and may not read a user manual or even the on-line help; some common rules-of-thumb include:

- the program should act in a way that least surprises the user
- it should always be evident to the user what can be done next and how to exit
- the program shouldn't let the users do something stupid without warning them.

What is SEI? CMM? CMMI? ISO? IEEE? ANSI? Will it help?

- SEI = 'Software Engineering Institute' at Carnegie-Mellon University; initiated by the U.S. Defense Department to help improve software development processes.
- CMM = 'Capability Maturity Model', now called the CMMI ('Capability Maturity Model Integration'), developed by the SEI. It's a model of 5 levels of process 'maturity' that determine effectiveness in delivering quality software. It is geared to large organizations such as large U.S. Defense Department contractors.



However, many of the QA processes involved are appropriate to any organization, and if reasonably applied can be helpful. Organizations can receive CMMI ratings by undergoing assessments by qualified auditors.

Level 1 - characterized by chaos, periodic panics, and heroic efforts required by individuals to successfully complete projects. Few if any processes in place; successes may not be repeatable.

Level 2 - software project tracking, requirements management, realistic planning, and configuration management processes are in place; successful practices can be repeated.

Level 3 - standard software development and maintenance Processes are integrated throughout an organization; a Software Engineering Process Group is in place to oversee software processes, and training programs are used to ensure understanding and compliance.

Level 4 - metrics are used to track productivity, processes, and products. Project performance is predictable, and quality is consistently high.

Level 5 - the focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required.

Perspective on CMM ratings: During 1997–2001, 1018 organizations were assessed. Of those, 27% were rated at Level 1, 39% at 2, 23% at 3, 6% at 4, and 5% at 5. (For ratings during the period 1992–96, 62% were at Level 1, 23% at 2, 13% at 3, 2% at 4, and 0.4% at 5.) The median size of organizations was 100 software engineering/maintenance personnel; 32% of organizations were U.S. federal contractors or agencies. For those rated at Level 1, the most problematical key process area was in Software Quality Assurance.

- ISO = 'International Organisation for Standardization' - The ISO 9001:2000 standard (which replaces the previous standard of 1994) concerns quality systems that are assessed by outside auditors, and it applies to many kinds of production and manufacturing organizations, not just software. It covers documentation, design, development, production, testing, installation, servicing, and other processes. The full set of standards consists of: (a)Q9001-2000 - Quality Management Systems: Requirements; (b)Q9000-2000 - Quality Management Systems: Fundamentals and Vocabulary; (c)Q9004-2000 - Quality Management Systems: Guidelines for Performance Improvements. To be ISO 9001 certified, a third-party auditor assesses an organization, and certification is typically good for about 3 years, after which a complete reassessment is required. Note that ISO

certification does not necessarily indicate quality products - it indicates only that documented processes are followed.

- IEEE = 'Institute of Electrical and Electronics Engineers' - among other things, creates standards such as 'IEEE Standard for Software Test Documentation' (IEEE/ANSI Standard 829), 'IEEE Standard of Software Unit Testing' (IEEE/ANSI Standard 1008), 'IEEE Standard for Software Quality Assurance Plans' (IEEE/ANSI Standard 730), and others.
- ANSI = 'American National Standards Institute', the primary industrial standards body in the U.S.; publishes some software-related standards in conjunction with the IEEE and ASQ (American Society for Quality).
- Other software development/IT management process assessment methods besides CMMI and ISO 9000 include SPICE, Trillium, TickIT, Bootstrap, ITIL, MOF, and CobiT.

What is the 'software life cycle'?

The life cycle begins when an application is first conceived and ends when it is no longer in use. It includes aspects such as initial concept, requirements analysis, functional design, internal design, documentation planning, test planning, coding, document preparation, integration, testing, maintenance, updates, retesting, phase-out, and other aspects.



What makes a good Software Test engineer?

A good test engineer has a 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Tact and diplomacy are useful in maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people is useful. Previous software development experience can be helpful as it provides a deeper understanding of the software development process, gives the tester an appreciation for the developers' point of view, and reduce the learning curve in automated test tool programming. Judgement skills are needed to assess high-risk areas of an application on which to focus testing efforts when time is limited.

What makes a good Software QA engineer?

The same qualities a good tester has are useful for a QA engineer. Additionally, they must be able to understand the entire software development process and how it can fit into the business approach and goals of the organization. Communication skills and the ability to understand various sides of issues are important. In organizations in the early stages of implementing QA processes, patience and diplomacy are especially needed. An ability to find problems as well as to see 'what's missing' is important for inspections and reviews.

What makes a good QA or Test manager?

A good QA, test, or QA/Test(combined) manager should:

- be familiar with the software development process
- be able to maintain enthusiasm of their team and promote a positive atmosphere, despite what is a somewhat 'negative' process (e.g., looking for or preventing problems)
- be able to promote teamwork to increase productivity
- be able to promote cooperation between software, test, and QA engineers
- have the diplomatic skills needed to promote improvements in QA processes
- have the ability to withstand pressures and say 'no' to other managers when quality is insufficient or QA processes are not being adhered to
- have people judgement skills for hiring and keeping skilled personnel
- be able to communicate with technical and non-technical people, engineers, managers, and customers.
- be able to run meetings and keep them focused

What's the role of documentation in QA?



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 developers 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, approaches, 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/responsibilities
- 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

Jagan

- Outside test organizations to be utilized and their purpose, responsibilities, 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 its 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



- Bug reporting date
- Name of developer/group/organization the problem is assigned to
- Description of problem cause
- Description of fix
- Code section/file/module/class/method that was fixed
- Date of fix
- Application version that contains the fix
- Tester responsible for retest
- Retest date
- Retest results
- Regression testing requirements
- Tester responsible for regression tests
- Regression testing results

A reporting or tracking process should enable notification of appropriate personnel at various stages. For instance, testers need to know when retesting is needed, developers need to know when bugs are found and how to get the needed information, and reporting/summary capabilities are needed for managers.

What is 'configuration management'?

Configuration management covers the processes used to control, coordinate, and track: code, requirements, documentation, problems, change requests, designs, tools/compilers/libraries/patches, changes made to them, and who makes the changes.

What if the software is so buggy it can't really be tested at all?

The best bet in this situation is for the testers to go through the process of reporting whatever bugs or blocking-type problems initially show up, with the focus being on critical bugs. Since this type of problem can severely affect schedules, and indicates deeper problems in the software development process (such as insufficient unit testing or insufficient integration testing, poor design, improper build or release procedures, etc.) managers should be notified, and provided with some documentation as evidence of the problem.

How can it be known when to stop testing?

This can be difficult to determine. Many modern software applications are so complex, and run in such an interdependent environment, that complete testing can never be done. Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point



- Bug rate falls below a certain level
- Beta or alpha testing period ends

What if there isn't enough time for thorough testing?

Use risk analysis to determine where testing should be focused. Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. This requires judgement skills, common sense, and experience. (If warranted, formal methods are also available.) Considerations can include:

- Which functionality is most important to the project's intended purpose?
- Which functionality is most visible to the user?
- Which functionality has the largest safety impact?
- Which functionality has the largest financial impact on users?
- Which aspects of the application are most important to the customer?
- Which aspects of the application can be tested early in the development cycle?
- Which parts of the code are most complex, and thus most subject to errors?
- Which parts of the application were developed in rush or panic mode?
- Which aspects of similar/related previous projects caused problems?
- Which aspects of similar/related previous projects had large maintenance expenses?
- Which parts of the requirements and design are unclear or poorly thought out?
- What do the developers think are the highest-risk aspects of the application?
- What kinds of problems would cause the worst publicity?
- What kinds of problems would cause the most customer service complaints?
- What kinds of tests could easily cover multiple functionalities?
- Which tests will have the best high-risk-coverage to time-required ratio?

What if the project isn't big enough to justify extensive testing?

Consider the impact of project errors, not the size of the project. However, if extensive testing is still not justified, risk analysis is again needed and the same considerations as described previously in '[What if there isn't enough time for thorough testing?](#)' apply. The tester might then do ad hoc testing, or write up a limited test plan based on the risk analysis.

How does a client/server environment affect testing?

Client/server applications can be quite complex due to the multiple dependencies among clients, data communications, hardware, and servers, especially in multi-tier systems. Thus testing requirements can be extensive. When time is limited (as it usually is) the focus should be on integration and system testing. Additionally, load/stress/performance



testing may be useful in determining client/server application limitations and capabilities. There are commercial tools to assist with such testing.

How can World Wide Web sites be tested?

Web sites are essentially client/server applications - with web servers and 'browser' clients. Consideration should be given to the interactions between html pages, TCP/IP communications, Internet connections, firewalls, applications that run in web pages (such as applets, javascript, plug-in applications), and applications that run on the server side (such as cgi scripts, database interfaces, logging applications, dynamic page generators, asp, etc.). Additionally, there are a wide variety of servers and browsers, various versions of each, small but sometimes significant differences between them, variations in connection speeds, rapidly changing technologies, and multiple standards and protocols. The end result is that testing for web sites can become a major ongoing effort. Other considerations might include:

- What are the expected loads on the server (e.g., number of hits per unit time?), and what kind of performance is required under such loads (such as web server response time, database query response times). What kinds of tools will be needed for performance testing (such as web load testing tools, other tools already in house that can be adapted, web robot downloading tools, etc.)?
- Who is the target audience? What kind of browsers will they be using? What kind of connection speeds will they be using? Are they intra- organization (thus with likely high connection speeds and similar browsers) or Internet-wide (thus with a wide variety of connection speeds and browser types)?
- What kind of performance is expected on the client side (e.g., how fast should pages appear, how fast should animations, applets, etc. load and run)?
- Will down time for server and content maintenance/upgrades be allowed? how much?
- What kinds of security (firewalls, encryptions, passwords, etc.) will be required and what is it expected to do? How can it be tested?
- How reliable are the site's Internet connections required to be? And how does that affect backup system or redundant connection requirements and testing?
- What processes will be required to manage updates to the web site's content, and what are the requirements for maintaining, tracking, and controlling page content, graphics, links, etc.?
- Which HTML specification will be adhered to? How strictly? What variations will be allowed for targeted browsers?
- Will there be any standards or requirements for page appearance and/or graphics throughout a site or parts of a site??
- How will internal and external links be validated and updated? how often?
- Can testing be done on the production system, or will a separate test system be required? How are browser caching, variations in browser option settings, dial-up connection variabilities, and real-world internet 'traffic congestion' problems to be accounted for in testing?



- How extensive or customized are the server logging and reporting requirements; are they considered an integral part of the system and do they require testing?
- How are cgi programs, applets, javascripts, ActiveX components, etc. to be maintained, tracked, controlled, and tested?
- Pages should be 3-5 screens max unless content is tightly focused on a single topic. If larger, provide internal links within the page.
- The page layouts and design elements should be consistent throughout a site, so that it's clear to the user that they're still within a site.
- Pages should be as browser-independent as possible, or pages should be provided or generated based on the browser-type.
- All pages should have links external to the page; there should be no dead-end pages.
- The page owner, revision date, and a link to a contact person or organization should be included on each page.

How is testing affected by object-oriented designs?

Well-engineered object-oriented design can make it easier to trace from code to internal design to functional design to requirements. While there will be little affect on black box testing (where an understanding of the internal design of the application is unnecessary), white-box testing can be oriented to the application's objects. If the application was well-designed this can simplify test design.

What is Extreme Programming and what's it got to do with testing?

Extreme Programming (XP) is a software development approach for small teams on risk-prone projects with unstable requirements. It was created by Kent Beck who described the approach in his book 'Extreme Programming Explained'. Testing ('extreme testing') is a core aspect of Extreme Programming. Programmers are expected to write unit and functional test code first - before writing the application code. Test code is under source control along with the rest of the code. Customers are expected to be an integral part of the project team and to help develop scenarios for acceptance/black box testing. Acceptance tests are preferably automated, and are modified and rerun for each of the frequent development iterations. QA and test personnel are also required to be an integral part of the project team. Detailed requirements documentation is not used, and frequent re-scheduling, re-estimating, and re-prioritizing is expected. For more info on XP and other 'agile' software development approaches (Scrum, Crystal, etc.).

Software Definitions - Testing Levels / Phases - Testing Roles - Testing Techniques - Categories of Testing Tools

Software Definitions

Software Assurance: The planned and systematic set of activities that ensure that software life cycle processes and products conform to requirements, standards, and procedures [IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology]. For NASA this includes the disciplines of Software Quality (functions of Software Quality Engineering, Software Quality Assurance, Software Quality Control), Software Safety, Software Reliability, Software Verification and Validation, and IV&V.

Software Quality: The discipline of software quality is a planned and systematic set of activities to ensure quality is built into the software. It consists of software quality assurance, software quality control, and software quality engineering. As an attribute, software quality is (1) the degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations [IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology].

Software Quality Assurance: The function of software quality that assures that the standards, processes, and procedures are appropriate for the project and are correctly implemented.

Software Quality Control: The function of software quality that checks that the project follows its standards, processes, and procedures, and that the project produces the required internal and external (deliverable) products.

Software Quality Engineering: The function of software quality that assures that quality is built into the software by performing analyses, trade studies, and investigations on the requirements, design, code and verification processes and results to assure that reliability, maintainability, and other quality factors are met.

Software Reliability: The discipline of software assurance that 1) defines the requirements for software controlled system fault/failure detection, isolation, and recovery; 2) reviews the software development processes and products for software error prevention and/ or controlled change to reduced functionality states; and 3) defines the process for measuring and analyzing defects and defines/ derives the reliability and maintainability factors.

Software Safety: The discipline of software assurance that is a systematic approach to identifying, analyzing, tracking, mitigating and controlling software hazards and hazardous functions (data and commands) to ensure safe operation within a system.

Verification: Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled [ISO/IEC 12207, Software life cycle processes]. In other words, verification ensures that “you built it right”.

Validation: Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled [ISO/IEC 12207, Software life cycle processes.] In other words, validation ensures that “you built the right thing”.

Independent Verification and Validation (IV&V): Verification and validation performed by an organization that is technically, managerially, and financially independent. IV&V, as a part of Software Assurance, plays a role in the overall NASA software risk mitigation strategy applied throughout the life cycle, to improve the safety and quality of software.

Testing Levels / Phases

Testing levels or phases should be applied against the application under test when the previous phase of testing is deemed to be complete . or .complete enough.. Any defects detected during any level or phase of testing need to be recorded and acted on appropriately.

Design Review

"The objective of Design Reviews is to verify all documented design criteria before development begins." The design deliverable or deliverables to be reviewed should be complete within themselves. The environment of the review should be a professional examination of the deliverable with the focus being the deliverable not the author (or authors). The review must ensure each design deliverable for: completeness, correctness, and fit (both within the business model, and system architecture).

Design reviews should be conducted by: system matter experts, testers, developers, and system architects to ensure all aspects of the design are reviewed.

Unit Test

"The objective of unit test is to test every line of code in a component or module." The unit of code to be tested can be tested independent of all other units. The environment of the test should be isolated to the immediate development environment and have little, if any, impact on other units being developed at the same time. The test data can be fictitious and does not have to bear any relationship to .real world. business events. The test data need only consist of what is required to ensure that the component and component interfaces conform to the system architecture. The unit test must ensure each component: compiles, executes, interfaces, and passes control from the unit under test to the next component in the process according to the process model.

The developer in conjunction with a peer should conduct unit test to ensure the component is stable enough to be released into the product stream.

Function Test

"The objective of function test is to measure the quality of the functional (business) components of the system." Tests verify that the system behaves correctly from the user / business perspective and functions according to the requirements, models, storyboards, or any other design paradigm used to specify the application. The function test must determine if each component or business event: performs in accordance to the specifications, responds correctly to all conditions that may be presented by incoming events / data, moves data correctly from one business event to the next (including data stores), and that business events are initiated in the order required to meet the business objectives of the system.

Function test should be conducted by an independent testing organization to ensure the various components are stable and meet minimum quality criteria before proceeding to System test.

System Test

"The objective of system test is to measure the effectiveness and efficiency of the system in the "real-world" environment." System tests are based on business processes (workflows) and performance criteria rather than processing conditions. The system test must determine if the deployed system: satisfies the operational and technical performance criteria, satisfies the business requirements of the System Owner / Users / Business Analyst, integrates properly with operations (business processes, work procedures, user guides), and that the business objectives for building the system were attained.

There are many aspects to System testing the most common are:

- ☛ **Security Testing:** The tester designs test case scenarios that attempt to subvert or bypass security.
- ☛ **Stress Testing:** The tester attempts to stress or load an aspect of the system to the point of failure; the goal being to determine weak points in the system architecture.
- ☛ **Performance Testing:** The tester designs test case scenarios to determine if the system meets the stated performance criteria (i.e. A Login request shall be responded to in 1 second or less under a typical daily load of 1000 requests per minute.)
- ☛ **Install (Roll-out) Testing:** The tester designs test case scenarios to determine if the installation procedures lead to an invalid or incorrect installation.
- ☛ **Recovery Testing:** The tester designs test case scenarios to determine if the system meets the stated fail-over and recovery requirements.

System test should be conducted by an independent testing organization to ensure the system is stable and meets minimum quality criteria before proceeding to User Acceptance test.

User Acceptance Test

"The objective of User Acceptance test is for the user community to measure the effectiveness and efficiency of the system in the "real-world" environment.". User Acceptance test is based on User Acceptance criteria, which can include aspects of Function and System test. The User Acceptance test must determine if the deployed system: meets the end Users expectations, supports all operational requirements (both recorded and non-recorded), and fulfills the business objectives (both recorded and non-recorded) for the system.

User Acceptance test should be conducted by the end users of the system and monitored by an independent testing organization. The Users must ensure the system is stable and meets the minimum quality criteria before proceeding to system deployment (roll-out).

Testing Roles

As in any organization or organized endeavor there are Roles that must be fulfilled within any testing organization. The requirement for any given role depends on the size, complexity, goals, and maturity of the testing organization. These are roles, so it is quite possible that one person could fulfill many roles within the testing organization.

Test Lead or Test Manager

The Role of Test Lead / Manager is to effectively lead the testing team. To fulfill this role the Lead must understand the discipline of testing and how to effectively implement a testing process while fulfilling the traditional leadership roles of a manager. What does this mean? The manager must manage and implement or maintain an effective testing process.

Test Architect

The Role of the Test Architect is to formulate an integrated test architecture that supports the testing process and leverages the available testing infrastructure. To fulfill this role the Test Architect must have a clear understanding of the short-term and long-term goals of the organization, the resources (both hard and soft) available to the organization, and a clear vision on how to most effectively deploy these assets to form an integrated test architecture.

Test Designer or Tester

The Role of the Test Designer / Tester is to: design and document test cases, execute tests, record test results, document defects, and perform test coverage analysis. To fulfill this role the designer must be able to apply the most appropriate testing techniques to test

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

Overtime the IT industry 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.

Regression tests

Regression testing is not a testing technique or test phase; it is the reuse of existing tests to test previously implemented functionality--it is included here only for clarification.

Equivalence Partitioning

Equivalence testing leverages the concept of "classes" of input conditions. A "class" of input could be "City Name" where testing one or several city names could be deemed equivalent to testing all city names. In other word each instance of a class in a test covers a large set of other possible tests.

Boundary-value Analysis

Boundary-value analysis is really a variant on Equivalence Partitioning but in this case the upper and lower end of the class and often values outside the valid range of the class are used for input into the test cases. For example, if the Class is "Numeric Month of the Year" then the Boundary-values could be 0, 1, 12, and 13.

Error Guessing

Error Guessing involves making an itemized list of the errors expected to occur in a particular area of the system and then designing a set of test cases to check for these expected errors. Error Guessing is more testing art than testing science but can be very effective given a tester familiar with the history of the system.

Output Forcing

Output Forcing involves making a set of test cases designed to produce a particular output from the system. The focus here is on creating the desired output not on the input that initiated the system response.

Categories of Testing Tools

A number of different types of automated and manual testing tools are required to support an automated testing framework.

Test Design Tools. Tools that are used to plan software testing activities. These tools are used to create test artifacts that drive later testing activities.

Static Analysis Tools. Tools that analyze programs without machines executing them. Inspections and walkthroughs are examples of static testing tools.

Dynamic Analysis Tools. Tools that involve executing the software in order to test it.

GUI Test Drivers and Capture/Replay Tools. Tools that use macrorecording capabilities to automate testing of applications employing GUIs.

Load and Performance Tools. Tools that simulate different user load conditions for automated stress and volume testing.

Non-GUI Test Drivers and Test Managers. Tools that automate test execution of applications that do not allow tester interaction via a GUI.

Other Test Implementation Tools. Miscellaneous tools that assist test implementation. We include the MS Office Suite of tools here.

Test Evaluation Tools. Tools that are used to evaluate the quality of a testing effort.

Software Testing - Dictionary

Acceptance Test. Formal tests (often performed by a customer) to determine whether or not a system has satisfied predetermined acceptance criteria. These tests are often used to enable the customer (either internal or external) to determine whether or not to accept a system.

Accessibility testing. Testing that determines if software will be usable by people with disabilities.

Ad Hoc Testing. Testing carried out using no recognised test case design technique. [BCS]

Algorithm verification testing. A software development and test phase focused on the validation and tuning of key algorithms using an iterative experimentation process.[Scott Loveland, 2005]

Alpha Testing. Testing of a software product or system conducted at the developer's site by the customer.

Artistic testing. Also known as Exploratory testing.

Assertion Testing. (NBS) A dynamic analysis technique which inserts assertions about the relationship between program variables into the program code. The truth of the assertions is determined as the program executes.

Automated Testing. Software testing which is assisted with software technology that does not require operator (tester) input, analysis, or evaluation.

Audit.

- (1) An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria. (IEEE)
- (2) To conduct an independent review and examination of system records and activities in order to test the adequacy and effectiveness of data security and data integrity procedures, to ensure compliance with established policy and operational procedures, and to recommend any necessary changes. (ANSI)

ABEND Abnormal END. A mainframe term for a program crash. It is always associated with a failure code, known as an ABEND code.[Scott Loveland, 2005]

Background testing. Is the execution of normal functional testing while the SUT is exercised by a realistic work load. This work load is being processed "in the background" as far as the functional testing is concerned. [Load Testing Terminology by Scott Stirling]

Basis path testing. Identifying tests based on flow and paths of the program or system. [William E. Lewis, 2000]

Basis test set. A set of test cases derived from the code logic which ensure that 100% branch coverage is achieved. [BCS]

Bug: glitch, error, goof, slip, fault, blunder, boner, howler, oversight, botch, delusion, elision. [B. Beizer, 1990], defect, issue, problem

Beta Testing. Testing conducted at one or more customer sites by the end-user of a delivered software product or system.

Benchmarks Programs that provide performance comparison for software, hardware, and systems.

Benchmarking is specific type of performance test with the purpose of determining performance baselines for comparison. [Load Testing Terminology by Scott Stirling]

Big-bang testing. Integration testing where no incremental testing takes place prior to all the system's components being combined to form the system.[BCS]

Black box testing. A testing method where the application under test is viewed as a black box and the internal behavior of the program is completely ignored. Testing occurs based upon the external specifications. Also known as behavioral testing, since only the external behaviors of the program are evaluated and analyzed.

Bottom-up Testing. An approach to integration testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested. [BCS]

Boundary Value Analysis (BVA). BVA is different from equivalence partitioning in that it focuses on "corner cases" or values that are usually out of range as defined by the specification. This means that if function expects all values in range of negative 100 to positive 1000, test inputs would include negative 101 and positive 1001. BVA attempts to derive the value often used as a technique for stress, load or volume testing. This type of validation is usually performed after positive functional validation has completed (successfully) using requirements specifications and user documentation.

Branch Coverage Testing. - Verify each branch has true and false outcomes at least once. [William E. Lewis, 2000]

Breadth test. - A test suite that exercises the full scope of a system from a top-down perspective, but does not test any aspect in detail [Dorothy Graham, 1999]

BRS - Business Requirement Specification

Capability Maturity Model (CMM). - A description of the stages through which software organizations evolve as they define, implement, measure, control and improve their software processes. The model is a guide for selecting the process improvement strategies

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]

Compatibility Testing. The process of determining the ability of two or more systems to exchange information. In a situation where the developed software replaces an already working program, an investigation should be conducted to assess possible comparability problems between the new software and other programs or systems.

Composability testing -testing the ability of the interface to let users do more complex tasks by combining different sequences of simpler, easy-to-learn tasks. [Timothy Dyck, 'Easy' and other lies, eWEEK April 28, 2003]

Condition Coverage. A test coverage criteria requiring enough test cases such that each condition in a decision takes on all possible outcomes at least once, and each point of entry to a program or subroutine is invoked at least once. Contrast with branch coverage, decision coverage, multiple condition coverage, path coverage, statement coverage.[G.Myers]

Configuration. The functional and/or physical characteristics of hardware/software as set forth in technical documentation and achieved in a product. (MIL-STD-973)

Configuration control. An element of configuration management, consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification. (IEEE)

Conformance directed testing. Testing that seeks to establish conformance to requirements or specification. [R. V. Binder, 1999]

Cookbook scenario. A test scenario description that provides complete, step-by-step details about how the scenario should be performed. It leaves nothing to change. [Scott Loveland, 2005]

Coverage analysis. Determining and assessing measures associated with the invocation of program structural elements to determine the adequacy of a test run. Coverage analysis is useful when attempting to execute each statement, branch, path, or iterative structure in a program. Tools that capture this data and provide reports summarizing relevant information have this feature. (NIST)

CRUD Testing. Build CRUD matrix and test all object creation, reads, updates, and deletion. [William E. Lewis, 2000]

Data-Driven testing. An automation approach in which the navigation and functionality of the test script is directed through external data; this approach separates test and control data from the test script. [Daniel J. Mosley, 2002]

Data flow testing. Testing in which test cases are designed based on variable usage within the code.[BCS]

Database testing. Check the integrity of database field values. [William E. Lewis, 2000]

Defect. The difference between the functional specification (including user documentation) and actual program text (source code and data). Often reported as problem and stored in defect-tracking and problem-management system

Defect. Also called a fault or a bug, a defect is an incorrect part of code that is caused by an error. An error of commission causes a defect of wrong or extra code. An error of omission results in a defect of missing code. A defect may cause one or more failures.[Robert M. Poston, 1996.]

Defect. A flaw in the software with potential to cause a failure.. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Age. A measurement that describes the period of time from the introduction of a defect until its discovery. . [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Density. A metric that compares the number of defects to a measure of size (e.g., defects per KLOC). Often used as a measure of defect quality. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Discovery Rate. A metric describing the number of defects discovered over a specified period of time, usually displayed in graphical form. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Removal Efficiency (DRE). A measure of the number of defects discovered in an activity versus the number that could have been found. Often used as a measure of test effectiveness. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Seeding. The process of intentionally adding known defects to those already in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of defects still remaining. Also called Error Seeding. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Defect Masked. An existing defect that hasn't yet caused a failure because another defect has prevented that part of the code from being executed. [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Depth test. A test case, that exercises some part of a system to a significant level of detail. [Dorothy Graham, 1999]

Decision Coverage. A test coverage criteria requiring enough test cases such that each decision has a true and false result at least once, and that each statement is executed at least once. Syn: branch coverage. Contrast with condition coverage, multiple condition coverage, path coverage, statement coverage.[G.Myers]

Design-based testing. Designing tests based on objectives derived from the architectural or detail design of the software (e.g., tests that execute specific invocation paths or probe the worst case behaviour of algorithms). [BCS]

Dirty testing Negative testing. [Beizer]

Dynamic testing. Testing, based on specific test cases, by execution of the test object or running programs [Tim Koomen, 1999]

End-to-End testing. Similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Equivalence Partitioning: An approach where classes of inputs are categorized for product or function validation. This usually does not include combinations of input, but rather a single state value based by class. For example, with a given function there may be several classes of input that may be used for positive testing. If function expects an integer and receives an integer as input, this would be considered as positive test assertion. On the other hand, if a character or any other input class other than integer is provided, this would be considered a negative test assertion or condition.

Error: An error is a mistake of commission or omission that a person makes. An error causes a defect. In software development one error may cause one or more defects in requirements, designs, programs, or tests.[Robert M. Poston, 1996.]

Errors: The amount by which a result is incorrect. Mistakes are usually a result of a human action. Human mistakes (errors) often result in faults contained in the source code, specification, documentation, or other product deliverable. Once a fault is encountered, the end result will be a program failure. The failure usually has some margin of error, either high, medium, or low.

Error Guessing: Another common approach to black-box validation. Black-box testing is when everything else other than the source code may be used for testing. This is the most common approach to testing. Error guessing is when random inputs or conditions are used for testing. Random in this case includes a value either produced by a computerized random number generator, or an ad hoc value or test conditions provided by engineer.

Error guessing. A test case design technique where the experience of the tester is used to postulate what faults exist, and to design tests specially to expose them [from BS7925-1]

Error seeding. The purposeful introduction of faults into a program to test effectiveness of a test suite or other quality assurance program. [R. V. Binder, 1999]

Exception Testing. Identify error messages and exception handling processes an conditions that trigger them. [William E. Lewis, 2000]

Exhaustive Testing.(NBS) Executing the program with all possible combinations of values for program variables. Feasible only for small, simple programs.

Exploratory Testing: An interactive process of concurrent product exploration, test design, and test execution. The heart of exploratory testing can be stated simply: The outcome of this test influences the design of the next test. [James Bach]

Failure: A failure is a deviation from expectations exhibited by software and observed as a set of symptoms by a tester or user. A failure is caused by one or more defects. The Causal Trail. A person makes an error that causes a defect that causes a failure.[Robert M. Poston, 1996]

Fix testing. Rerunning of a test that previously found the bug in order to see if a supplied fix works. [Scott Loveland, 2005]

Follow-up testing, we vary a test that yielded a less-than-spectacular failure. We vary the operation, data, or environment, asking whether the underlying fault in the code can yield a more serious failure or a failure under a broader range of circumstances.[Measuring the Effectiveness of Software Testers,Cem Kaner, STAR East 2003]

Formal Testing. (IEEE) Testing conducted in accordance with test plans and procedures that have been reviewed and approved by a customer, user, or designated level of management. Antonym: informal testing.

Framework scenario. A test scenario definition that provides only enough high-level information to remind the tester of everything that needs to be covered for that scenario. The description captures the activity's essence, but trusts the tester to work through the specific steps required.[Scott Loveland, 2005]

Free Form Testing. Ad hoc or brainstorming using intuition to define test cases. [William E. Lewis, 2000]

Functional Decomposition Approach. An automation method in which the test cases are reduced to fundamental tasks, navigation, functional tests, data verification, and return navigation; also known as Framework Driven Approach. [Daniel J. Mosley, 2002]

Functional testing Application of test data derived from the specified functional requirements without regard to the final program structure. Also known as black-box testing.

Function verification test (FVT). Testing of a complete, yet containable functional area or component within the overall software package. Normally occurs immediately after Unit test. Also known as Integration test. [Scott Loveland, 2005]

Gray box testing. Tests involving inputs and outputs, but test design is educated by information about the code or the program operation of a kind that would normally be out of scope of view of the tester.[Cem Kaner]

Gray box testing. Test designed based on the knowledge of algorithm, internal states, architectures, or other high -level descriptions of the program behavior. [Doug Hoffman]

Gray box testing. Examines the activity of back-end components during test case execution. Two types of problems that can be encountered during gray-box testing are:

- A component encounters a failure of some kind, causing the operation to be aborted. The user interface will typically indicate that an error has occurred.
- The test executes in full, but the content of the results is incorrect. Somewhere in the system, a component processed data incorrectly, causing the error in the results. [Elfriede Dustin. "Quality Web Systems: Performance, Security & Usability."]

Grooved Tests. Tests that simply repeat the same activity against a target product from cycle to cycle. [Scott Loveland, 2005]

High-level tests. These tests involve testing whole, complete products [Kit, 1995]

Incremental integration testing. Incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.

Inspection. A formal evaluation technique in which software requirements, design, or code are examined in detail by person or group other than the author to detect faults, violations of development standards, and other problems [IEEE94]. A quality improvement process for written material that consists of two dominant components: product (document) improvement and process improvement (document production and inspection).

Integration. The process of combining software components or hardware components or both into overall system.

Integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.

Integration Testing. Testing conducted after unit and feature testing. The intent is to expose faults in the interactions between software modules and functions. Either top-down

or bottom-up approaches can be used. A bottom-up method is preferred, since it leads to earlier unit testing (step-level integration). This method is contrary to the big-bang approach where all source modules are combined and tested in one step. The big-bang approach to integration should be discouraged.

Interface Tests. Programs that probe test facilities for external interfaces and function calls. Simulation is often used to test external interfaces that currently may not be available for testing or are difficult to control. For example, hardware resources such as hard disks and memory may be difficult to control. Therefore, simulation can provide the characteristics or behaviors for specific function.

Internationalization testing (I18N) - testing related to handling foreign text and data within the program. This would include sorting, importing and exporting test and data, correct handling of currency and date and time formats, string parsing, upper and lower case handling and so forth. [Clinton De Young, 2003].

Interoperability Testing which measures the ability of your software to communicate across the network on multiple machines from multiple vendors each of whom may have interpreted a design specification critical to your success differently.

Inter-operability Testing. True inter-operability testing concerns testing for unforeseen interactions with other packages with which your software has no direct connection. In some quarters, inter-operability testing labor equals all other testing combined. This is the kind of testing that I say shouldn't be done because it can't be done.[from Quality Is Not The Goal. By Boris Beizer, Ph. D.]

Inspection. A formal evaluation technique in which software requirements, design, or code are examined in detail by person or group other than the author to detect faults, violations of development standards, and other problems [IEEE94].

Install/uninstall testing. Testing of full, partial, or upgrade install/uninstall processes.

Latent bug A bug that has been dormant (unobserved) in two or more releases. [R. V. Binder, 1999]

Lateral testing. A test design technique based on lateral thinking principals, to identify faults. [Dorothy Graham, 1999]

Limits testing. See Boundary Condition testing.

Load testing. Testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.

Load & Stress test. A test is design to determine how heavy a load the application can handle.

Load-stability test. Test design to determine whether a Web application will remain serviceable over extended time span.

Load Isolation test. The workload for this type of test is designed to contain only the subset of test cases that caused the problem in previous testing.

Longevity testing. See Reliability testing.

Long-haul Testing. See Reliability testing.

Master Test Planning. An activity undertaken to orchestrate the testing effort across levels and organizations.[Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Monkey Testing. (smart monkey testing) Input are generated from probability distributions that reflect actual expected usage statistics -- e.g., from user profiles. There are different levels of IQ in smart monkey testing. In the simplest, each input is considered independent of the other inputs. That is, a given test requires an input vector with five components. In low IQ testing, these would be generated independently. In high IQ monkey testing, the correlation (e.g., the covariance) between these input distribution is taken into account. In all branches of smart monkey testing, the input is considered as a single event.[Visual Test 6 Bible by Thomas R. Arnold, 1998]

Monkey Testing. (brilliant monkey testing) The inputs are created from a stochastic regular expression or stochastic finite-state machine model of user behavior. That is, not only are the values determined by probability distributions, but the sequence of values and the sequence of states in which the input provider goes is driven by specified probabilities.[Visual Test 6 Bible by Thomas R. Arnold, 1998]

Monkey Testing. (dumb-monkey testing)Inputs are generated from a uniform probability distribution without regard to the actual usage statistics.[Visual Test 6 Bible by Thomas R. Arnold, 1998]

Maximum Simultaneous Connection testing. This is a test performed to determine the number of connections which the firewall or Web server is capable of handling.

Migration Testing. Testing to see if the customer will be able to transition smoothly from a prior version of the software to a new one. [Scott Loveland, 2005]

Mutation testing. A testing strategy where small variations to a program are inserted (a mutant), followed by execution of an existing test suite. If the test suite detects the mutant, the mutant is 'retired.' If undetected, the test suite must be revised. [R. V. Binder, 1999]

Multiple Condition Coverage. A test coverage criteria which requires enough test cases such that all possible combinations of condition outcomes in each decision, and all points

of entry, are invoked at least once.[G.Myers] Contrast with branch coverage, condition coverage, decision coverage, path coverage, statement coverage.

Negative test. A test whose primary purpose is falsification; that is tests designed to break the software[B.Beizer1995]

Noncritical code analysis. Examines software elements that are not designated safety-critical and ensures that these elements do not cause a hazard. (IEEE)

Orthogonal array testing: Technique can be used to reduce the number of combination and provide maximum coverage with a minimum number of TC.Pay attention to the fact that it is an old and proven technique. The OAT was introduced for the first time by Plackett and Burman in 1946 and was implemented by G. Taguchi, 1987

Orthogonal array testing: Mathematical technique to determine which variations of parameters need to be tested. [William E. Lewis, 2000]

Oracle. Test Oracle: a mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test [from BS7925-1]

Parallel Testing. Testing a new or an alternate data processing system with the same source data that is used in another system. The other system is considered as the standard of comparison. Syn: parallel run.[ISO]

Penetration testing. The process of attacking a host from outside to ascertain remote security vulnerabilities.

Performance Testing. Testing conducted to evaluate the compliance of a system or component with specific performance requirements [BS7925-1]

Performance testing can be undertaken to: 1) show that the system meets specified performance objectives, 2) tune the system, 3) determine the factors in hardware or software that limit the system's performance, and 4) project the system's future load-handling capacity in order to schedule its replacements" [Software System Testing and Quality Assurance. Beizer, 1984, p. 256]

Postmortem. Self-analysis of interim or fully completed testing activities with the goal of creating improvements to be used in future.[Scott Loveland, 2005]

Preventive Testing Building test cases based upon the requirements specification prior to the creation of the code, with the express purpose of validating the requirements [Systematic Software Testing by Rick D. Craig and Stefan P. Jaskiel 2002]

Prior Defect History Testing. Test cases are created or rerun for every defect found in prior tests of the system. [William E. Lewis, 2000]

Qualification Testing. (IEEE) Formal testing, usually conducted by the developer for the consumer, to demonstrate that the software meets its specified requirements. See: acceptance testing.

Quality. The degree to which a program possesses a desired combination of attributes that enable it to perform its specified end use.

Quality Assurance (QA) Consists of planning, coordinating and other strategic activities associated with measuring product quality against external requirements and specifications (process-related activities).

Quality Control (QC) Consists of monitoring, controlling and other tactical activities associated with the measurement of product quality goals.

Our definition of Quality: Achieving the target (not conformance to requirements as used by many authors) & minimizing the variability of the system under test

Race condition defect. Many concurrent defects result from data-race conditions. A data-race condition may be defined as two accesses to a shared variable, at least one of which is a write, with no mechanism used by either to prevent simultaneous access. However, not all race conditions are defects.

Recovery testing Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.

Regression Testing. Testing conducted for the purpose of evaluating whether or not a change to the system (all CM items) has introduced a new failure. Regression testing is often accomplished through the construction, execution and analysis of product and system tests.

Regression Testing. - testing that is performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program [Glenford J.Myers, 1979]

Reengineering. The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering (analyzing a system and producing a representation at a higher level of abstraction, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), recommendation (analyzing a system and producing user and support documentation), forward engineering (using software products derived from an existing system, together with new requirements, to produce a new system), and translation (transforming source code from one language to another or from one version of a language to another).

Reference testing. A way of deriving expected outcomes by manually validating a set of actual outcomes. A less rigorous alternative to predicting expected outcomes in advance of test execution. [Dorothy Graham, 1999]

Reliability testing. Verify the probability of failure free operation of a computer program in a specified environment for a specified time.

Reliability of an object is defined as the probability that it will not fail under specified conditions, over a period of time. The specified conditions are usually taken to be fixed, while the time is taken as an independent variable. Thus reliability is often written $R(t)$ as a function of time t , the probability that the object will not fail within time t .

Any computer user would probably agree that most software is flawed, and the evidence for this is that it does fail. All software flaws are designed in -- the software does not break, rather it was always broken. But unless conditions are right to excite the flaw, it will go unnoticed -- the software will appear to work properly. [Professor Dick Hamlet. Ph.D.]

Range Testing. For each input identifies the range over which the system behavior should be the same. [William E. Lewis, 2000]

Risk management. An organized process to identify what can go wrong, to quantify and access associated risks, and to implement/control the appropriate approach for preventing or handling each risk identified.

Robust test. A test, that compares a small amount of information, so that unexpected side effects are less likely to affect whether the test passed or fails. [Dorothy Graham, 1999]

Sanity Testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is often crashing systems, bogging down systems to a crawl, or destroying databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.

Scalability testing is a subtype of performance test where performance requirements for response time, throughput, and/or utilization are tested as load on the SUT is increased over time. [Load Testing Terminology by Scott Stirling]

Sensitive test. A test, that compares a large amount of information, so that it is more likely to detect unexpected differences between the actual and expected outcomes of the test. [Dorothy Graham, 1999]

Service test. Test software fixes, both individually and bundled together, for software that is already in use by customers. [Scott Loveland, 2005]

Skim Testing A testing technique used to determine the fitness of a new build or release of an AUT to undergo further, more thorough testing. In essence, a "pretest" activity that could form one of the acceptance criteria for receiving the AUT for testing [Testing IT: An Off-the-Shelf Software Testing Process by John Watkins]

Smoke test describes an initial set of tests that determine if a new version of application performs well enough for further testing.[Louise Tamres, 2002]

Sniff test. A quick check to see if any major abnormalities are evident in the software.[Scott Loveland, 2005]

Specification-based test. A test, whose inputs are derived from a specification.

Spike testing. to test performance or recovery behavior when the system under test (SUT) is stressed with a sudden and sharp increase in load should be considered a type of load test.[Load Testing Terminology by Scott Stirling]

SRS. Business Requirement Specification.

STEP (Systematic Test and Evaluation Process) Software Quality Engineering's copyrighted testing methodology.

State-based testing Testing with test cases developed by modeling the system under test as a state machine [R. V. Binder, 1999]

State Transition Testing. Technique in which the states of a system are first identified and then test cases are written to test the triggers to cause a transition from one condition to another state. [William E. Lewis, 2000]

Static testing. Source code analysis. Analysis of source code to expose potential defects.

Statistical testing. A test case design technique in which a model is used of the statistical distribution of the input to construct representative test cases. [BCS]

Stealth bug. A bug that removes information useful for its diagnosis and correction. [R. V. Binder, 1999]

Storage test. Study how memory and space is used by the program, either in resident memory or on disk. If there are limits of these amounts, storage tests attempt to prove that the program will exceed them. [Cem Kaner, 1999, p55]

Streamable Test cases. Test cases which are able to run together as part of a large group. [Scott Loveland, 2005]

Stress / Load / Volume test. Tests that provide a high degree of activity, either using boundary conditions as inputs or multiple copies of a program executing in parallel as examples.

Structural Testing. (1)(IEEE) Testing that takes into account the internal mechanism [structure] of a system or component. Types include branch testing, path testing, statement testing. (2) Testing to insure each program statement is made to execute during testing and

that each program statement performs its intended function. Contrast with functional testing. Syn: white-box testing, glass-box testing, logic driven testing.

System testing Black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.

System verification test. (SVT). Testing of an entire software package for the first time, with all components working together to deliver the project's intended purpose on supported hardware platforms. [Scott Loveland, 2005]

Table testing. Test access, security, and data integrity of table entries. [William E. Lewis, 2000]

Test Bed. An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test [IEEE 610].

Test Case. A set of test inputs, executions, and expected results developed for a particular objective.

Test conditions. The set of circumstances that a test invokes. [Daniel J. Mosley, 2002]

Test Coverage The degree to which a given test or set of tests addresses all specified test cases for a given system or component.

Test Criteria. Decision rules used to determine whether software item or software feature passes or fails a test.

Test data. The actual (set of) values used in the test or that are necessary to execute the test. [Daniel J. Mosley, 2002]

Test Documentation. (IEEE) Documentation describing plans for, or results of, the testing of a system or component, Types include test case specification, test incident report, test log, test plan, test procedure, test report.

Test Driver A software module or application used to invoke a test item and, often, provide test inputs (data), control and monitor execution. A test driver automates the execution of test procedures.

Test Harness A system of test drivers and other tools to support test execution (e.g., stubs, executable test cases, and test drivers). See: test driver.

Test Item. A software item which is the object of testing.[IEEE]

Test Log A chronological record of all relevant details about the execution of a test.[IEEE]

Test Plan. A high-level document that defines a testing project so that it can be properly measured and controlled. It defines the test strategy and organized elements of the test life cycle, including resource requirements, project schedule, and test requirements

Test Procedure. A document, providing detailed instructions for the [manual] execution of one or more test cases. [BS7925-1] Often called - a manual test script.

Test Rig A flexible combination of hardware, software, data, and interconnectivity that can be configured by the Test Team to simulate a variety of different Live Environments on which an AUT can be delivered.[Testing IT: An Off-the-Shelf Software Testing Process by John Watkins]

Test strategy. Describes the general approach and objectives of the test activities. [Daniel J. Mosley, 2002]

Test Status. The assessment of the result of running tests on software.

Test Stub A dummy software component or object used (during development and testing) to simulate the behaviour of a real component. The stub typically provides test output.

Test Suites A test suite consists of multiple test cases (procedures and data) that are combined and often managed by a test harness.

Test Tree. A physical implementation of Test Suite. [Dorothy Graham, 1999]

Testability. Attributes of software that bear on the effort needed for validating the modified software [ISO 8402]

Testability Hooks. Those functions, integrated in the software that can be invoked through primarily undocumented interfaces to drive specific processing which would otherwise be difficult to exercise. [Scott Loveland, 2005]

Testing. The execution of tests with the intent of providing that the system and application under test does or does not perform according to the requirements specification.

(TPI) Test Process Improvement. A method for baselining testing processes and identifying process improvement opportunities, using a static model developed by Martin Pol and Tim Koomen.

Thread Testing. A testing technique used to test the business functionality or business logic of the AUT in an end-to-end manner, in much the same way a User or an operator might interact with the system during its normal use.[Testing IT: An Off-the-Shelf Software Testing Process by John Watkins]

Timing and Serialization Problems. A class of software defect, usually in multithreaded code, in which two or more tasks attempt to alter a shared software resource without properly coordinating their actions. Also known as Race Conditions.[Scott Loveland, 2005]

Translation testing. See internationalization testing.

Thrasher. A type of program used to test for data integrity errors on mainframe system. The name is derived from the first such program, which deliberately generated memory thrashing (the overuse of large amount of memory, leading to heavy paging or swapping) while monitoring for corruption. [Scott Loveland, 2005]

Unit Testing. Testing performed to isolate and expose faults and failures as soon as the source code is available, regardless of the external interfaces that may be required. Oftentimes, the detailed design and requirements documents are used as a basis to compare how and what the unit is able to perform. White and black-box testing methods are combined during unit testing.

Usability testing. Testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer.

Validation. The comparison between the actual characteristics of something (e.g. a product of a software project and the expected characteristics). Validation is checking that you have built the right system.

Verification The comparison between the actual characteristics of something (e.g. a product of a software project) and the specified characteristics. Verification is checking that we have built the system right.

Volume testing. Testing where the system is subjected to large volumes of data.[BS7925-1]

Walkthrough In the most usual form of term, a walkthrough is step by step simulation of the execution of a procedure, as when walking through code line by line, with an imagined set of inputs. The term has been extended to the review of material that is not procedural, such as data descriptions, reference manuals, specifications, etc.

White Box Testing (glass-box). Testing is done under a structural testing strategy and require complete access to the object's structureÂ;that is, the source code.[B. Beizer, 1995 p8],

SOFTWARE TESTING

(ESTIMATION, FRAMEWORK, PROCESS & PLAN)

TESTING ESTIMATION PROCESS

One of the most difficult and critical activities in IT is the estimation process. I believe that it occurs because when we say that one project will be accomplished in such time by such cost, it must happen. If it does not happen, several things may follow: from peers' comments and senior management's warnings to being fired depending on the reasons and seriousness of the failure.

Before even thinking of moving to Systems test at my organization, I always heard from the development group members that the estimations made by the Systems test group were too long and expensive. Then, when I arrived at my new seat, I tried to understand the testing estimation process.

The testing estimation process in place was quite simple. The inputs for the process, provided by the development team, were: the size of the development team and the number of working days needed for building a solution before starting systems tests.

The testing estimation process said that the number of testing engineers would be half of the number of development engineers and one third of the number of development working days.

A spreadsheet was created in order to find out the estimation and calculate the duration of tests and testing costs. They are based on the following formulas:

Article I. Testing working days = (Development working days) / 3.

Article II.

Article III. Testing engineers = (Development engineers) / 2.

Article IV.

Testing costs = Testing working days * Testing engineers * person daily costs.

As the process was only playing with numbers, it was not necessary to register anywhere how the estimation was obtained.

To exemplify how the process worked, if one development team said that to deliver a solution for systems testing it would need 4 engineers and 66 working days then, the systems test would need 2 engineers (half) and 21 working days (one third). So, the solution would be ready for delivery to the customer after 87 (66+21) working days.

Just to be clear, in testing time, it was not included the time for developing the testcases and preparing the testing environment. Normally, it would need an extra 10 days for the testing team.

The new testing estimation process

Besides being simple, that process worked fine for different projects and years. But, I was not happy with this approach and my officemates from the development group were not, either. Metrics, project analogies, expertise, requirements, nothing were being used to support the estimation process.

I mentioned my thoughts to the testing group. We could not stand the estimation process for very long. I, myself, was not convinced to support it any more. Then, some rules were implemented in order to establish a new process.

Those rules are being shared below. I know that they are not complete and it was not my intention for estimating but, from now, I have strong arguments to discuss my estimation when someone doubts my numbers.

The Rules

1st Rule: Estimation shall be always based on the software requirements

All estimation should be based on what would be tested, i.e., the software requirements.

Normally, the software requirements were only established by the development team without any or just a little participation from the testing team. After the specification have been established and the project costs and duration have been estimated, the development team asks how long would take for testing the solution. The answer should be said almost right away.

Then, the software requirements shall be read and understood by the testing team, too. Without the testing participation, no serious estimation can be considered.

2nd Rule: Estimation shall be based on expert judgment

Before estimating, the testing team classifies the requirements in the following categories:

- ◆ Critical: The development team has little knowledge in how to implement it;
- ◆ High: The development team has good knowledge in how to implement it but it is not an easy task;
- ◆ Normal: The development team has good knowledge in how to implement.

The experts in each requirement should say how long it would take for testing them. The categories would help the experts in estimating the effort for testing the requirements.

3rd Rule: Estimation shall be based on previous projects

All estimation should be based on previous projects. If a new project has similar requirements from a previous one, the estimation is based on that project.

4th Rule: Estimation shall be based on metrics

My organization has created an OPD, Organization Process Database, where the project metrics are recorded. We have recorded metrics from three years ago obtained from dozens of projects.

The number of requirements is the basic information for estimating a testing project. From it, my organization has metrics that guide us to estimate a testing project. The table below shows the metrics used to estimate a testing project. The team size is 01 testing engineer.

Metric	Value
1 Number of testcases created for each requirement	4,53
2 Number of testcases developed by Working day	14,47
3 Number of testcases executed by Working day	10,20
4 Number of ARs for testcase	0,77
5 Number of ARs verified by Working day	24,64

For instance, if we have a project with 70 functional requirements and a testing team size of 2 engineers, we reach the following estimates:

Metric	Value
Number of testcases – based on metric 1	317,10
Preparation phase – based on metric 2	11 working days
Execution phase – based on metric 3	16 working days
Number of ARs – based on metric 4	244 ARs

Regression phase – based on metric 5	6 working days
--------------------------------------	----------------

The testing duration is estimated in 22 (16+6) working days. Plus, 11 working days for preparing it.

5th Rule: Estimation shall never forget the past

I have not sent away the past. The testing team continues using the old process and the spreadsheet. After the estimation is done following the new rules, the testing team estimates again using the old process in order to compare both results.

Normally, the results from the new estimate process are cheaper and faster than the old one in about 20 to 25%. If the testing team gets a different percentage, the testing team returns to the process in order to understand if something was missed.

6th Rule: Estimation shall be recorded

All decisions should be recorded. It is very important because if requirements change for any reason, the records would help the testing team to estimate again. The testing team would not need to return for all steps and take the same decisions again. Sometimes, it is an opportunity to adjust the estimation made earlier.

7th Rule: Estimation shall be supported by tools

A new spreadsheet has been created containing metrics that help to reach the estimation quickly. The spreadsheet calculates automatically the costs and duration for each testing phase.

There is also a letter template that contains some sections such as: cost table, risks, and free notes to be filled out. This letter is sent to the customer. It also shows the different options for testing that can help the customer decides which kind of test he needs.

8th Rule: Estimation shall always be verified

Finally, All estimation should be verified

UNIT TESTING TEST CASE

Introduction

The design of tests is subject to the same basic engineering principles as the design of software. Good design consists of a number of stages which progressively elaborate the design. Good test design consists of a number of stages which progressively elaborate the design of tests:

- Test strategy;
- Test planning;
- Test specification;
- Test procedure.

These four stages of test design apply to all levels of testing, from unit testing through to system testing. This paper concentrates on the specification of unit tests; i.e. the design of individual unit test cases within unit test specifications. A more detailed description of the four stages of test design can be found in the IPL paper "An Introduction to Software Testing".

The design of tests has to be driven by the specification of the software. For unit testing, tests are designed to verify that an individual unit implements all design decisions made in the unit's design specification. A thorough unit test specification should include positive testing, that the unit does what it is supposed to do, and also negative testing, that the unit does not do anything that it is not supposed to do.

Producing a test specification, including the design of test cases, is the level of test design which has the highest degree of creative input. Furthermore, unit test specifications will usually be produced by a large number of staff with a wide range of experience, not just a few experts.

2. Developing Unit Test Specifications

Once a unit has been designed, the next development step is to design the unit tests. An important point here is that it is more rigorous to design the tests before the code is written. If the code was written first, it would be too tempting to test the software against what it is observed to do (which is not really testing at all), rather than against what it is specified to do.

A unit test specification comprises a sequence of unit test cases. Each unit test case should include four essential elements:

- A statement of the initial state of the unit, the starting point of the test case (this is only applicable where a unit maintains state between calls);
- The inputs to the unit, including the value of any external data read by the unit;
- What the test case actually tests, in terms of the functionality of the unit and the analysis used in the design of the test case (for example, which decisions within the unit are tested);

The expected outcome of the test case (the expected outcome of a test case should always be defined in the test specification, prior to test execution).

2.1. Step 1 - Make it Run

The purpose of the first test case in any unit test specification should be to execute the unit under test in the simplest way possible. When the tests are actually executed, knowing that at least the first unit test will execute is a good confidence boost. If it will not execute, then it is preferable to have something as simple as possible as a starting point for debugging.

Suitable techniques:

- Specification derived tests
- Equivalence partitioning

2.2. Step 2 - Positive Testing

Test cases should be designed to show that the unit under test does what it is supposed to do. The test designer should walk through the relevant specifications; each test case should test one or more statements of specification. Where more than one specification is involved, it is best to make the sequence of test cases correspond to the sequence of statements in the primary specification for the unit.

Suitable techniques:

- Specification derived tests
- Equivalence partitioning
- State-transition testing

2.3. Step 3 - Negative Testing

Existing test cases should be enhanced and further test cases should be designed to show that the software does not do anything that it is not specified to do. This step depends primarily upon error guessing, relying upon the experience of the test designer to anticipate problem areas.

Suitable techniques:

- Error guessing
- Boundary value analysis
- Internal boundary value testing
- State-transition testing

Step 4 - Special Considerations

Where appropriate, test cases should be designed to address issues such as performance, safety requirements and security requirements. Particularly in the cases of safety and security, it can be convenient to give test cases special emphasis to facilitate security analysis or safety analysis and certification. Test cases already designed which address security issues or safety hazards should be identified in the unit test specification. Further test cases should then be added to the unit test specification to ensure that all security issues and safety hazards applicable to the unit will be fully addressed.

Suitable techniques:

- Specification derived tests

2.5. Step 5 - Coverage Tests

The test coverage likely to be achieved by the designed test cases should be visualised. Further test cases can then be added to the unit test specification to achieve specific test coverage objectives. Once coverage tests have been designed, the test procedure can be developed and the tests executed.

Suitable techniques:

- Branch testing
- Condition testing
- Data definition-use testing
- State-transition testing

2.6. Test Execution

A test specification designed using the above five steps should in most cases provide a thorough test for a unit. At this point the test specification can be used to develop an actual test procedure, and the test procedure used to execute the tests. For users of AdaTEST or Cantata, the test procedure will be an AdaTEST or Cantata test script. Execution of the test procedure will identify errors in the unit which can be corrected and the unit re-tested. Dynamic analysis during

execution of the test procedure will yield a measure of test coverage, indicating whether coverage objectives have been achieved. There is therefore a further coverage completion step in the process of designing test specifications.

Step 6 - Coverage Completion

Depending upon an organisation's standards for the specification of a unit, there may be no structural specification of processing within a unit other than the code itself. There are also likely to have been human errors made in the development of a test specification. Consequently, there may be complex decision conditions, loops and branches within the code for which coverage targets may not have been met when tests were executed. Where coverage objectives are not achieved, analysis must be conducted to determine why. Failure to achieve a coverage objective may be due to:

- Infeasible paths or conditions - the corrective action should be to annotate the test specification to provide a detailed justification of why the path or condition is not tested. AdaTEST provides some facilities to help exclude infeasible conditions from Boolean coverage metrics.
- Unreachable or redundant code - the corrective action will probably be to delete the offending code. It is easy to make mistakes in this analysis, particularly where defensive programming techniques have been used. If there is any doubt, defensive programming should not be deleted.
- Insufficient test cases - test cases should be refined and further test cases added to a test specification to fill the gaps in test coverage.

Ideally, the coverage completion step should be conducted without looking at the actual code. However, in practice some sight of the code may be necessary in order to achieve coverage targets. It is vital that all test designers should recognize that use of the coverage completion step should be minimized. The most effective testing will come from analysis and specification, not from experimentation and over dependence upon the coverage completion step to cover for sloppy test design.

Suitable techniques:

- Branch testing
- Condition testing
- Data definition-use testing
- State-transition testing

2.8. General Guidance

Note that the first five steps in producing a test specification can be achieved:

- Solely from design documentation;
- Without looking at the actual code;
- Prior to developing the actual test procedure.

It is usually a good idea to avoid long sequences of test cases which depend upon the outcome of preceding test cases. An error identified by a test case early in the sequence could cause secondary errors and reduce the amount of real testing achieved when the tests are executed.

The process of designing test cases, including executing them as "thought experiments", often identifies bugs before the software has even been built. It is not uncommon to find more bugs when designing tests than when executing tests.

Throughout unit test design, the primary input should be the specification documents for the unit under test. While use of actual code as an input to the test design process may be necessary in some circumstances, test designers must take care that they are not testing the code against itself. A test specification developed from the code will only prove that the code does what the code does, not that it does what it is supposed to do.

3. Test Case Design Techniques

Test case design techniques can be broadly split into two main categories. Black box techniques use the interface to a unit and a description of functionality, but do not need to know how the inside of a unit is built. White box techniques make use of information about how the inside of a unit works. There are also some other techniques which do not fit into either of the above categories. Error guessing falls into this category.

Black box (functional)	White box (structural)	Other
Specification derived tests	Branch testing	Error guessing
Equivalence partitioning	Condition testing	
Boundary value analysis	Data definition-use testing	
State-transition testing	Internal boundary value testing	

Table 3.1 - Categories of Test Case Design Techniques

The most important ingredients of any test design are experience and common sense. Test designers should not let any of the given techniques obstruct the application of experience and common sense.

Specification Derived Tests

As the name suggests, test cases are designed by walking through the relevant specifications. Each test case should test one or more statements of specification. It is often practical to make the sequence of test cases correspond to the sequence of statements in the specification for the unit under test. For example, consider the specification for a function to calculate the square root of a real number, shown in figure 3.1.

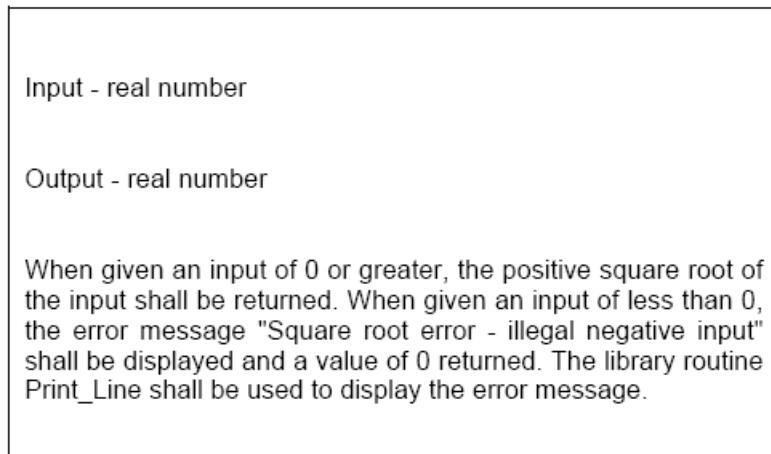


Figure 3.1 - Functional Specification for Square Root

There are three statements in this specification, which can be addressed by two test cases. Note that the use of Print_Line conveys structural information in the specification.

Test Case 1: Input 4, Return 2

- Exercises the first statement in the specification
("When given an input of 0 or greater, the positive square root of the input shall be returned.").

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.

- Exercises the second and third statements in the specification

("When given an input of less than 0, the error message "Square root error - illegal negative input" shall be displayed and a value of 0 returned. The library routine Print_Line shall be used to display the error message.").

Specification derived test cases can provide an excellent correspondence to the sequence of statements in the specification for the unit under test, enhancing the readability and maintainability of the test specification. However, specification derived testing is a positive test case design technique. Consequently, specification derived test cases have to be supplemented by negative test cases in order to provide a thorough unit test specification.

A variation of specification derived testing is to apply a similar technique to a security analysis, safety analysis, software hazard analysis, or other document which provides supplementary information to the unit's specification.

Equivalence Partitioning

Equivalence partitioning is a much more formalized method of test case design. It is based upon splitting the inputs and outputs of the software under test into a number of partitions, where the behaviour of the software is equivalent for any value within a particular partition. Data which forms partitions is not just routine parameters. Partitions can also be present in data accessed by the software, in time, in input and output sequence, and in state. Equivalence partitioning assumes that all values within any individual partition are equivalent for test purposes. Test cases should therefore be designed to test one value in each partition. Consider again the square root

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.

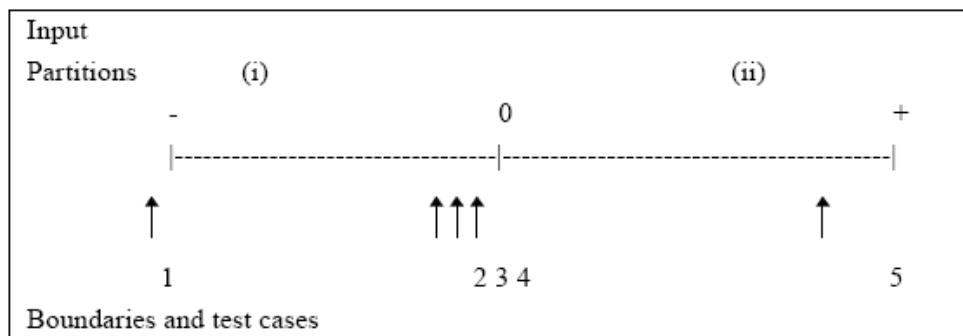


Figure 3.2 - Input Partition Boundaries in Square Root

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

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line. - Exercises the invalid input processing branch

However, there could be many different structural implementations of the square root function. The following structural specifications are all valid implementations of the square root function, but the above test cases would only achieve decision coverage of the first and third versions of the specification.

```

If input<0 THEN
    CALL Print_Line "Square root error - illegal negative
    input"
    RETURN 0
ELSE
    Use maths co-processor to calculate the answer
    RETURN the answer
END_IF

```

```

If input<0 THEN
    CALL Print_Line "Square root error - illegal negative
    input"
    RETURN 0
ELSE
    IF input=0 THEN
        RETURN 0
    ELSE
        Use maths co-processor to calculate the
        answer
        RETURN the answer
    END_IF
END_IF

```

Figure 3.3(a) - Specification 1

Figure 3.3(b) - Specification 2

```

Use maths co-processor to calculate the answer
Examine co-processor status registers
If status=error THEN
    CALL Print_Line "Square root error - illegal negative
    input"
    RETURN 0
ELSE
    RETURN the answer
END_IF

```

```

If input<0 THEN
    CALL Print_Line "Square root error - illegal negative
    input"
    RETURN 0
ELSE_IF input=0 THEN
    RETURN 0
ELSE
    Calculate first approximation
    LOOP
        Calculate error
        EXIT_LOOP WHEN error<desired
        accuracy
        Adjust approximation
    END_LOOP
    RETURN the answer
END_IF

```

Figure 3.3(c) - Specification 3

Figure 3.3(d) - Specification 4

It can be seen that branch testing works best with a structural specification for the unit. A structural unit specification will enable branch test cases to be designed to achieve decision coverage, but a purely functional unit specification could lead to coverage gaps.

One thing to beware of is that by concentrating upon branches, a test designer could lose sight of the overall functionality of a unit. It is important to always remember that it is the overall

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 4). 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!

It is possible to split data definition-use tests into two categories: uses which affect control flow (predicate uses) and uses which are purely computational. Refer to "Software Testing Techniques" 2nd Edition, B Beizer, Van Nostrand Reinhold, New York 1990, for a more detailed description of predicate and computational uses.

3.8. Internal Boundary Value Testing

In many cases, partitions and their boundaries can be identified from a functional specification for a unit, as described under equivalence partitioning and boundary value analysis above. However, a unit may also have internal boundary values which can only be identified from a structural specification. Consider a fragment of the successive approximation version of the square root unit specification, as shown in figure 3.5 (derived from figure 3.3(d) - Specification 4).

```
:  
:  
Calculate first approximation  
LOOP  
    Calculate error  
    EXIT_LOOP WHEN error<desired accuracy  
    Adjust approximation  
END_LOOP  
RETURN the answer  
:  
:
```

Figure 3.5 - Fragment of Specification 4

The calculated error can be in one of two partitions about the desired accuracy, a feature of the structural design for the unit which is not apparent from a purely functional specification. An analysis of internal boundary values yields three conditions for which test cases need to be designed.

Test Case 1: Error just greater than the desired accuracy

Test Case 2: Error equal to the desired accuracy

Test Case 3: Error just less than the desired accuracy

Internal boundary value testing can help to bring out some elusive bugs. For example, suppose " \leq " had been coded instead of the specified " $<$ ". Nevertheless, internal boundary value testing is a luxury to be applied only as a final supplement to other test case design techniques.

3.9. Error Guessing

Error guessing is based mostly upon experience, with some assistance from other techniques such as boundary value analysis. Based on experience, the test designer guesses the types of errors that could occur in a particular type of software and designs test cases to uncover them. For example, if any type of resource is allocated dynamically, a good place to look for errors is in the deallocation of resources. Are all resources correctly deallocated, or are some lost as the software executes?

Error guessing by an experienced engineer is probably the single most effective method of designing tests which uncover bugs. A well placed error guess can show a bug which could easily be missed by many of the other test case design techniques presented in this paper. Conversely, in the wrong hands error guessing can be a waste of time.

To make the maximum use of available experience and to add some structure to this test case design technique, it is a good idea to build a check list of types of errors. This check list can then be used to help "guess" where errors may occur within a unit. The check list should be maintained with the benefit of experience gained in earlier unit tests, helping to improve the overall effectiveness of error guessing.

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 not 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.

Step (8)

Test unit H, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F and G, and stubs for units I and J.

Step (9)

Test unit I, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G and H, and a stub for units J.

Step (10)

Test unit J, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G, H and I.

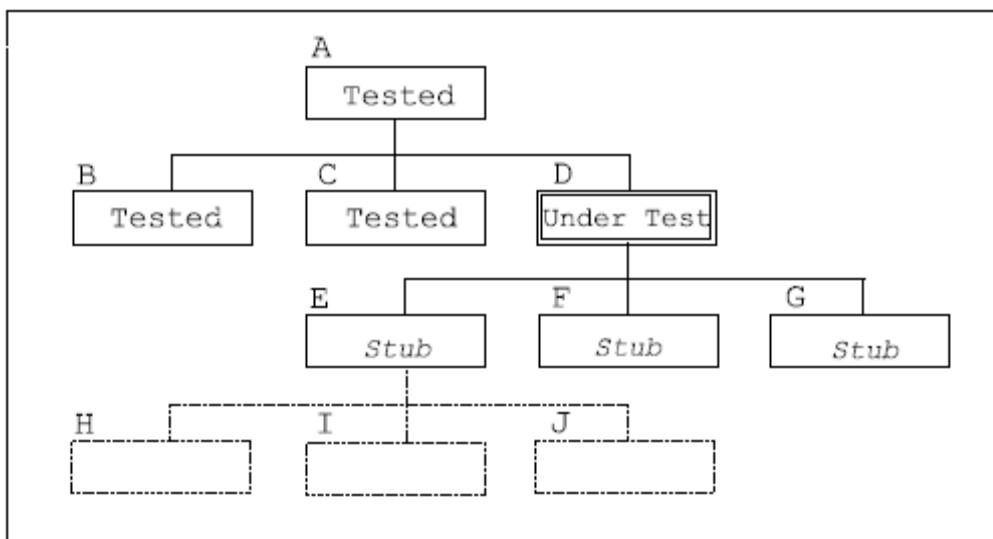


Figure 2.1 - Top Down Testing

Advantages

Top down unit testing provides an early integration of units before the software integration phase. In fact, top down unit testing is really a combined unit test and software integration strategy.

The detailed design of units is top down, and top down unit testing implements tests in the sequence units are designed, so development time can be shortened by overlapping unit testing with the detailed design and code phases of the software lifecycle.

In a conventionally structured design, where units at the top of the hierarchy provide high level functions, with units at the bottom of the hierarchy implementing details, top down unit testing will provide an early integration of 'visible' functionality. This gives a very requirements oriented approach to unit testing.

Redundant functionality in lower level units will be identified by top down unit testing, because there will be no route to test it. (However, there can be some difficulty in distinguishing between redundant functionality and untested functionality).

Disadvantages

Top down unit testing is controlled by stubs, with test cases often spread across many stubs. With each unit tested, testing becomes more complicated, and consequently more expensive to develop and maintain.

As testing progresses down the unit hierarchy, it also becomes more difficult to achieve the good structural coverage which is essential for high integrity and safety critical applications, and which are required by many standards. Difficulty in achieving structural coverage can also lead to a confusion between genuinely redundant functionality and untested functionality. Testing some low level functionality, especially error handling code, can be totally impractical.

Changes to a unit often impact the testing of sibling units and units below it in the hierarchy. For example, consider a change to unit D. Obviously, the unit test for unit D would have to change and be repeated. In addition, unit tests for units E, F, G, H, I and J, which use the tested unit D, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit D, even though units E, F, G, H, I and J had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The design of test cases for top down unit testing requires structural knowledge of when the unit under test calls other units. The sequence in which units can be tested is constrained by the hierarchy of units, with lower units having to wait for higher units to be tested, forcing a 'long and thin' unit test phase. (However, this can overlap substantially with the detailed design and code phases of the software lifecycle).

The relationships between units in the example program in figure 2.1 is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. All of the disadvantages of a top down approach to unit testing are compounded by a unit being referenced from more than one other unit.

Overall

A top down strategy will cost more than an isolation based strategy, due to complexity of testing units below the top of the unit hierarchy, and the high impact of changes. The top down organisational approach is not a good choice for unit testing. However, a top down approach to the integration of units, where the units have already been tested in isolation, can be viable.

Bottom up Testing

Description

In **bottom up** unit testing, units are tested in isolation from the units which call them, but using the actual units called as part of the test.

The lowest level units are tested first, then used to facilitate the testing of higher level units. Other units are then tested, using previously tested called units. The process is repeated until the unit at the top of the hierarchy has been tested. Bottom up testing requires test drivers, but does not require test stubs.

Figure 3.1 illustrates the test driver and tested units needed to test unit D, assuming that units E, F, G, H, I and J have already been tested in a bottom up approach.

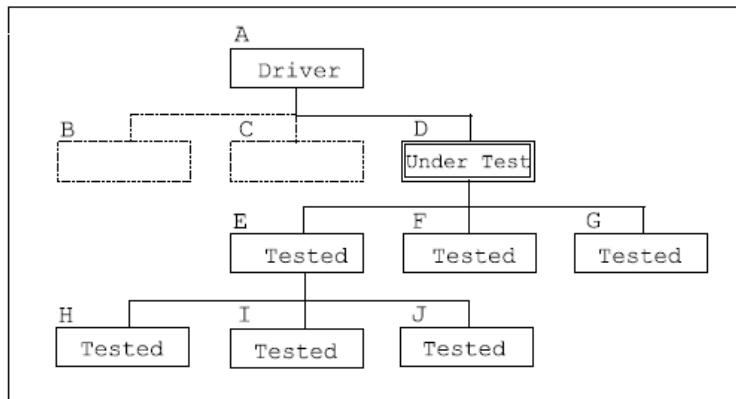


Figure 3.1 - Bottom Up Testing

Bottom up Organizational approach, could read as follows:

Step (1)

(Note that the sequence of tests within this step is unimportant, all tests within step 1 could be executed in parallel.)

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;
 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 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.

Step (2)

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

Step (3)

Test unit D, using a driver to call it in place of unit A and tested units E, F, G, H, I and J.
 (Shown in figure 3.1).

Step (4)

Test unit A, using tested units B, C, D, E, F, G, H, I and J.

Advantages

Like top down unit testing, bottom up unit testing provides an early integration of units before the software integration phase. Bottom up unit testing is also really a combined unit test and software integration strategy. All test cases are controlled solely by the test driver, with no stubs required. This can make unit tests near the bottom of the unit hierarchy relatively simple. (However, higher level unit tests can be very complicated). Test cases for bottom up testing may be designed solely from functional design information, requiring no structural design information (although structural design information may be useful in achieving full coverage). This makes the bottom up approach to unit testing useful when the detailed design documentation lacks structural detail.

Bottom up unit testing provides an early integration of low level functionality, with higher level functionality being added in layers as unit testing progresses up the unit hierarchy. This makes bottom up unit testing readily compatible with the testing of objects.

Disadvantages

As testing progresses up the unit hierarchy, bottom up unit testing becomes more complicated, and consequently more expensive to develop and maintain. As testing progresses up the unit hierarchy, it also becomes more difficult to achieve good structural coverage. Changes to a unit often impact the testing of units above it in the hierarchy. For example, consider a change to unit H. Obviously, the unit test for unit H would have to change and be repeated. In addition, unit tests for units A, D and E, which use the tested unit H, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit H, even though units A, D and E had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The sequence in which units can be tested is constrained by the hierarchy of units, with higher units having to wait for lower units to be tested, forcing a 'long and thin' unit test phase. The first units to be tested are the last units to be designed, so unit testing cannot overlap with the detailed design phase of the software lifecycle. The relationships between units in the example program in figure 2.2 is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. As for top down unit testing, the disadvantages of a bottom up approach to unit testing are compounded by a unit being referenced from more than one other unit.

Overall

The bottom up organisational approach can be a reasonable choice for unit testing, particularly when objects and reuse are considered. However, the bottom up approach is biased towards functional testing, rather than structural testing. This can present difficulties in achieving the high levels of structural coverage essential for high integrity and safety critical applications, and which are required by many standards. The bottom up approach to unit testing conflicts with the tight timescales required of many software developments. Overall, a bottom up strategy will cost more than an isolation based strategy, due to complexity of testing units above the bottom level in the unit hierarchy and the high impact of changes.

Isolation Testing

Description

Isolation testing tests each unit in isolation from the units which call it and the units it calls. Units can be tested in any sequence, because no unit test requires any other unit to have been tested. Each unit test requires a test driver and all called units are replaced by stubs. Figure 4.1 illustrates the test driver and tested stubs needed to test unit D.

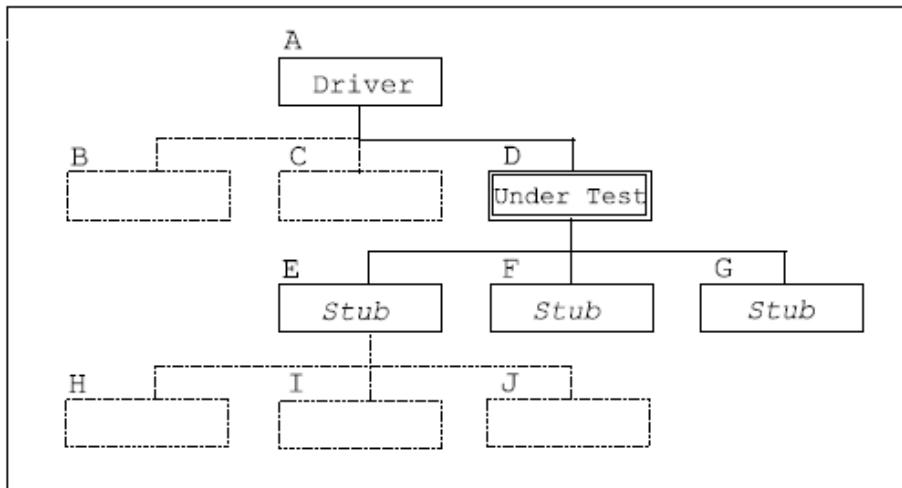


Figure 4.1 - Isolation Testing

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

when changes are made. An isolation approach provides a distinct separation of unit testing from integration testing, allowing developers to focus on unit testing during the unit test phase of the software lifecycle, and on integration testing during the integration phase of the software lifecycle. Isolation testing is the only pure approach to unit testing, both top down testing and bottom up testing result in a hybrid of the unit test and integration phases. Unlike the top down and bottom up approaches, the isolation approach to unit testing is not affected by a unit being referenced from more than one other unit.

Disadvantages

The main disadvantage of an isolation approach to unit testing is that it does not provide any early integration of units. Integration has to wait for the integration phase of the software lifecycle. (Is this really a disadvantage?).

An isolation approach to unit testing requires structural design information and the use of both stubs and drivers. This can lead to higher costs than bottom up testing for units near the bottom of the unit hierarchy. However, this will be compensated by simplified testing for units higher in the unit hierarchy, together with lower costs each time a unit is changed.

Overall

An isolation approach to unit testing is the best overall choice. When supplemented with an appropriate integration strategy, it enables shorter development timescales and provides the lowest cost, both during development and for the overall lifecycle. Following unit testing in isolation, tested units can be integrated in a top down or bottom up sequence, or any convenient groupings and combinations of groupings. However, a bottom up integration is the most compatible strategy with current trends in object oriented and object biased designs.

An isolation approach to unit testing is the best way of achieving the high levels of structural coverage essential for high integrity and safety critical applications, and which are required by many standards. With all the difficult work of achieving good structural coverage achieved by unit testing, integration testing can concentrate on overall functionality and the interactions between units.

WRITING EFFECTIVE DEFECT REPORTS

Introduction

Defect reports are among the most important deliverables to come out of test. They are as important as the test plan and will have more impact on the quality of the product than most other deliverables from test. It is worth the effort to learn how to write effective defect reports. Effective defect reports will:

- ~ Reduce the number of defects returned from development
- ~ Improve the speed of getting defect fixes
- ~ Improve the credibility of test
- ~ Enhance teamwork between test and development

Why do some testers get a much better response from development than others? Part of the answer lies in the defect report. Following a few simple rules can smooth the way for a much more productive environment. The objective is not to write the perfect defect report, but to write an effective defect report that conveys the proper message, gets the job done, and simplifies the process for everyone.

It focuses on two aspects of defect reports, 1) the remarks or description and 2) the abstract. First, lets take a look at the essentials for writing effective remarks.

Defect Remarks

Here are some key points to make sure the next defect report you write is an effective one.

1. **Condense** - Say it clearly but briefly
2. **Accurate** - Is it a defect or could it be user error, misunderstanding, etc.?
3. **Neutralize** - Just the facts. No zingers. No humor. No emotion.
4. **Precise** - Explicitly, what is the problem?
5. **Isolate** - What has been done to isolate the problem?
6. **Generalize** - What has been done to understand how general the problem is?
7. **Re-create** - What are the essentials in triggering/re-creating this problem? (environment, steps, conditions)
8. **Impact** - What is the impact to the customer? What is the impact to test? Sell the defect.
9. **Debug** - What does development need to make it easier to debug? (traces, dumps, logs, immediate access, etc.)
10. **Evidence** - What documentation will prove the existence of the error?

It is not just good technical writing skills that leads to effective defect reports. It is more important to make sure that you have asked and answered the right questions. It is key to make sure that you have covered the essential items that will be of most benefit to the intended audience of the defect report.

Essentials for Effective Defect Remarks

Condense

Say it clearly but briefly. First, eliminate unnecessary wordiness. Second, don't add in extraneous information. It is important that you include all relevant information, but make sure that the information is relevant. In situations where it is unclear how to reproduce the problem or the understanding of the problem is vague for whatever reason you will probably need to capture more information. Keep in mind that irrelevant information can be just as problematic as too little relevant information.

Condense Example	Defect Remark
<p>Don't:</p> <p>Suffers from TMI (Too Much Information), most of which is not helpful.</p>	<p>I was setting up a test whose real intent was to detect memory errors. In the process I noticed a new GUI field that I was not familiar with. I decided to exercise the new field. I tried many boundary and error conditions that worked just fine. Finally, I cleared the field of any data and attempted to advance to the next screen, then the program abended. Several retries revealed that anytime there is not any data for the "product description" field you cannot advance to the next screen or even exit or cancel without abending.</p>
<p>Do:</p>	<p>The "exit", "next", and "cancel" functions for the "Product Information" screen abends when the "product description" field is empty or blank.</p>

Accurate

Make sure that what you are reporting is really a bug. You can lose credibility very quickly if you get a reputation of reporting problems that turn out to be setup problems, user errors, or misunderstandings of the product. Before you write up the problem, make sure that you have done your homework. Before writing up the problem consider:

- ~ Is there something in the setup that could have caused this? For example, are the correct versions installed and all dependencies met? Did you use the correct login, security, command/task sequence and so fourth?
- ~ Could an incomplete cleanup, incomplete results, or manual interventions from a previous test cause this?
- ~ Could this be the result of a network or some other environmental problem?
- ~ Do you really understand how this is supposed to work?

There are always numerous influences that can affect the outcome of a test. Make sure that you understand what these influences are and consider their role in the perceived bug you are reporting. This is one area that quickly separates the experienced tester from the novice. If you are unsure about the validity of the problem it may be wise to consult with an experienced tester or developer prior to writing up the problem.

As a rule of thumb it helps to remember the adage that “it is a sin to over report, but it is a crime to under report.” Don’t be afraid to write up problems. Do your best to ensure that they are valid problems. When you discover that you have opened a problem and it turns out to be an incorrectly reported problem, make sure that you learn from it.

Neutralize

State the problem objectively. Don’t try to use humor and don’t use emotionally charged zingers. What you think is funny when you write the defect may not be interpreted as funny by a developer who is working overtime and is stressed by deadlines. Using emotionally charged statements doesn’t do anything for fixing the problem. Emotional statements just create barriers to communication and teamwork. Even if the developers doubted you and returned your previous defect and now you have proof that you are correct and they are wrong, just state the problem and the additional information that will be helpful to the developer. In the long run this added bit of professionalism will gain you respect and credibility. Read over your problem description before submitting it and remove or restate those comments that could be interpreted as being negative towards a person.

Neutralize Example:	Defect Remark
This example is a response to a developer returning a defect for more information and requesting more details on what values caused the problem.	
Don't: The first clause will probably be interpreted as a jab at the developer and adds no useful information.	As could have been determined from the original defect with very little effort, function ABC does indeed abend with any negative value as input.
Do:	Function ABC abends with any negative value. Examples of some values tested include -1, -36, -32767.

Precise

The person reading the problem description should not have to be a detective to determine what the problem is. Right up front in the description, describe exactly what you perceive the problem to be. Some descriptions detail a series of actions and results. For example, “I hit the enter key and action A happened. Then I hit the back arrow and action B happened. Then I entered the “xyz” command and action C happened.” The reader may not know if you think all three resulting actions were incorrect, or which one, if any is incorrect. In all cases, but especially if the description is long, you need to summarize the problem(s) at the beginning of the description. Don’t depend on an abstract in a different field of the defect report to be available or used by everyone who reads the problem description. Don’t assume that others will draw the same conclusions that you do. Your goal is not to write a description that is possible to understand, but to write a description that cannot be misunderstood. The only way to make that happen is to explicitly and precisely describe the problem rather than just giving a description of what happened.

Precise Example	Defect Remark
<p>Don't:</p> <p>In this example, it is hard to tell if the problem is 1) the twinax port not timing out or 2) the printer not returning to ready or 3) the message on the op panel.</p>	<p>Issuing a cancel print when job is in PRT state (job is already in the printer and AS/400 is waiting to receive print complete from printer) causes the Twinax port to not time out. The printer never returns to a READY state and indefinitely displays "PRINTING IPDS FROM TRAY1" in the op-panel.</p>
<p>Do:</p> <p>Precede the description with a short summary of exactly what you perceive the problem to be.</p>	<p>Canceling a job while it is printing causes the printer to hang.</p> <p>Issuing a cancel print when job is in PRT state (job is already in the printer and AS/400 is waiting to receive print complete from printer) causes the Twinax port to not time out. The printer never returns to a READY state and indefinitely displays "PRINTING IPDS FROM TRAY1" in the op-panel.</p>

Isolate

Each organization has its own philosophy and expectations on how much the tester is required to isolate the problem. Regardless of what is required, a tester should always invest some reasonable amount of effort into isolating the problem. Consider the following when isolating problems.

- ~ Try to find the shortest, simplest set of the steps required to reproduce the problem. That usually goes a long way towards isolating the problem.
- ~ Ask yourself if anything external to the specific code being tested contributed to the problem. For example, if you experience a hang or delay, could it have been due to a network problem? If you are doing end-to-end testing can you tell which component along the way had the failure? Are there some things you could do to help narrow down which component had the failure?
- ~ If your test has multiple input conditions, vary the inputs until you can find which one with which values triggered the problem. In the problem description, to the extent possible, describe the exact inputs used. For example, if you found a problem while printing a Postscript document, even if you think the problem occurs with any Postscript document, specify the exact document that you used to find the problem.

Your ability to isolate, in large part, defines your value-add as a tester. Effective isolation saves everyone along the line a great deal of time. It also saves you a lot of time when you have to verify a fix.

Generalize

Often times, the developers will fix exactly what you report, without even realizing the problem is a more general problem that needs a more general fix. For example, I may report that my word processor "save file" function failed and the word processor abended when I tried to save the file "myfile". A little more investigation may have revealed that this same failure occurs anytime I save a zero length file. Perhaps, on this release it abends on every save to a remote disk, a read only disk, and so forth. To already know this when you write the report will save the developer a lot of time and enhance the possibility of a better fix to handle the general case. When you detect a problem, take reasonable steps to determine if it is more general than is immediately obvious

Generalize Example	Defect Remark
Don't:	Error message for "file not found" error has garbage characters for the file name.
Do:	Error message for "file not found" error has garbage characters for the file name. Every message I tried that expected data to be inserted in the message had the same problem. Messages without inserts were okay.

Re-create

Some bugs are easy to re-create and some are not. If you can re-create the bug you should explain exactly what is required to do the re-create. You should list all the steps, include the exact syntax, file names, sequences that you used to encounter or re-create the problem. If you believe that the problem will happen with any file, any sequence, etc. then mention that but still provide an explicit example that can be used to do the re-create. If in your effort to verify that the bug is re-creatable you find a shorter and reliable means of re-creating, document the shortest, easiest means of re-creation.

If you cannot re-create the problem or if you suspect that you may not be able to re-create the problem gather all the relevant information that you can that may provide useful information to the person who has to try and fix the problem. This may be a time when you consider asking a developer if they want to examine the system while it is still in the problem state or if there is any particular information that should be captured before cleaning up the problem state and restoring the system. Don't assume that it can be re-created if you haven't verified that it can be re-created. If you cannot or have not re-created the problem it is important to note that in the defect remarks.

Impact

What is the impact if the bug were to surface in the customer environment? The impact of some bugs is self-evident. For example, "entire system crashes when I hit the enter key." Some bugs are not so obvious. For example, you may discover a typo on a window. This may seem very minor, even trivial unless you point out that every time someone uses your product this is the first thing they see and the typo results in an offensive word. In this case, even though it is just a typo it may be something that absolutely must be fixed prior to shipping the product. Make your best judgment. If you think it is possible that this defect will not get sufficient priority then state the potential impact and sell the defect.

Don't oversell, but make sure the readers of the defect have an accurate understanding of the probable impact on the customer.

Debug

What will the developer need to be able to debug this problem? Are there traces, dumps, logs, and so forth that should be captured and made available with this defect report? Document what has been captured and how it can be accessed.

Evidence

What exists that will prove the existence of the error? Have you provided both the expected results and the actual results? Is there documentation that supports your expected results? Since you are writing a problem report it is obvious that you believe there is a problem. Provide anything

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.

DEFECT REMARK TEMPLATE

Product Details:

Product Name and Number:	
Version, Revision, build and disk number:	

System Details:

Computer Type: PC model, mainframe type, OS Level, etc.	
Memory:	
Disk Space:	
Peripherals attached and used:	
Network connectivity:	
Configuration Details:	

Problem Summary:

Problem Description: (include expected and actual results)

Is this reproducible?

Steps and conditions to reproduce:

Has this problem been isolated?

Has this problem been generalized?

Additional Debug Information: (How to access logs, dumps, etc.)

In effective defect reporting, as in many situations, it is not a matter of if you got the answers correct but more a matter of did you answer the correct questions? These ten points:

- ~ Condense
- ~ Accurate
- ~ Neutralize
- ~ Precise
- ~ Isolate
- ~ Generalize
- ~ Re-create
- ~ Impact
- ~ Debug
- ~ Evidence

Provide a quick checklist to ensure that your defect reports answer the right questions that will be of most benefit to your organization.

Defect Abstracts

The short one line abstract that gets associated with most defects is a very powerful communication tool. Often times, the abstract is the only portion of the defect that gets read by the decision-makers. It is the abstract, not the full description that gets included in reports. It is the abstract that the project managers, screeners, team leads and other managers look at when trying to understand the defects associated with the product.

The abstract must be concise and descriptive and convey an accurate message. The abstract is usually very limited in length. Because of the space limitations, abbreviations are okay and short accurate messages take priority over good grammar. A good use of key words is essential since many searches are based on the abstract. Keywords such as abend, hang, typo and so forth are both descriptive and prove useful as search words. Where space permits it is helpful to mention the environment, the impact, and any of the who, what, when, where, why questions that you can address in such a short space. Some defect tracking tools provide default abstracts by using the first line of the problem description or similar defaulting mechanisms. Never take the default abstract. Be as specific as possible. For example, the following abstract is true but doesn't provide nearly as much information as it could.

Abstract: Problems found when saving and restoring data member. Perhaps a more descriptive abstract would be:

Abstract: xyz's save/restore of data member on WinNT fails, data corrupted You can never get everything you want in an abstract. Here is a list of items and tips that you try to include in an abstract.

Abstract Checklist

Mandatory:

1. Concisely, explicitly state what the problem is. (not just that there is a problem)

Recommended (space permitting):

1. Use meaningful keywords
2. State environment and impact
3. Answer who, what, when, where, why, and how
4. Okay to use abbreviations
5. Grammar is secondary over conveying the message

6. Don't use defaults

Summary

Testers spend a significant amount of time seeking out and discovering software problems. Once detected, it greatly enhances productivity to report the defect in such a way as to increase the likelihood of getting the problem fixed with the least amount of effort. Making sure that the proper information is provided is more important than superior writing skills.

- ~ **Condense**
- ~ **Accurate**
- ~ **Neutralize**
- ~ **Precise**
- ~ **Isolate**
- ~ **Generalize**
- ~ **Re-create**
- ~ **Impact**
- ~ **Debug**
- ~ **Evidence**

Will go a long way toward help you provide the right information in every defect report. Not everyone reads the entire defect report. Many decision-makers rely on the one-line defect abstract to base their decisions on. It is important to write abstracts that accurately convey the right message about the abstract.

The better you are at writing defect reports and abstracts, the more likely it is that the problems will actually get fixed and in a more timely manner. Your credibility and value-add to the business will increase as developers, managers, and other testers are better able to do their jobs because your defect reports are well written and reliable.

Software Testing Framework

Testing Framework

Through experience they determined, that there should be 30 defects per 1000 lines of code. If testing does not uncover 30 defects, a logical solution is that the test process was not effective.

1.0 Introduction

Testing plays an important role in today's System Development Life Cycle. During Testing, we follow a systematic procedure to uncover defects at various stages of the life cycle.

This framework is aimed at providing the reader various Test Types, Test Phases, Test Models and Test Metrics and guide as to how to perform effective Testing in the project.

All the definitions and standards mentioned in this framework are existing one's. I have not altered any definitions, but where ever possible I tried to explain them in simple words. Also, the framework, approach and suggestions are my experiences. My intention of this framework is to help Test Engineers to understand the concepts of testing, various techniques and apply them effectively in their daily work. This framework is not for publication or for monetary distribution.

If you have any queries, suggestions for improvements or any points found missing, kindly write back to me.

1.2 Traditional Testing Cycle

Let us look at the traditional Software Development life cycle. The figure below depicts the same.

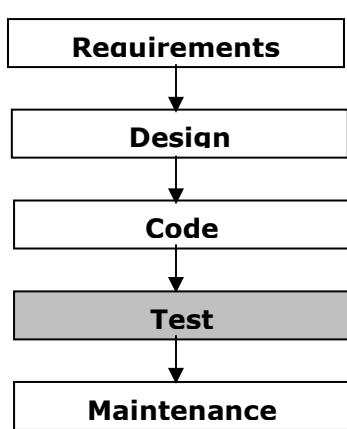


Fig A

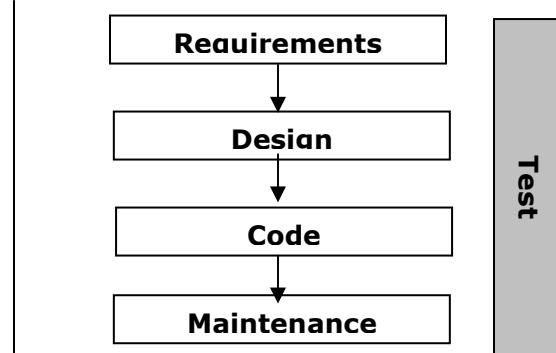


Fig B

In the above diagram (Fig A), the Testing phase comes after the Coding is complete and before the product is launched and goes into maintenance.

But, the recommended test process involves testing in every phase of the life cycle (Fig B). During the requirement phase, the emphasis is upon validation to determine that the defined requirements meet the needs of the project. During the design and program phases, the

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	<ul style="list-style-type: none"> Determine verification approach. Determine adequacy of requirements. Generate functional test data. Determine consistency of design with requirements.
Design	<ul style="list-style-type: none"> Determine adequacy of design. Generate structural and functional test data. Determine consistency with design
Program (Build)	<ul style="list-style-type: none"> Determine adequacy of implementation Generate structural and functional test data for programs.
Test	<ul style="list-style-type: none"> Test application system.
Installation	<ul style="list-style-type: none"> Place tested system into production.
Maintenance	<ul style="list-style-type: none"> 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.

		design specification.	
--	--	-----------------------	--

2.1.1 Review's

The focus of Review is on a work product (e.g. Requirements document, Code etc.). After the work product is developed, the Project Leader calls for a Review. The work product is distributed to the personnel who involves in the review. The main audience for the review should be the Project Manager, Project Leader and the Producer of the work product.

Major reviews include the following:

- 1. In Process Reviews**
- 2. Decision Point or Phase End Reviews**
- 3. Post Implementation Reviews**

Let us discuss in brief about the above mentioned reviews. As per statistics Reviews uncover over 65% of the defects and testing uncovers around 30%. So, it's very important to maintain reviews as part of the V&V strategies.

In-Process Review

In-Process Review looks at the product during a specific time period of a life cycle, such as activity. They are usually limited to a segment of a project, with the goal of identifying defects as work progresses, rather than at the close of a phase or even later, when they are more costly to correct.

Decision-Point or Phase-End Review

This review looks at the product for the main purpose of determining whether to continue with planned activities. They are held at the end of each phase, in a semiformal or formal way. Defects found are tracked through resolution, usually by way of the existing defect tracking system. The common phase-end reviews are *Software Requirements Review*, *Critical Design Review* and *Test Readiness Review*.

- The **Software Requirements Review** is aimed at validating and approving the documented software requirements for the purpose of establishing a baseline and identifying analysis packages. The Development Plan, Software Test Plan, Configuration Management Plan are some of the documents reviews during this phase.
- The **Critical Design Review** baselines the detailed design specification. Test cases are reviewed and approved.
- The **Test Readiness Review** is performed when the appropriate application components are near completing. This review will determine the readiness of the application for system and acceptance testing.

Post Implementation Review

These reviews are held after implementation is complete to audit the process based on actual results. Post-Implementation reviews are also known as *Postmortems* and are held to assess the success of the overall process after release and identify any opportunities for process improvement. They can be held up to three to six months after implementation, and are conducted in a format.

There are three general classes of reviews:

- 1. Informal or Peer Review**
- 2. Semiformal or Walk-Through**

3. Format or Inspections

Peer Review is generally a one-to-one meeting between the author of a work product and a peer, initiated as a request for input regarding a particular artifact or problem. There is no agenda, and results are not formally reported. These reviews occur on an as needed basis throughout each phase of a project.

2.1.2 Inspections

A knowledgeable individual called a moderator, who is not a member of the team or the author of the product under review, facilitates inspections. A recorder who records the defects found and actions assigned assists the moderator. The meeting is planned in advance and material is distributed to all the participants and the participants are expected to attend the meeting well prepared. The issues raised during the meeting are documented and circulated among the members present and the management.

2.1.3 Walkthroughs

The author of the material being reviewed facilitates walk-Through. The participants are led through the material in one of two formats; the presentation is made without interruptions and comments are made at the end, or comments are made throughout. In either case, the issues raised are captured and published in a report distributed to the participants. Possible solutions for uncovered defects are not discussed during the review.

2.2 Validation Strategies

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

Validation Strategy	Performed By	Explanation	Deliverable
Unit Testing.	Developers / Test Engineers.	Testing of single program, modules, or unit of code.	Software unit ready for testing with other system component.
Integration Testing.	Test Engineers.	Testing of integrated programs, modules, or units of code.	Portions of the system ready for testing with other portions of the system.
System Testing.	Test Engineers.	Testing of entire computer system. This kind of testing usually includes functional and structural testing.	Tested computer system, based on what was specified to be developed.
Production Environment Testing.	Developers, Test Engineers.	Testing of the whole computer system before rolling out to the UAT.	Stable application.
User Acceptance Testing.	Users.	Testing of computer system to make sure it will work in the system regardless of what the system requirements indicate.	Tested and accepted system based on the user needs.
Installation Testing.	Test Engineers.	Testing of the Computer System	Successfully installed application.

		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 known 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 least 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.

3.1.2 Flow Graph Notation

The flow graph depicts logical control flow using a diagrammatic notation. Each structured construct has a corresponding flow graph symbol.

3.1.3 Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of a basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.

Computing Cyclomatic Complexity

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of the three ways:

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity, $V(G)$, for a flow graph, G is defined as

$$V(G) = E - N + 2$$

Where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, $V(G)$ for a flow graph, G is also defined as:

$$V(G) = P + 1$$

Where P is the number of predicate nodes contained in the flow graph G .

3.1.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix* can be quite useful.

A *Graph Matrix* is a square matrix whose size is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections between nodes.

3.1.5 Control Structure Testing

Described below are some of the variations of Control Structure Testing.

3.1.5.1 Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module.

3.1.5.2 Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

3.1.6 Loop Testing

Loop Testing is a white box testing technique that focuses exclusively on the validity of loop constructs. Four classes of loops can be defined: Simple loops, Concatenated loops, nested loops, and unstructured loops.

3.1.6.1 Simple Loops

The following sets of tests can be applied to simple loops, where 'n' is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.

4. 'm' passes through the loop where $m < n$.
5. $n-1, n, n+1$ passes through the loop.

3.1.6.2 Nested Loops

If we extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values. Add other tests for out-of-range or exclude values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

3.1.6.3 Concatenated Loops

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.

3.1.6.4 Unstructured Loops

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

3.2 Black Box Testing

Black box testing, also known as behavioral testing focuses on the functional requirements of the software. All the functional requirements of the program will be used to derive sets of input conditions for testing.

Black Box Testing Types

The following are the most famous/frequently used Black Box Testing Types.

3.2.1 Graph Based Testing Methods

Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and error are uncovered.

3.2.2 Equivalence Partitioning

Equivalence partitioning is a black box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

EP can be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and one two invalid classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

3.2.3 Boundary Value Analysis

BVA is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning.

3.2.4 Comparison Testing

Situations where independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer based system. These independent versions from the basis of a black box testing technique called Comparison testing or back-to-back testing.

3.2.5 Orthogonal Array Testing

The orthogonal array testing method is particularly useful in finding errors associated with region faults – an error category associated with faulty logic within a software component.

3.3 Scenario Based Testing (SBT)

What is Scenario Based Testing and How/Where is it useful is an interesting question. I shall explain in brief the above two mentioned points.

Scenario Based Testing is categorized under Black Box Tests and are most helpful when the testing is concentrated on the Business logic and functional behavior of the application. Adopting SBT is effective when testing complex applications. Now, every application is complex, then it's the teams call as to implement SBT or not. I would personally suggest using SBT when the functionality to test includes various features and functions. A best example would be while testing banking application. As banking applications require utmost care while testing, handling various functions in a single scenario would result in effective results.

A sample transaction (scenario) can be, a customer logging into the application, checking his balance, transferring amount to another account, paying his bills, checking his balance again and logging out.

In brief, use Scenario Based Tests when:

1. Testing complex applications.
2. Testing Business functionality.

When designing scenarios, keep in mind:

1. The scenario should be close to the real life scenario.
2. Scenarios should be realistic.
3. Scenarios should be traceable to any/combination of functionality.
4. Scenarios should be supported by sufficient data.

3.4 Exploratory Testing

Exploratory Tests are categorized under Black Box Tests and are aimed at testing in conditions when sufficient time is not available for testing or proper documentation is not available.

Exploratory testing is 'Testing while Exploring'. When you have no idea of how the application works, exploring the application with the intent of finding errors can be termed as Exploratory Testing.

Performing Exploratory Testing

This is one big question for many people. The following can be used to perform Exploratory Testing:

- Learn the Application.
- Learn the Business for which the application is addressed.

- Learn the technology to the maximum extent on which the application has been designed.
- Learn how to test.
- Plan and Design tests as per the learning.

4.0 Structural System Testing Techniques

The following are the structural system testing techniques.

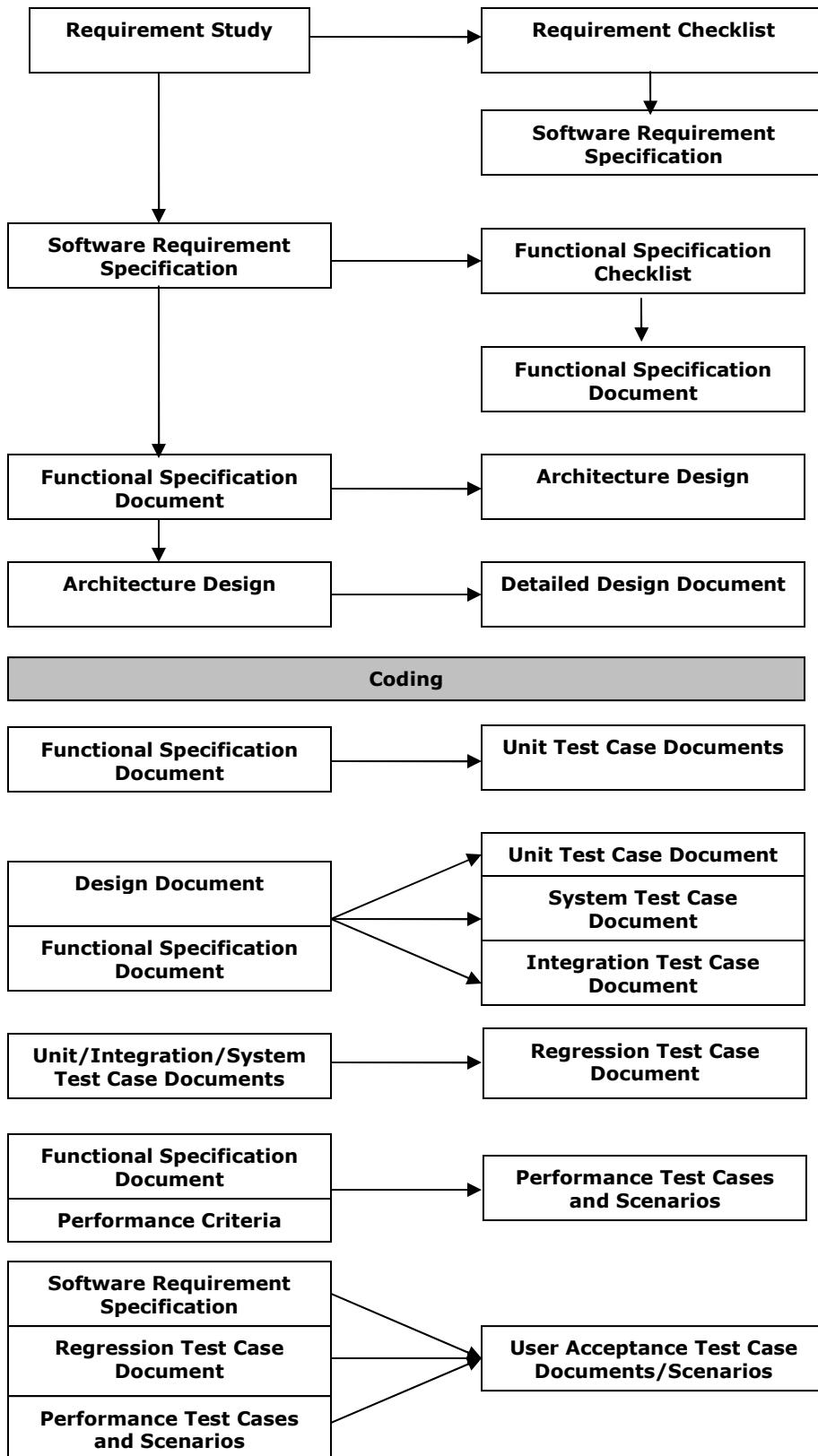
Technique	Description	Example
Stress	Determine system performance with expected volumes.	Sufficient disk space allocated.
Execution	System achieves desired level of proficiency.	Transaction turnaround time adequate.
Recovery	System can be returned to an operational status after a failure.	Evaluate adequacy of backup data.
Operations	System can be executed in a normal operational status.	Determine systems can run using document.
Compliance	System is developed in accordance with standards and procedures.	Standards follow.
Security	System is protected in accordance with importance to organization.	Access denied.

5.0 Functional System Testing Techniques

The following are the functional system testing techniques.

Technique	Description	Example
Requirements	System performs as specified.	Prove system requirements.
Regression	Verifies that anything unchanged still performs correctly.	Unchanged system segments function.
Error Handling	Errors can be prevented or detected and then corrected.	Error introduced into the test.
Manual Support	The people-computer interaction works.	Manual procedures developed.
Intersystems.	Data is correctly passed from system to system.	Intersystem parameters changed.
Control	Controls reduce system risk to an acceptable level.	File reconciliation procedures work.
Parallel	Old systems and new system are run and the results compared to detect unplanned differences.	Old and new system can reconcile.

4.0 Testing Phases



4.2 Unit Testing

Goal of Unit testing is to uncover defects using formal techniques like Boundary Value Analysis (BVA), Equivalence Partitioning, and Error Guessing. Defects and deviations in Date formats, Special requirements in input conditions (for example Text box where only numeric or alphabets should be entered), selection based on Combo Box's, List Box's, Option buttons, Check Box's would be identified during the Unit Testing phase.

4.3 Integration Testing

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

Usually, the following methods of Integration testing are followed:

1. Top-down Integration approach.
2. Bottom-up Integration approach.

4.3.1 Top-down Integration

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module. Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

1. The Integration process is performed in a series of five steps:
2. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
3. Depending on the integration approach selected subordinate stubs are replaced one at a time with actual components.
4. Tests are conducted as each component is integrated.
5. On completion of each set of tests, another stub is replaced with the real component.
6. Regression testing may be conducted to ensure that new errors have not been introduced.

4.3.2 Bottom-up Integration

Bottom-up integration testing begins construction and testing with atomic modules (i.e. components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

1. A Bottom-up integration strategy may be implemented with the following steps:
2. Low level components are combined into clusters that perform a specific software sub function.
3. A driver is written to coordinate test case input and output.
4. The cluster is tested.
5. Drivers are removed and clusters are combined moving upward in the program structure.

4.4 Smoke Testing

"Smoke testing might be characterized as a rolling integration strategy".

Smoke testing is an integration testing approach that is commonly used when “shrink-wrapped” software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis.

The smoke test should exercise the entire system from end to end. Smoke testing provides benefits such as:

- 1) Integration risk is minimized.
- 2) The quality of the end-product is improved.
- 3) Error diagnosis and correction are simplified.
- 4) Progress is easier to asses.

4.5 System Testing

System testing is a series of different tests whose primary purpose is to fully exercise the computer based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

The following tests can be categorized under System testing:

1. Recovery Testing.
2. Security Testing.
3. Stress Testing.
4. Performance Testing.

4.5.1. Recovery Testing

Recovery testing is a system test that focuses the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic, reinitialization, checkpointing mechanisms, data recovery and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

4.5.2. Security Testing

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During Security testing, password cracking, unauthorized entry into the software, network security are all taken into consideration.

4.5.3. Stress Testing

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. The following types of tests may be conducted during stress testing;

- Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
- Input data rates may be increases by an order of magnitude to determine how input functions will respond.
- Test Cases that require maximum memory or other resources.
- Test Cases that may cause excessive hunting for disk-resident data.
- Test Cases that my cause thrashing in a virtual operating system.

4.5.4. Performance Testing

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

4.5.5. Regression Testing

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side affects.

Regression may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

The Regression test suit contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

4.6 Alpha Testing

The Alpha testing is conducted at the developer sites and in a controlled environment by the end-user of the software.

4.7 User Acceptance Testing

User Acceptance testing occurs just before the software is released to the customer. The end-users along with the developers perform the User Acceptance Testing with a certain set of test cases and typical scenarios.

4.8 Beta Testing

The Beta testing is conducted at one or more customer sites by the end-user of the software. The beta test is a live application of the software in an environment that cannot be controlled by the developer.

5.0 Metrics

Metrics are the most important responsibility of the Test Team. Metrics allow for deeper understanding of the performance of the application and its behavior. The fine tuning of the application can be enhanced only with metrics. In a typical QA process, there are many metrics which provide information.

The following can be regarded as the fundamental metric:

IEEE Std 982.2 - 1988 defines a Functional or Test Coverage Metric. It can be used to measure test coverage prior to software delivery. It provides a measure of the percentage of the software tested at any point during testing.

It is calculated as follows:

Function Test Coverage = FE/FT

Where

FE is the number of test requirements that are covered by test cases that were executed against the software

FT is the total number of test requirements

Software Release Metrics

The software is ready for release when:

1. It has been tested with a test suite that provides 100% functional coverage, 80% branch coverage, and 100% procedure coverage.
2. There are no level 1 or 2 severity defects.
3. The defect finding rate is less than 40 new defects per 1000 hours of testing
4. The software reaches 1000 hours of operation
5. Stress testing, configuration testing, installation testing, Naïve user testing, usability testing, and sanity testing have been completed

IEEE Software Maturity Metric

IEEE Std 982.2 - 1988 defines a Software Maturity Index that can be used to determine the readiness for release of a software system. This index is especially useful for assessing release readiness when changes, additions, or deletions are made to existing software systems. It also provides an historical index of the impact of changes. It is calculated as follows:

$$\text{SMI} = \text{Mt} - (\text{Fa} + \text{Fc} + \text{Fd})/\text{Mt}$$

Where

SMI is the Software Maturity Index value

Mt is the number of software functions/modules in the current release

Fc is the number of functions/modules that contain changes from the previous release

Fa is the number of functions/modules that contain additions to the previous release

Fd is the number of functions/modules that are deleted from the previous release

Reliability Metrics

Perry offers the following equation for calculating reliability.

$$\text{Reliability} = 1 - \frac{\text{Number of errors (actual or predicted)}}{\text{Total number of lines of executable code}}$$

This reliability value is calculated for the number of errors during a specified time interval.

Three other metrics can be calculated during extended testing or after the system is in production. They are:

MTTFF (Mean Time to First Failure)

MTTFF = The number of time intervals the system is operable until its first failure

MTBF (Mean Time Between Failures)

MTBF = Sum of the time intervals the system is operable

Number of failures for the time period

MTTR (Mean Time To Repair)

MTTR = sum of the time intervals required to repair the system

The number of repairs during the time period

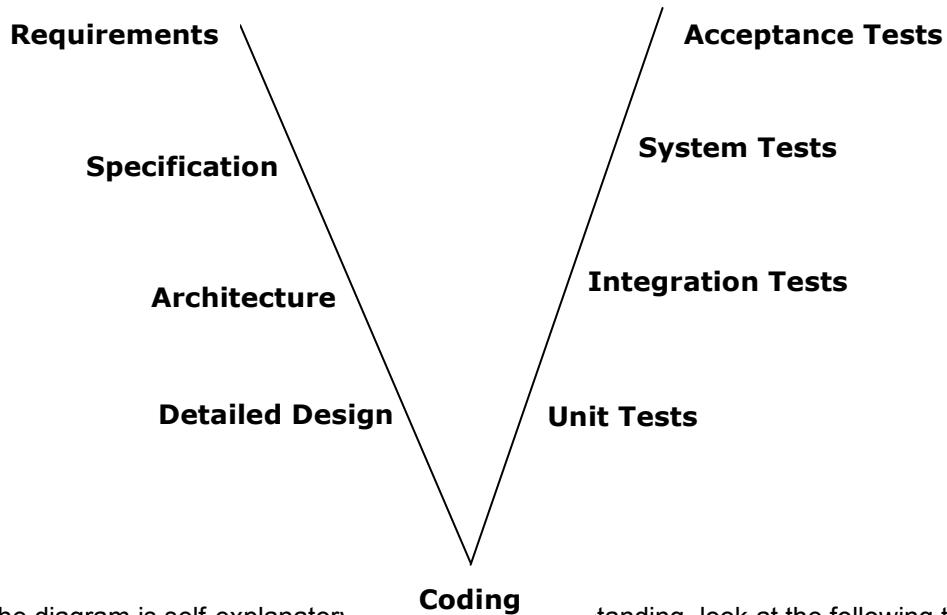
6.0 Test Models

There are various models of Software Testing. Here in this framework I would explain the three most commonly used models:

1. The 'V' Model.
2. The 'W' Model.
3. The Butterfly Model

6.1 The 'V' Model

The following diagram depicts the 'V' Model



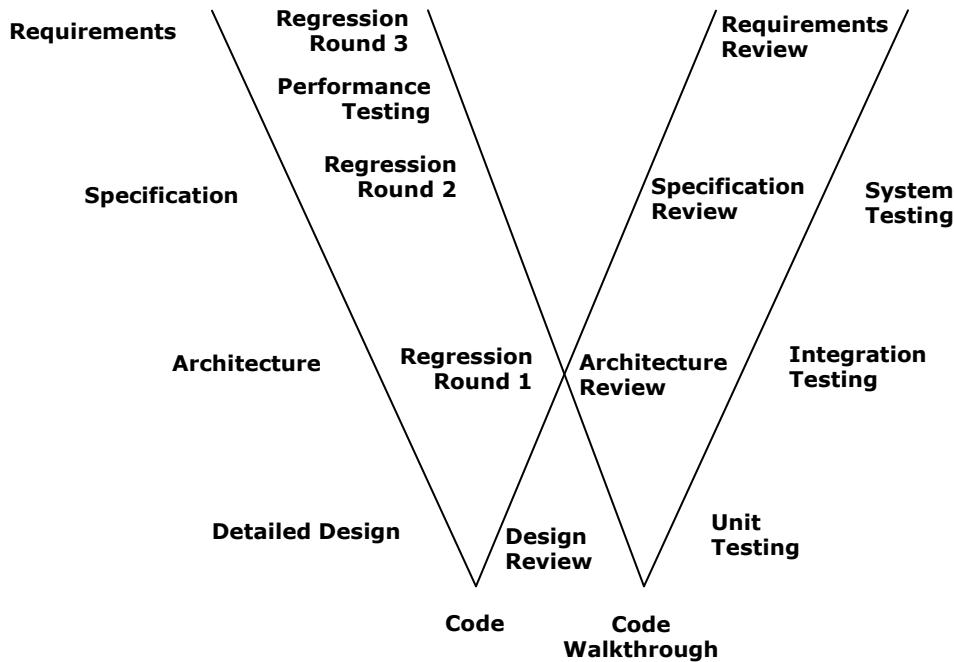
The diagram is self-explanatory

For better understanding, look at the following table:

SDLC Phase	Test Phase
1. Requirements	1. Build Test Strategy. 2. Plan for Testing. 3. Acceptance Test Scenarios Identification.
2. Specification	1. System Test Case Generation.
3. Architecture	1. Integration Test Case Generation.
4. Detailed Design	1. Unit Test Case Generation

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

In the second 'V', I have mentioned Acceptance/Beta Test Scenario Identification. This is because, the customer might want to design the Acceptance Tests. In this case as the development team executes the Beta Tests at the client place, the same team can identify the Scenarios.

Regression Rounds are performed at regular intervals to check whether the defects, which have been raised and fixed, are re-tested.

6.3 The Butterfly Model

The testing activities for testing software products are preferable to follow the **Butterfly Model**. The following picture depicts the test methodology.

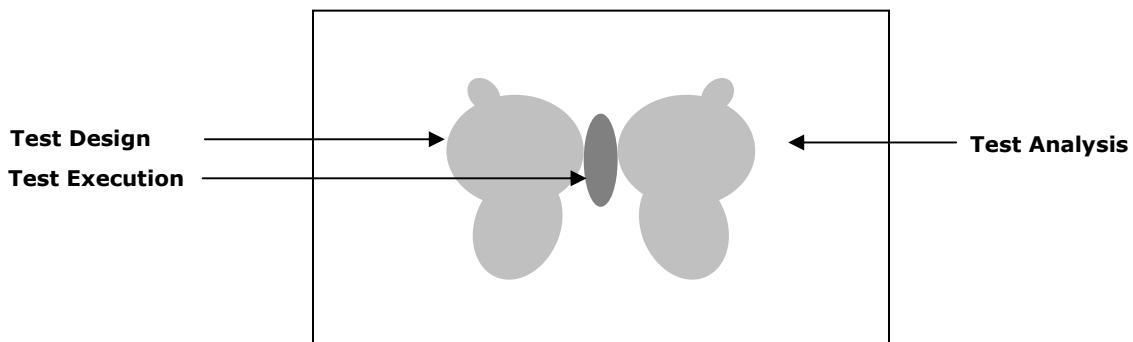


Fig: Butterfly Model

In the **Butterfly** model of Test Development, the left wing of the butterfly depicts the **Test Analysis**. The right wing depicts the **Test Design**, and finally the body of the butterfly depicts the **Test Execution**. How this exactly happens is described below.

Test Analysis

Analysis is the key factor which drives in any planning. During the analysis, the analyst understands the following:

- Verify that each requirement is tagged in a manner that allows correlation of the tests for that requirement to the requirement itself. (Establish Test Traceability)
 - Verify traceability of the software requirements to system requirements.
 - Inspect for contradictory requirements.
 - Inspect for ambiguous requirements.
 - Inspect for missing requirements.
 - Check to make sure that each requirement, as well as the specification as a whole, is understandable.
 - Identify one or more measurement, demonstration, or analysis method that may be used to verify the requirement's implementation (during formal testing).
 - Create a test "sketch" that includes the tentative approach and indicates the test's objectives.
- During Test Analysis the required documents will be carefully studied by the Test Personnel, and the final **Analysis Report** is documented.

The following documents would be usually referred:

1. Software Requirements Specification.
2. Functional Specification.
3. Architecture Document.

4. Use Case Documents.

The **Analysis Report** would consist of the understanding of the application, the functional flow of the application, number of modules involved and the effective Test Time.

Test Design

The right wing of the butterfly represents the act of designing and implementing the test cases needed to verify the design artifact as replicated in the implementation. Like test analysis, it is a relatively large piece of work. Unlike test analysis, however, the focus of test design is not to assimilate information created by others, but rather to implement procedures, techniques, and data sets that achieve the test's objective(s).

The outputs of the test analysis phase are the foundation for test design. Each requirement or design construct has had at least one technique (a measurement, demonstration, or analysis) identified during test analysis that will validate or verify that requirement. The tester must now implement the intended technique.

Software test design, as a discipline, is an exercise in the prevention, detection, and elimination of bugs in software. Preventing bugs is the primary goal of software testing. Diligent and competent test *design* prevents bugs from ever reaching the implementation stage. Test design, with its attendant test analysis foundation, is therefore the premiere weapon in the arsenal of developers and testers for limiting the cost associated with finding and fixing bugs.

During Test Design, basing on the Analysis Report the test personnel would develop the following:

1. Test Plan.
2. Test Approach.
3. Test Case documents.
4. Performance Test Parameters.
5. Performance Test Plan.

Test Execution

Any test case should adhere to the following principals:

1. Accurate – tests what the description says it will test.
2. Economical – has only the steps needed for its purpose.
3. Repeatable – tests should be consistent, no matter who/when it is executed.
4. Appropriate – should be apt for the situation.
5. Traceable – the functionality of the test case should be easily found.

During the Test Execution phase, keeping the Project and the Test schedule, the test cases designed would be executed. The following documents will be handled during the test execution phase:

1. Test Execution Reports.
2. Daily/Weekly/monthly Defect Reports.
3. Person wise defect reports.

After the Test Execution phase, the following documents would be signed off.

1. Project Closure Document.
2. Reliability Analysis Report.
3. Stability Analysis Report.
4. Performance Analysis Report.
5. Project Metrics.

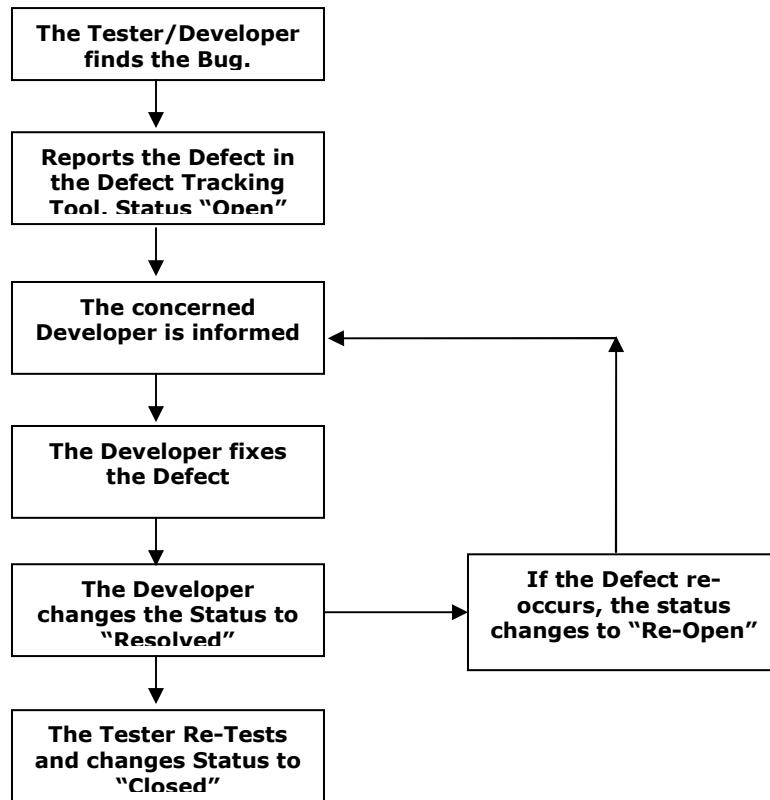
7.0 Defect Tracking Process

The Defect Tracking process should answer the following questions:

1. When is the defect found?
2. Who raised the defect?
3. Is the defect reported properly?
4. Is the defect assigned to the appropriate developer?
5. When was the defect fixed?
6. Is the defect re-tested?
7. Is the defect closed?

The defect tracking process has to be handled carefully and managed efficiently.

The following figure illustrates the defect tracking process:



Defect Classification

This section defines a defect Severity Scale framework for determining defect criticality and the associated defect Priority Levels to be assigned to errors found software.

The defects can be classified as follows:

Classification	Description
Critical	There is a functionality block. The application is not able to proceed any further.
Major	The application is not working as desired. There are variations in the functionality.
Minor	There is no failure reported due to the defect, but certainly needs to be rectified.
Cosmetic	Defects in the User Interface or Navigation.
Suggestion	Feature which can be added for betterment.

Priority Level of the Defect

The priority level describes the time for resolution of the defect. The priority level would be classified as follows:

Classification	Description
Immediate	Resolve the defect with immediate effect.
At the Earliest	Resolve the defect at the earliest, on priority at the second level.
Normal	Resolve the defect.
Later	Could be resolved at the later stages.

8.0 Test Process for a Project

In this section, I would explain how to go about planning your testing activities effectively and efficiently. The process is explained in a tabular format giving the phase of testing, activity and person responsible.

For this, I assume that the project has been identified and the testing team consists of five personnel: Test Manager, Test Lead, Senior Test Engineer and 2 Test Engineer's.

SDLC Phase	Testing Phase/Activity	Personnel
1. Requirements	1. Study the requirements for Testability. 2. Design the Test Strategy. 3. Prepare the Test Plan. 4. Identify scenarios for Acceptance/Beta Tests	Test Manager / Test Lead
2. Specification	1. Identify System Test Cases / Scenarios. 2. Identify Performance Tests.	Test Lead, Senior Test Engineer, and Test Engineers.
3. Architecture	1. Identify Integration Test Cases / Scenarios. 2. Identify Performance Tests.	Test Lead, Senior Test Engineer, and Test Engineers.
4. Detailed Design	1. Generate Unit Test Cases	Test Engineers.

9.0 Deliverables

The Deliverables from the Test team would include the following:

1. Test Strategy.
2. Test Plan.
3. Test Case Documents.
4. Defect Reports.
5. Status Reports (Daily/weekly/Monthly).
6. Test Scripts (if any).
7. Metric Reports.
8. Product Sign off Document.

UNIT TESTING FRAMEWORK

The Python unit testing framework, often referred to as ``PyUnit," is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. Each is the de facto standard unit-testing framework for its respective language.

PyUnit supports test automation, sharing of setup and shutdown code for tests aggregation of tests into collections, and independence of the tests from the reporting framework. The `unittest` module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, PyUnit supports some important concepts:

Test fixture

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

Test case

A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. PyUnit provides a base class, `TestCase`, which may be used to create new test cases.

Test suite

A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

Test runner

A test runner is a component, which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

The test case and test fixture concepts are supported through the `TestCase` and `FunctionTestCase` classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a PyUnit-driven framework. When building test fixtures using `TestCase`, the `setUp()` and `tearDown()` methods can be overridden to provide initialization and cleanup for the fixture. With `FunctionTestCase`, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the `TestCase` will only be used to run a single test method, so a new fixture is created for each test.

The `TestSuite` class implements test suites. This class allows individual tests and tests suites to be aggregated; when the suite is executed, all tests added directly to the suite and in ``child" test suites are run.

A test runner is an object that provides a single method, `run()`, which accepts a `TestCase` or `TestSuite` object as a parameter, and returns a result object. The class `TestResult` is provided for use as the result object. PyUnit provide the `TextTestRunner` as an example test runner, which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

IEEE Standard for Software Test Documentation

(ANSI/IEEE Standard 829-1983)

This is a summary of the ANSI/IEEE Standard 829-1983. It describes a test plan as:
“A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.”

This standard specifies the following test plan outline:

Test Plan Identifier:

- A unique identifier

Introduction

- Summary of the items and features to be tested
- Need for and history of each item (optional)
- References to related documents such as project authorization, project plan, QA plan, configuration management plan, relevant policies, relevant standards
- References to lower level test plans

Test Items

- Test items and their version
- Characteristics of their transmittal media
- References to related documents such as requirements specification, design specification, users guide, operations guide, installation guide
- References to bug reports related to test items
- Items which are specifically not going to be tested (optional)

Features to be Tested

- All software features and combinations of features to be tested
- References to test-design specifications associated with each feature and combination of features

Features Not to Be Tested

- All features and significant combinations of features which will not be tested
- The reasons these features won't be tested

Approach

- Overall approach to testing
- For each major group of features or combinations of features, specify the approach
- Specify major activities, techniques, and tools which are to be used to test the groups
- Specify a minimum degree of comprehensiveness required
- Identify which techniques will be used to judge comprehensiveness
- Specify any additional completion criteria
- Specify techniques which are to be used to trace requirements
- Identify significant constraints on testing, such as test-item availability, testing-resource availability, and deadline

Item Pass/Fail Criteria

- Specify the criteria to be used to determine whether each test item has passed or failed testing

Suspension Criteria and Resumption Requirements

- Specify criteria to be used to suspend the testing activity
- Specify testing activities which must be redone when testing is resumed

Test Deliverables

- Identify the deliverable documents: test plan, test design specifications, test case specifications, test procedure specifications, test item transmittal reports, test logs, test incident reports, test summary reports
- Identify test input and output data
- Identify test tools (optional)

Testing Tasks

- Identify tasks necessary to prepare for and perform testing
- Identify all task interdependencies
- Identify any special skills required

Environmental Needs

- Specify necessary and desired properties of the test environment: physical characteristics of the facilities including hardware, communications and system software, the mode of usage (i.e., stand-alone), and any other software or supplies needed
- Specify the level of security required
- Identify special test tools needed
- Identify any other testing needs
- Identify the source for all needs which are not currently available

Responsibilities

- Identify groups responsible for managing, designing, preparing, executing, witnessing, checking and resolving
- Identify groups responsible for providing the test items identified in the Test Items section
- Identify groups responsible for providing the environmental needs identified in the Environmental Needs section

Staffing and Training Needs

- Specify staffing needs by skill level
- Identify training options for providing necessary skills

Schedule

- Specify test milestones
- Specify all item transmittal events
- Estimate time required to do each testing task
- Schedule all testing tasks and test milestones
- For each testing resource, specify its periods of use

Risks and Contingencies

- Identify the high-risk assumptions of the test plan
- Specify contingency plans for each

Approvals

- Specify the names and titles of all persons who must approve the plan
- Provide space for signatures and dates

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 control. 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 information 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 builds a model of the real-world situation showing its important properties.

System Design: The system designer makes high-level decisions about the overall architecture. During system design, the target system is organized into subsystems based on both the analysis structure and the proposed architecture.

Object Design: The object designer builds a design model based on the analysis model but containing implementation details. The designer adds details to the design model in accordance with the strategy established during system design.

Implementation: The object classes and relationships developed during object design are finally translated into a particular programming language, database, or hardware implementation.

The Three Models

The OMT methodology uses three kinds of models to describe the Object Oriented System:

1. **Object Model**, describing the objects in the system and their relationships, which are static structures.
2. **Dynamic Model**, describing the interactions among objects in the system, which change over the time.
3. **Functional Model**, describing the data transformations of the system.

Object Oriented Themes

1. Abstraction.
2. Encapsulation.
3. Polymorphism.

Understanding, Designing and Testing Use Case's

Introduction

Unified Modeling Language (UML) is fast gaining popularity and is being used by many organizations. Testing projects/products using the UML notation calls for Test Engineers who are well versed with UML and its structure. This paper aims at those individuals who are into testing these UML based applications.

UML has been conceived by Ivar Jacobson (aka as Father of Use Cases), James Raumbaugh and Grady Booch. All the three are senior scientists for Rational Corporation.

In this paper I have concentrated to explain the understanding, designing and testing of Use Case's, one of the 9 UML diagrams.

The introduction section (Pages 3-5) has been taken from various books on UML which include few authored by the scientists themselves, because the concepts are important and should be understood thoroughly.

What is UML?

UML stands for Unified Modeling Language (UML), a well approached and planned language based on the Object Oriented concept.

Without going much deep into UML and its structure, I would brief here an outline of what the UML contains.

Diagrams in UML

A Diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

There are nine diagrams included in UML:

1. Class Diagram
2. Object Diagram
3. Use Case Diagram
4. Sequence Diagram
5. Collaboration Diagram
6. Statechart Diagram
7. Activity Diagram
8. Component Diagram
9. Deployment Diagram

Class Diagrams

Class Diagrams describe the static structure of a system, or how it is structured rather than how it behaves.

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.

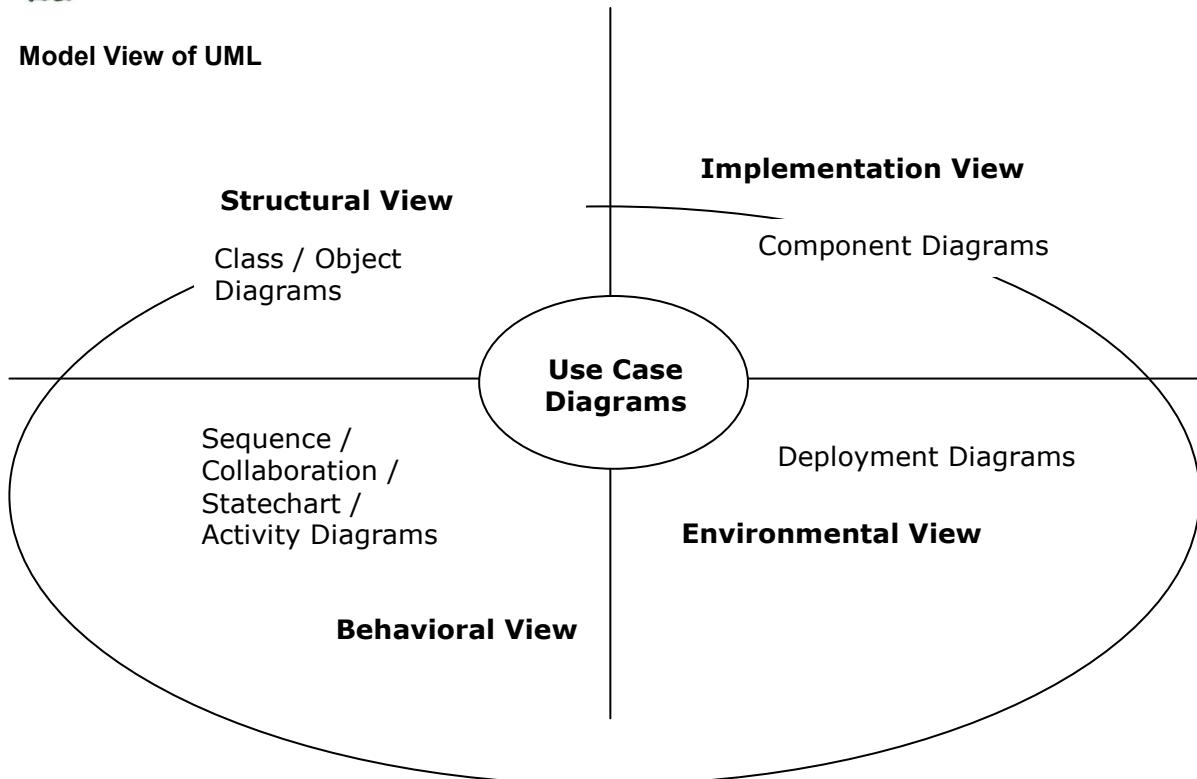
Component Diagrams

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.

Model View of UML



The *Use Case View* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts and testers. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, Statechart diagrams and activity diagrams.

The *Design View* of a system encompasses the classes, interfaces and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With UML, the static aspects of this view are captured in class diagrams and object diagrams, the dynamic aspects of this view are captured in interaction diagrams, Statechart diagrams and activity diagrams.

The *Process View* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The *Implementation View* of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, Statechart diagrams, and activity diagrams.

The *Deployment View* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery and installation of the parts that make up the physical system. With the UML, the static aspects of

this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, Statechart diagrams, and activity diagrams.

Use Case Diagrams

As we have discussed earlier, a Use Case view explains the behavior of the system to end users, analysts and testers.

Designing a Use Case

With a small example, let us see how we can design a Use Case.

Suppose that we are designing an application which has the login screen functionality for authentication. The login screen is one of the most important in an application and the design for the same should be perfect.

The requirements for the login functionality are like this:

1. The login screen should have User Name, Password and Connect to fields.
2. The Connect to field is not mandatory. This field helps the user to connect to a particular database. If the user needs, he can use this option or by default it should connect to the database which we have pre-defined to be default.
3. Login, Clear and Cancel buttons should be available.
4. Depending on the login name, the user should be taken to the Administrator page or the normal user page.

We have our requirements clear. Now we need to approach to design use case for this.

As the definition goes, a use case is used to explain the behavior of the system. The UML Specification Document, UML Notation document or the UML Schematic document does not specify that a use case should be structured in a specific way. This leads to a debate of how the use case should be structured and designed. There are many formats and ways in which a use case can be designed and used. Here, I would be depicting the usually way in which I design a use case.

A typical use case contains the following information:

Use Case Section	Description
Name	An appropriate name for the Use Case.
Brief Description	A brief description of the use case's role and purpose.
Actors	Persons who are users to the functionality.
Flow of Events	A textual representation of what the system does with regard to the use case. Usual notation for the flow of events are: Main Flow Alternative Flow Exceptional Flow
Special Requirements	A textual description that collects all requirements, such as non-functional requirements, on the use case, that are not considered in the use-case model, but that need to be taken care of during design or implementation.
Pre-Conditions	A textual description that defines any constraints on the system at the time the use case may start.

Post-Conditions	A textual description that defines any constraints on the system at the time the use case will terminate.
-----------------	---

After having seen what the use case should typically contain, let us write the use case for the login requirements what we have defined.

Use Case Name: Login

Description: This use case explains the login functionality.

Actors: Administrator, User.

Main Flow: Login

Action	Response
1. Actor invokes the application. 2. Actor enters User ID, Password and clicks on <Login> button. Actor enters wrong User ID, see alternative flow 1. Actor enters wrong Password, see alternative flow 2. Actor chooses to connect to different database, see alternative flow 3. Actor clicks on <Clear>, see alternative flow 4. Actor clicks on <Cancel>, see alternative flow 5.	1. Display login page with User ID, Password, Connect fields along with Login, Clear and Cancel buttons. 2. Authenticate and display the home page.

Alternate Flow 1: Actor enters wrong User ID

Action	Response
1. Actor enters wrong User ID and password and clicks <Login>.	1. Display error message "Invalid User ID, Please try again".

Alternate Flow 2: Actor enters wrong Password

Action	Response
1. Actor enters User ID and wrong password and clicks <Login>.	1. Display error message "Invalid Password, Please try again".

Alternate Flow 3: Actor chooses different database

Action	Response
1. Actor enters User ID, Password and chooses to connect to a different database by selecting from the list.	1. Connect to the mentioned database and display home page.

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.	<ol style="list-style-type: none"> 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	

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

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 thorough 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 extension) 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.

STRATEGY FOR DESIGNING THE TEST CASES AND TEST SCENARIO'S

The following steps typically speak of a strategy for designing the Test Case(s) and Test Scenario(s) from the Use Case document.

- Step 1:** Identify the module / class / function for which the Use Case belongs.
- Step 2:** Identify the functionality of the Use Case with respect to the overall functionality of the system.
- Step 4:** Identify the Functional Points in the Use Case and make a Functional Point Document.
- Step 3:** Identify the Actors involved in the Use Case.
- Step 4:** Identify if the Use Case "Extends" or "Uses" any other Use Case.
- Step 5:** Identify the pre-conditions.
- Step 6:** Understand the Typical Flow of the Use Case.
- Step 7:** Understand the Alternate Flow of the Use Case.
- Step 8:** Identify the Business Rule's.
- Step 9:** Check for any post-conditions and special requirements.
- Step 10:** Document Test Cases for the Typical Flow (include any actions made in the alternate flow if applicable).
- Step 11:** Document the Test Cases for Business Rules.
- Step 12:** Document Test Cases for Alternate Flow.
- Step 13:** Finally, identify the main functionality of the Use Case and document a complete positive end-to-end scenario.

Make a Cross Reference Matrix with respect to each:

1. Use Case Document and Functional Point Document.
2. Functional Point Document and Test Case Document.
3. Test Case Document and Defect Report.

These Cross-Reference Matrix Documents would be helpful for easily identifying and tracking out the defects and the functionality.

Version Control

For every Test Case Document maintain Version Control Document for tracking out the changes in the Test Case Document. The template can be made in the following way:

Version Number	Date	Comments

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.

QA personnel should not ever perform Quality Control unless it is to validate QC.

Quality Engineering Tools

The Seven Quality Engineering Tools are as follows:

- Flow Charts.
- Run Charts.
- Check Sheet.
- Histogram.
- Pareto Diagram.
- Ishikawa Diagram.
- Scatter Diagram.

The tools can be used in the following areas: Problem Representation (Flow Charts, Run Charts)

Data Collection (Check Sheet, Histogram)

Data Analysis and Synthesis (Histogram, Pareto Diagram, Ishikawa Diagram and Scatter Diagram)

The Quality Management Tools

The Seven Quality Management Tools are as follows:

- Process Decision Program Chart (PDPC).
- Activity Network.
- Tree Diagram.
- Matrix Data Analysis Diagram.
- Relationship Diagram.
- Affinity Diagram.
- Prioritization Matrix.

The tools can be used in the following areas:

Problem Representation (PDPC, Activity Network, Tree Diagram)

Data Collection (Matrix Data Analysis Diagram)

Data Analysis and Synthesis (Relationship Diagram, Tree Diagram, Affinity Diagram, Prioritization Matrix)

Recovery Tests: Recovery testing is a system test that focuses the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic, reinitialization, checkpointing mechanisms, data recovery and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Regression Tests: Regression tests address the changes in the functionality. If the application undergoes changes, tests are conducted to verify if the basic functionality is working properly and also the changes are working as intended. Regression tests become indispensable when there are continuous changes in the application.

Security Testing: Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During Security testing, password cracking, unauthorized entry into the software, network security are all taken into consideration.

Smoke Tests: "Smoke testing might be characterized as a rolling integration strategy".

Smoke testing is an integration testing approach that is commonly used when "shrink-wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. The smoke test should exercise the entire system from end to end. Smoke testing provides benefits such as:

- 1) Integration risk is minimized.
- 2) The quality of the end-product is improved.
- 3) Error diagnosis and correction are simplified.

4) Progress is easier to asses.

Stress Tests: Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

The following types of tests may be conducted during stress testing;

- Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
- Input data rates may be increases by an order of magnitude to determine how input functions will respond.
- Test Cases that require maximum memory or other resources.
- Test Cases that may cause excessive hunting for disk-resident data.
- Test Cases that my cause thrashing in a virtual operating system.

Structural Tests: Structural Tests find bugs in low-level operations such as those that occur down at the levels of lines of code, database schemas, chips, subassemblies, and interfaces.

System Tests: System tests concentrate on testing the behavior of the application. These tests are usually conducted based on the Requirements to check if the system is performing the required.

Test: An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

Test Case: A set of test inputs, execution conditions, and expected results developed for a particular objective.

Test Case Types: There are six types of test cases mentioned by Bill Hetzel, Requirements Based, Design-Based, Code-Based, Randomized, Extracted and Abnormal.

Test Procedure: The detailed instruction for the set-up, execution, and evaluation of results for a given test case.

Test Design: Test Design depicts the design of tests which need to be executed basing on the Design of the application.

Test Granularity: Test Granularity refers to the fitness or coarseness of a test's focus.

Test Plan: A Test Plan depicts the plan for conducting tests. This usually includes: Test Strategy, Schedules and Resource allocation.

Unit Tests: Testing the individual units of the software. Unit can be an input field, a screen, group of screens performing one single task, one part of the program or a complete program.

Validation Methods

Proven test-design methods provide a more intelligent and effective means of identifying tests than a purely random approach. Black-box methods for function-based tests The following methods are commonly used:

1. Equivalence Partitioning.
2. Boundary Value Analysis.
3. Error Guessing.

The following are the lesser-used methods.

1. Cause-Effect graphing.
2. Syntax Testing.
3. State Transition Testing.
4. Graph Matrix.

White Box Tests: White Box Tests aim at testing the program to the set standards rather than merely checking if the application is running as per requirement or not.



SAMPLE TESTPLAN

Company Name
Test Plan- HR & PAY ROLL SYSTEM

Revision C

Revision History

Table of Contents

1. INTRODUCTION.....	79
1.1. TEST PLAN OBJECTIVES.....	79
2. SCOPE.....	79
2.1. DATA ENTRY	79
2.2. REPORTSFILE TRANSFER.....	79
2.3. FILE TRANSFER	79
2.4. SECURITY	79
3. TEST STRATEGY.....	80
3.1. SYSTEM TEST	80
3.2. PERFORMANCE TEST	80
3.3. SECURITY TEST	80
3.4. AUTOMATED TEST.....	80
3.5. STRESS AND VOLUME TEST.....	80
3.6. RECOVERY TEST.....	80
3.7. DOCUMENTATION TEST	80
3.8. BETA TEST	80
3.9. USER ACCEPTANCE TEST.....	80
4. ENVIRONMENT REQUIREMENTS.....	81
4.1. DATA ENTRY WORKSTATIONS	81
4.2 MAINFRAME.....	81
5. TEST SCHEDULE	81
6. CONTROL PROCEDURES	81
6.1 REVIEWS	81
6.2 BUG REVIEW MEETINGS.....	81
6.3 CHANGE REQUEST	81
6.4 DEFECT REPORTING	82
7. FUNCTIONS TO BE TESTED	82
8. RESOURCES AND RESPONSIBILITIES	82
8.1. RESOURCES	82
8.2. RESPONSIBILITIES.....	83

9. DELIVERABLES.....	83
10. SUSPENSION / EXIT CRITERIA.....	85
11. RESUMPTION CRITERIA	85
12. DEPENDENCIES.....	85
12.1 PERSONNEL DEPENDENCIES	85
12.2 SOFTWARE DEPENDENCIES.....	85
12.3 HARDWARE DEPENDANCIES	85
12.3 TEST DATA & DATABASE.....	85
13. RISKS.....	85
13.1. SCHEDULE.....	85
13.2. TECHNICAL	85
13.3. MANAGEMENT.....	85
13.4. PERSONNEL.....	86
13.5 REQUIREMENTS	86
14. TOOLS	86
15. DOCUMENTATION	86
16. APPROVALS	87

Article V. 1. Introduction

The company has outgrown its current payroll system & is developing a new system that will allow for further growth and provide additional features. The software test department has been tasked with testing the new system.

The new system will do the following:

- Provide the users with menus, directions & error messages to direct him/her on the various options.
- Handle the update/addition of employee information.
- Print various reports.
- Create a payroll file and transfer the file to the mainframe.
- Run on the Banyan Vines Network using IBM compatible PCs as data entry terminals

1.1. Test Plan Objectives

This Test Plan for the new Payroll System supports the following objectives:

- Define the activities required to prepare for and conduct System, Beta and User Acceptance testing.
- Communicate to all responsible parties the System Test strategy.
- Define deliverables and responsible parties.
- Communicate to all responsible parties the various Dependencies and Risks

Article VI. 2. Scope

2.1. Data Entry

The new payroll system should allow the payroll clerks to enter employee information from IBM compatible PC workstations running DOS 3.3 or higher. The system will be menu driven and will provide error messages to help direct the clerks through various options.

2.2. Reports

The system will allow the payroll clerks to print 3 types of reports. These reports are:

- A pay period transaction report
- A pay period exception report
- A three month history report

2.3. File Transfer

Once the employee information is entered into the LAN database, the payroll system will allow the clerk to create a payroll file. This file can then be transferred, over the network, to the mainframe.

2.4. Security

Each payroll clerk will need a userid and password to login to the system. The system will require the clerks to change the password every 30 days.

3. Test Strategy

The test strategy consists of a series of different tests that will fully exercise the payroll system. The primary purpose of these tests is to uncover the systems limitations and measure its full capabilities. A list of the various planned tests and a brief explanation follows below.

3.1. System Test

The System tests will focus on the behavior of the payroll system. User scenarios will be executed against the system as well as screen mapping and error message testing.

Overall, the system tests will test the integrated system and verify that it meets the requirements defined in the requirements document.

3.2. Performance Test

Performance test will be conducted to ensure that the payroll system's response times meet the user expectations and does not exceed the specified performance criteria. During these tests, response times will be measured under heavy stress and/or volume.

3.3. Security Test

Security tests will determine how secure the new payroll system is. The tests will verify that unauthorized user access to confidential data is prevented.

3.4. Automated Test

A suite of automated tests will be developed to test the basic functionality of the payroll system and perform regression testing on areas of the systems that previously had critical/major defects. The tool will also assist us by executing user scenarios thereby emulating several users.

3.5. Stress and Volume Test

We will subject the payroll system to high input conditions and a high volume of data during the peak times. The System will be stress tested using twice (20 users) the number of expected users.

3.6. Recovery Test

Recovery tests will force the system to fail in a various ways and verify the recovery is properly performed. It is vitally important that all payroll data is recovered after a system failure & no corruption of the data occurred.

3.7. Documentation Test

Tests will be conducted to check the accuracy of the user documentation. These tests will ensure that no features are missing, and the contents can be easily understood.

3.8. Beta Test

The Payroll department will beta tests the new payroll system and will report any defects they find. This will subject the system to tests that could not be performed in our test environment.

3.9. User Acceptance Test

Once the payroll system is ready for implementation, the Payroll department will perform User Acceptance Testing. The purpose of these tests is to confirm that the system is developed according to the specified user requirements and is ready for operational use.

Article VII. 4. Environment Requirements**4.1. Data Entry workstations**

- 20 IBM compatible PCs (10 will be used by the automation tool to emulate payroll clerks).
- 286 processor (minimum)
- 4mb RAM
- 100 mb Hard Drive
- DOS 3.3 or higher
- Attached to Banyan Vines network
- A Network attached printer
- 20 user ids and passwords (10 will be used by the automation tool to emulate payroll clerks).

4.2 MainFrame

- Attached to the Banyan Vines network
- Access to a test database (to store payroll information transferred from LAN payroll system)

Article VIII. 5. Test Schedule

▪ Ramp up / System familiarization	6/01/98	-	6/15/98
▪ System Test	6/16/98	-	8/26/98
▪ Beta Test	7/28/98	-	8/18/98
▪ User Acceptance Test	8/29/98	-	9/03/98

Article IX. 6. Control Procedures**6.1 Reviews**

The project team will perform reviews for each Phase. (i.e. Requirements Review, Design Review, Code Review, Test Plan Review, Test Case Review and Final Test Summary Review). A meeting notice, with related documents, will be emailed to each participant.

6.2 Bug Review meetings

Regular weekly meeting will be held to discuss reported defects. The development department will provide status/updates on all defects reported and the test department will provide addition defect information if needed. All member of the project team will participate.

6.3 Change Request

Once testing begins, changes to the payroll system are discouraged. If functional changes are required, these proposed changes will be discussed with the Change Control Board (CCB). The CCB will determine the impact of the change and if/when it should be implemented.

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

8.2. Responsibilities

Project Manager	Responsible for Project schedules and the overall success of the project. Participate on CCB.
Lead Developer	Serve as a primary contact/liaison between the development department and the project team. Participate on CCB.
Test Lead	Ensures the overall success of the test cycles. He/she will coordinate weekly meetings and will communicate the testing status to the project team. Participate on CCB.
Testers	Responsible for performing the actual system testing.
Payroll Department Manager	Serves as Liaison between Payroll department and project team. He/she will help coordinate the Beta and User Acceptance testing efforts. Participate on CCB.
Payroll Clerks	Will assist in performing the Beta and User Acceptance testing.

9. Deliverables

Deliverable	Responsibility	Completion Date
Develop Test cases	Testers	6/11/98
Test Case Review	Test Lead, Dev. Lead, Testers	6/12/98
Develop Automated test suites	Testers	7/01/98
Requirements Validation Matrix	Test Lead	6/16/98
Obtain User ids and Passwords for payroll system/database	Test Lead	5/27/98
Execute manual and automated tests	Testers & Test Lead	8/26/98
Complete Defect Reports	Everyone testing the product	On-going
Document and communicate test status/coverage	Test Lead	Weekly
Execute Beta tests	Payroll Department Clerks	8/18/98
Document and communicate Beta test status/coverage	Payroll Department Manager	8/18/98
Execute User Acceptance tests	Payroll Department Clerks	9/03/98

Document and communicate Acceptance test status/coverage	Payroll Department Manager	9/03/98
Final Test Summary Report	Test Lead	9/05/98

Article XII. 10. Suspension / Exit Criteria

If any defects are found which seriously impact the test progress, the QA manager may choose to Suspend testing. Criteria that will justify test suspension are:

- Hardware/software is not available at the times indicated in the project schedule.
- Source code contains one or more critical defects, which seriously prevents or limits testing progress.
- Assigned test resources are not available when needed by the test team.

11. Resumption Criteria

IF TESTING IS SUSPENDED, RESUMPTION WILL ONLY OCCUR WHEN THE PROBLEM(S) THAT CAUSED THE SUSPENSION HAS BEEN RESOLVED. WHEN A CRITICAL DEFECT IS THE CAUSE OF THE SUSPENSION, THE "FIX" MUST BE VERIFIED BY THE TEST DEPARTMENT BEFORE TESTING IS RESUMED.

12. Dependencies

12.1 Personnel Dependencies

The test team requires experience testers to develop, perform and validate tests. These

The test team will also need the following resources available: Application developers and Payroll Clerks.

12.2 Software Dependencies

The source code must be unit tested and provided within the scheduled time outlined in the Project Schedule.

12.3 Hardware Dependencies

The Mainframe, 10 PCs (with specified hardware/software) as well as the LAN environment need to be available during normal working hours. Any downtime will affect the test schedule.

12.3 Test Data & Database

Test data (mock employee information) & database should also be made available to the testers for use during testing.

13. Risks

13.1. Schedule

The schedule for each phase is very aggressive and could affect testing. A slip in the schedule in one of the other phases could result in a subsequent slip in the test phase. Close project management is crucial to meeting the forecasted completion date.

13.2. Technical

Since this is a new payroll system, in the event of a failure the old system can be used. We will run our test in parallel with the production system so that there is no downtime of the current system.

13.3. Management

Management support is required so when the project falls behind, the test schedule does not

get squeezed to make up for the delay. Management can reduce the risk of delays by supporting the test team throughout the testing phase and assigning people to this project with the required skills set.

13.4. Personnel

Due to the aggressive schedule, it is very important to have experienced testers on this project. Unexpected turnovers can impact the schedule. If attrition does happen, all efforts must be made to replace the experienced individual

13.5 Requirements

The test plan and test schedule are based on the current Requirements Document. Any changes to the requirements could affect the test schedule and will need to be approved by the CCB.

Article XIII. 14. Tools

The Acme Automated test tool will be used to help test the new payroll system. We have the licensed product onsite and installed. All of the testers have been trained on the use of this test tool.

15. Documentation

The following documentation will be available at the end of the test phase:

- Test Plan
- Test Cases
- Test Case review
- Requirements Validation Matrix
- Defect reports
- Final Test Summary Report



Article XIV. 16. Approvals

Name (Print)	Signature	Date
1.		
2.		
3.		
4.		
5.		

TERMINOLOGY

Abstract Syntax Notation One (ASN.1)

The language used by the OSI protocols for describing abstract syntax. This language is also used to encode SNMP packets. ASN.1 is defined in ISO documents 8824.2 and 8825.2.

Ad hoc

Something that is ad hoc or that is done on an ad hoc basis happens or is done only when the situation makes it necessary or desirable, rather than being arranged in advance or being part of a general plan.

Ad hoc testing

Testing carried out using no recognised test case design technique.

Ad-lib test

(also ad hoc test), a test executed without prior planning; especially if the expected test outcome is not predicted beforehand. An undocumented test.

Anomaly

An anomaly is a rule or practice that is different from what is normal or usual, and which is therefore unsatisfactory.

Anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents.

Attack

An attempt to bypass security controls on a computer. The attack may alter, release, or deny data. Whether an attack will succeed depends on the vulnerability of the computer system and the effectiveness of existing countermeasures.

The act of trying to bypass security controls on a system. An attack may be active, resulting in the alteration of data; or passive, resulting in the release of data. Note: The fact that an attack is made does not necessarily mean that it will succeed. The degree of success depends on the vulnerability of the system or activity and the effectiveness of existing countermeasures.

Attack potential

The perceived potential for success of an attack, should an attack be launched, expressed in terms of an attacker's expertise, resources and motivation.

Availability

Assuring information and communications services will be ready for use when expected.

Availability of data

The state when data are in the place needed by the user, at the time the user needs them, and in the form needed by the user.

Backus-Naur Form

(also Backus normal form, BNF), a metalanguage used to formally describe the syntax of another language.

A metalanguage used to formally describe the syntax of a language.

Basic Encoding Rules (BER)

Standard rules for encoding data units described in ASN.1. Sometimes incorrectly lumped under the term ASN.1, which properly refers only to the abstract syntax description language, not the encoding technique.

Black-box testing

Functional test case design: Test case selection that is based on an analysis of the specification of the component without reference to its internal workings.

Functional testing. Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to the selected inputs and execution conditions.

Boundary value

A data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component. See also: stress testing.

An input value or output value which is on the boundary between equivalence classes, or an incremental distance either side of the boundary.

Boundary value analysis

(NBS) A selection technique in which test data are chosen to lie along "boundaries" of the input domain [or output range] classes, data structures, procedure parameters, etc. Choices often include maximum, minimum, and trivial values or parameters. This technique is often called stress testing.

A test case design technique for a component in which test cases are designed which include representatives of boundary values.

Boundary value coverage

The percentage of boundary values of the component's equivalence classes which have been exercised by a test case suite.

Boundary value testing

A testing technique using input values at, just below, and just above, the defined limits of an input domain; and with input values causing outputs to be at, just below, and just above, the defined limits of an output domain.

Branch coverage

Metric of the number of branches executed under test; "100% branch coverage" means that every branch in a program has been executed at least once under some test

Breach

The successful defeat of security controls which could result in a penetration of the system. A violation of controls of a particular information system such that information assets or system components are unduly exposed.

Brute force attack

(I) A cryptanalysis technique or other kind of attack method involving an exhaustive procedure that tries all possibilities, one-by-one.

(C) For example, for ciphertext where the analyst already knows the decryption algorithm, a brute force technique to finding the original plaintext is to decrypt the message with every possible key.

Buffer overflow

This happens when more data is put into a buffer or holding area, then the buffer can handle. This is due to a mismatch in processing rates between the producing and consuming processes. This can result in system crashes or the creation of a back door leading to system access.

Bug

A fault in a program which causes the program to perform in an unintended or unanticipated manner. See: anomaly, defect, error, exception, fault.

Certification

The comprehensive evaluation of the technical and nontechnical security features of an AIS and other safeguards, made in support of the accreditation process, that establishes the extent to which a particular design and implementation meet a specified set of security requirements.

Classification

A classification is the separation or ordering of objects (or specimens) into classes [WEBOL 1998]. Classifications that are created non-empirically are called *a priori* classifications [...; Simpson 1961; WEBOL 1998]. Classifications that are created empirically by looking at the data are called *a posteriori* classifications

Code coverage

An analysis method that determines which parts of the software have been executed (covered) by the test case suite and which parts have not been executed and therefore may require additional attention.

Component

An object of testing. An integrated assembly of one or more units and/or associated data objects or one or more components and/or associated data objects. By this (recursive) definition, a component can be anything from a unit to a system.

Compromise

An intrusion into a computer system where unauthorized disclosure, modification or destruction of sensitive information may have occurred.

A violation of the security policy of a system such that unauthorized disclosure of sensitive information may have occurred.

Confidentiality

Assuring information will be kept secret, with access limited to appropriate persons.

The concept of holding sensitive data in confidence, limited to an appropriate set of individuals or organizations.

Cost-risk analysis

The assessment of the costs of providing data protection for a system versus the cost of losing or compromising the data.

COTS Software

Commercial Off the Shelf - Software acquired by government contract through a commercial vendor. This software is a standard product, not developed by a vendor for a particular government project.

Coverage

Any metric of completeness with respect to a test selection criterion. Without qualification, usually means branch or statement coverage.

Crash

The sudden and complete failure of a computer system or component.

Debugger

One who engages in the intuitive art of correctly determining the cause (e.g., bug) of a set of symptoms.

Defect

Nonconformance to requirements.

Denial of Service

Action(s) which prevent any part of an AIS from functioning in accordance with its intended purpose.

Any action or series of actions that prevent any part of a system from functioning in accordance with its intended purpose. This includes any action that causes unauthorized destruction, modification, or delay of service. Synonymous with interdiction.

Intentional degradation or blocking of computer or network resources.

(I) The prevention of authorized access to a system resource or the delaying of system operations and functions.

Disclosure of information

Dissemination of information to anyone who is not authorized to access that information.

Dynamic analysis

The process of evaluating a system or component based on its behavior during execution.

(NBS) Analysis that is performed by executing the program code. Contrast with static analysis. See: testing.

Error

(1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. (2) An incorrect step, process, or data definition. Also: fault. (3) An incorrect result. Also: failure. (4) A human action that produces an incorrect result. Also: mistake.

(ISO) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. See: anomaly, bug, defect, exception, fault.

An error is a mistake made by a developer. It might be typographical error, a misleading of a specifications, a misunderstanding of what a subroutine does, and so on (IEEE 1990). An error might lead to one or more faults. Faults are located in the text of the program. More precisely, a fault is the difference between incorrect program and the correct version (IEEE 1990).

Error guessing

A test case design technique where the experience of the tester is used to postulate what faults might occur, and to design tests specifically to expose them.

Error seeding

The process of intentionally adding known faults to those already in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of faults remaining in the program.

Contrast with mutation analysis.

Evaluation

Evaluation is a decision about significance, value, or quality of something, based on careful study of its good and bad features.

Assessment of a PP [Protection Profile], an ST [Security Target] or a TOE [Target of Evaluation], against defined criteria.

Exception

An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception.

Exercised

A program element is exercised by a test case when the input value causes the execution of that element, such as a statement, branch, or other structural element.

Exhaustive testing

A test case design technique in which the test case suite comprises all combinations of input values and preconditions for component variables

(NBS) Executing the program with all possible combinations of values for program variables.
Feasible only for small, simple programs.

Exploit

(verb) To, in some way, take advantage of a vulnerability in a system in the pursuit or achievement of some objective. All vulnerability exploitations are attacks but not all attacks exploit vulnerabilities

(noun) Colloquially for exploit script: a script, program, mechanism, or other technique by which a vulnerability is used in the pursuit or achievement of some information assurance objective. It is common speech in this field to use the terms exploit and exploit script to refer to any mechanism, not just scripts, that uses a vulnerability.

Exploitation (of vulnerability)

The exploitation of an access control vulnerability is whatever causes the operating system to perform operations that are in conflict with the security policy as defined by the access control matrix

External IT entity

Any IT product or system, untrusted or trusted, outside of the TOE [Target of Evaluation] that interacts with the TOE.

Failure

Deviation of the software from its expected delivery or service.

The inability of a system or component to perform its required functions within specified performance requirements.

False Negative

Occurs when an actual intrusive action has occurred but the system allows it to pass as non-intrusive behavior.

False Positive

Occurs when the system classifies an action as anomalous (a possible intrusion) when it is a legitimate action.

Fault

An incorrect step, process, or data definition in a computer program.

A manifestation of an error in software. A fault, if encountered may cause a failure

An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. See: anomaly, bug, defect, error, exception

Fault injection

The hypothesized errors that software fault injection uses are created by either: (1) adding code to the code under analysis, (2) changing the code that is there, or (3) deleting code from the code under analysis. Code that is added to the program for the purpose of either simulating errors or detecting the effects of those errors is called {it instrumentation code}. To perform fault injection, some amount of instrumentation is always necessary, and although this can be added manually, it is usually performed by a tool.

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

(NBS) A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variants [mutants] of the program. Contrast with error seeding.

A method to determine test case suite thoroughness by measuring the extent to which a test case suite can discriminate the program from slight variants (mutants) of the program. See also error seeding.

Mutation testing

A testing methodology in which two or more program mutations are executed using the same test cases to evaluate the ability of the test cases to detect differences in the mutations.

Mutually suspicious

The state that exists between interacting processes (subsystems or programs) in which neither process can expect the other process to function securely with respect to some property.

Negative tests

Tests aimed at showing that software does not work (also called dirty testing); e.g., most effective tests

Network protocol stack

Software package that provides general purpose networking services to application software, independent of the particular type of data link being used.

Operational testing

Testing conducted to evaluate a system or component in its operational environment

Oracle

A mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test. [\[4\]](#) (after Adriion)

Any (often automated) means that provides information about the (correct) expected behavior of a component (HOWD86). Without qualification, this term is often used synonymously with input/output oracle.

Path coverage

Metric applied to all path-testing strategies: in a hierarchy by path length, where length is measured by the number of graph links traversed by the path or path segment; e.g. coverage with respect to path segments two links long, three links long, etc. Unqualified, this term usually means coverage with respect to the set of entry/exit paths. Often used erroneously as synonym for statement coverage.

Penetration

(I) Successful, repeatable, unauthorized access to a protected system resource.

The successful unauthorized access to an automated system

The successful act of bypassing the security mechanisms of a system.

Penetration Testing

The portion of security testing in which the evaluators attempt to circumvent the security features of a system. The evaluators may be assumed to use all system design and implementation documentation, that may include listings of system source code, manuals, and circuit diagrams. The evaluators work under the same constraints applied to ordinary users

(C) Penetration testing may be performed under various constraints and conditions. However, for a TCSEC evaluation, testers are assumed to have all system design and implementation

documentation, including source code, manuals, and circuit diagrams, and to work under no greater constraints than those applied to ordinary users.

Point of Control and Observation, PCO

A place (point) within a testing environment where the occurrence of test events is to be controlled and observed as defined by the particular abstract test method used.

Precondition

Environmental and state conditions which must be fulfilled before the component can be executed with a particular input value

Proprietary

(I) Refers to information (or other property) that is owned by an individual or organization and for which the use is restricted by that entity.

Protection profile

An implementation-independent set of security requirements for a category of TOEs [Target of Testing] that meet specific consumer needs.

Protocol

A set of conventions that govern the interaction of processes, devices, and other components within a system.

(ISO) A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication.

(I) A set of rules (i.e., formats and procedures) to implement and control some type of association (e.g., communication) between systems. (E.g., see: Internet Protocol.)

Agreed-upon methods of communications used by computers. A specification that describes the rules and procedures that products should follow to perform activities on a network, such as transmitting data. If they use the same protocols, products from different vendors should be able to communicate on the same network

A set of rules and formats, semantic and syntactic, that permits entities to exchange information

Code of correct conduct: "safety protocols"; "academic protocol"

Forms of ceremony and etiquette observed by diplomats and heads of state.

Protocol Data Unit, PDU

A PDU is a message of a given protocol comprising payload and protocol-specific control information, typically contained in a header. PDUs pass over the protocol interfaces which exist between the layers of protocols (per OSI model)

Regression testing

Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Reliability

The probability of a given system performing its mission adequately for a specified period of time under the expected operating conditions.

Software reliability is the probability that software will provide failure-free operation in a fixed environment for a fixed interval of time. Probability of failure is the probability that the software will fail on the next input selected. Software reliability is typically measured per some unit of time, whereas probability of failure is generally time independent. These two measures can be easily related if you know the frequency with which inputs are executed per unit of time. Mean-time-to-failure is the average interval of time between failures; this is also sometimes referred to as Mean-time-before-failure.

Residual risk

The portion of risk that remains after security measures have been applied

- (I) The risk that remains after countermeasures have been applied

Risk

The probability that a particular threat will exploit a particular vulnerability of the system

- (I) An expectation of loss expressed as the probability that a particular threat will exploit a particular vulnerability with a particular harmful result

Risk analysis

The process of identifying security risks, determining their magnitude, and identifying areas needing safeguards. Risk analysis is a part of risk management. Synonymous with risk assessment.

- (C) The analysis lists risks in order of cost and criticality, thereby determining where countermeasures should be applied first. It is usually financially and technically infeasible to counteract all aspects of risk, and so some residual risk will remain, even after all available countermeasures have been deployed

Risk assessment

A study of vulnerabilities, threats, likelihood, loss or impact, and theoretical effectiveness of security measures. The process of evaluating threats and vulnerabilities, known and postulated, to determine expected loss and establish the degree of acceptability to system operations.

Risk management

The total process of identifying, controlling, and eliminating or minimizing uncertain events that may affect system resources. It includes risk analysis, cost benefit analysis, selection, implementation and test, security evaluation of safeguards, and overall security review.

- (I) The process of identifying, controlling, and eliminating or minimizing uncertain events that may affect system resources

Robustness

The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Safety

(DOD) Freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment.

- (I) The property of a system being free from risk of causing harm to system entities and outside entities

Software is deemed safe if it is impossible (or at least highly unlikely) that the software could ever produce an output that would cause a catastrophic event for the system that the software controls. Examples of catastrophic events include loss of physical property, physical harm, and loss-of-life.

Safety-critical software

Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state

A condition that results from the establishment and maintenance of protective measures that ensure a state of inviolability from hostile acts or influences. hat information systems are imbued with the condition of being secure, as well as the means of establishing, testing, auditing, and otherwise maintaining that condition.

(I) (1.) Measures taken to protect a system. (2.) The condition of a system that results from the establishment and maintenance of measures to protect the system. (3.) The condition of system resources being free from unauthorized access and from unauthorized or accidental change, destruction, or loss.

Security is concerned with the protection of assets from threats, where threats are categorised as the potential for abuse of protected assets. All categories of threats should be considered; but in the domain of security greater attention is given to those threats that are related to malicious or other human activities.

Security evaluation

An evaluation done to assess the degree of trust that can be placed in systems for the secure handling of sensitive information. One type, a product evaluation, is an evaluation performed on the hardware and software features and assurances of a computer product from a perspective that excludes the application environment. The other type, a system evaluation, is done for the purpose of assessing a system's security safeguards with respect to a specific operational mission and is a major step in the certification and accreditation process. [\[8\]](#)

Security flaw

An error of commission or omission in a system that may allow protection mechanisms to be bypassed. [\[8\]](#)

Security function

A part or parts of the TOE [Target of Testing] that have to be relied upon for enforcing a closely related subset of the rules from the TSP [TOE Security Policy]. [\[9\]](#)

Security measures

Elements of software, firmware, hardware, or procedures that are included in a system for the satisfaction of security specifications. [\[8\]](#)

Security requirement

Security requirements generally include both requirements for the presence of desired behaviour and requirements for the absence of undesired behaviour. It is normally possible to demonstrate, by use or testing, the presence of the desired behaviour. It is not always possible to perform a conclusive demonstration of absence of undesired behaviour. Testing, design review, and implementation review contribute significantly to reducing the risk that such undesired behaviour is present. [\[9\]](#)

Security target

A set of security requirements and specifications to be used as the basis for evaluation of an identified TOE [Target of Testing]. [\[9\]](#)

Security testing

Testing whether the system meets its specified security objectives. [\[4\]](#)

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. ... Given enough time and resources, good security testing will ultimately penetrate a system. [\[20\]](#) (p.652)

A process used to determine that the security features of a system are implemented as designed. This includes hands-on functional testing, penetration testing, and verification. [\[8\]](#)

Silver bullet

A methodology, practice, or prescription that promises miraculous results if followed - e.g., structured programming will rid you of all bugs, as will human sacrifices to the Atlantean god Fugawe. Named either after the Lone Ranger whose silver bullets always brought justice or, alternatively, as the only known antidote to werewolves. [\[5\]](#)

Smart testing

Tests that based on theory or experience are expected to have a high probability of detecting specified classes of bugs; tests aimed at specific bug types. [\[5\]](#)

Snake oil

Derogatory term applied to a product whose developers describe it with misleading, inconsistent, or incorrect technical statements. [\[18\]](#)

Sneaker

An individual hired to break into places in order to test their security; analogous to tiger team. [\[7\]](#)

Software reliability

(IEEE) (1) the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform its required functions accurately and reproducibly under stated conditions for a specified period of time. [\[10\]](#)

Statement coverage

Metric of the number of source language statements executed under test. [\[5\]](#)

Static analysis

The process of evaluating a system or component based on its form, structure, content, or documentation. Contrast with: dynamic analysis. [\[6\]](#)

Analysis of a program carried out without executing the program. [\[4\]](#)

(NBS) Analysis of a program that is performed without executing the program. [\[10\]](#)

Stress testing

Testing in which a system is subjected to unrealistically harsh inputs or load with inadequate resources with the intention of breaking it. [\[5\]](#)

Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. See also: boundary value. [\[6\]](#)

Stress tests are designed to confront programs with abnormal situations. ... Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. ... Essentially, the tester attempts to break the program. [\[20\]](#) (p.652-653)

Structural testing

Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing. Syn: glass-box testing; white-box testing. Contrast with: functional testing (1) [\[6\]](#)

(1) (IEEE) Testing that takes into account the internal mechanism [structure] of a system or component. Types include branch testing, path testing, statement testing. (2) Testing to insure each program statement is made to execute during testing and that each program statement performs its intended function. Contrast with functional testing. Syn: white-box testing, glass-box testing, logic driven testing. [\[10\]](#)

Subtest

The smallest identifiable part of a test consisting of at least one input and one outcome. [\[5\]](#)

Symbolic execution

A software analysis technique in which program execution is simulated using symbols, such as variable names, rather than actual values for input data, and program outputs are expressed as logical or mathematical expressions involving these symbols. [\[6\]](#)

Syntax

The structural or grammatical rules that define how symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs. [\[10\]](#)

Syntax testing

A test case design technique for a component or system in which test case design is based upon the syntax of the input. [\[4\]](#)

System testing

The testing of a complete system prior to delivery. The purpose of system testing is to identify defects that will only surface when a complete system is assembled. That is, defects that cannot be attributed to individual components or the interaction between two components. System testing includes testing of performance, security, configuration sensitivity, startup and recovery from failure modes. [\[21\]](#)

System Under Test, SUT

The real open system in which the Implementation Under Test (IUT) resides. [\[17\]](#)

Target of evaluation, TOE

An IT product or system and its associated administrator and user guidance documentation that is the subject of evaluation. [\[9\]](#)

Taxonomy

A scheme that partitions a body of knowledge and defines the relationships among the pieces. It is used for classifying and understanding the body of knowledge. [\[22\]](#)

A taxonomy is the theoretical study of classification, including its bases, principles, procedures and rules [Simpson 1945; ...; WEBOL 1998]. [\[11\]](#)

Technical attack

An attack that can be perpetrated by circumventing or nullifying hardware and software protection mechanisms, rather than by subverting system personnel or other users. [\[8\]](#)

Technical vulnerability

A hardware, firmware, communication, or software flaw that leaves a computer processing system open for potential exploitation, either externally or internally, thereby resulting in risk for the owner, user, or manager of the system. [\[8\]](#)

Test

(1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded and an evaluation is made of some aspect of the system or component. (2) To conduct an activity as in (1). (3) A set of one or more test cases. (4) A set of one or more test procedures. (5) A set of one or more test cases and procedures. [\[6\]](#)

Subtests are grouped into tests, which must be run as a set, typically because the outcome of one subtest is the input or the initial condition for the next subtest in the test. Tests can be run independently of one another but are typically defined over the same database.

Test bed

An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.

Any system whose primary purpose is to provide a framework within which other systems can be tested. Test beds are usually tailored to a specific programming language and implementation technique, and often to a specific application. Typically a test bed provides some means of simulating the environment of the system under test, of test-data generation and presentation, and of recording test results

Test bed configuration

This includes many things: hardware physical configuration, platform software configuration, operating system version, sysgen details, test terminals, test tools, etc. It must be possible to precisely recreate the entire test situation

Test case

(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [do178b?]. (2) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item [\[24\]](#). See also: test case generator; test case specification. [\[6\]](#)

A document describing a single test instance in terms of input data, test procedure, test execution environment and expected outcome. Test cases also reference test objectives such as verifying compliance with a particular requirement or execution of a particular program path. [\[21\]](#)

Test case generator

A software tool that accepts as input source code, test criteria, specifications, or data structure definitions; uses these inputs to generate test input data; and, sometimes, determines expected results. Syn: test data generator, test generator. [\[6\]](#)

Test case specification

A document that specifies the test inputs, execution conditions, and predicted results for an item to be tested. Syn: test description, test specification. [\[6\]](#)

Test case suite

A collection of one or more test cases for the software under test. [\[4\]](#)

Test cycle

A formal test cycle consists of all tests performed. In software development, it can consist of, for example, the following tests: unit/component testing, integration testing, system testing, user acceptance testing and the code inspection. [\[25\]](#)

Test design

Documentation specifying the details of the test approach for a software feature or combination of software features and identifying the associated tests. [\[6\]](#) [\[24\]](#)

Test driver

A program or testing tool used to execute and control testing. Includes initialization, data object support, preparation of input values, call to tested object, recording and comparison of outcomes to required outcomes. [\[5\]](#)

A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results. Syn: test harness. [\[6\]](#)

A program or test tool used to execute software against a test case suite. [\[4\]](#)

Test environment

A description of the hardware and software environment in which the tests will be run, and any other software with which the software under test interacts when under test including stubs and test drivers. [\[4\]](#)

Test execution

The processing of a test case suite by the software under test, producing an outcome. [\[4\]](#)

Test generator

A program that generates tests in accordance to a specified strategy or heuristic. [\[5\]](#)

See: test case generator. [\[6\]](#)

Test item

A software item which is an object of testing. [\[6\]](#) [\[24\]](#)

Test log

A chronological record of all relevant details about the execution of a test. [\[6\]](#)

Test plan

A document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning. [\[6\]](#) [\[24\]](#)

A record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice. [\[4\]](#)

Test procedure

(1) Detailed instructions for the set-up, execution, and evaluation of results for a given test case. (2) A document containing a set of associated instructions as in (1). (3) Documentation specifying a sequence of actions for the execution of a test [\[24\]](#) [\[6\]](#)

(NIST) A formal document developed from a test plan that presents detailed instructions for the setup, operation, and evaluation of the results for each defined test. See: test case. [\[10\]](#)

Test report

A document that summarizes the outcome of testing in terms of items tested, summary of results (e.g. defect density), effectiveness of testing and lessons learned. [\[21\]](#)

A document that describes the conduct and results of the testing carried out for a system or component. Syn: test summary report. [\[6\]](#)

Test result analyzer

A software tool used to test output data reduction, formatting, and printing. [\[10\]](#)

Test strategy

Any method for generating tests based on formally or informally defined criteria of test completeness (also test technique). [\[5\]](#)

Test suite

A test suite is a set of related tests, usually pertaining to a group of features or software component and usually defined over the same database. Suites are combined into groups. [\[5\]](#) (p.448)

A group of tests with a common purpose and database, usually run as a group. [\[5\]](#)

Tester

One who writes and/or executes tests of software with the intention of demonstrating that the software does not work. Contrast with programmer whose tests (if any) are intended to show that the program does work. [\[5\]](#)

Testing

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. [\[26\]](#)

(1) 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.
(2) 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 items. [\[6\]](#)

Thrashing

A state in which a computer system is expending most or all of its resources on overhead operations, such as swapping data between main and auxiliary storage, rather than on intended computing functions. [\[6\]](#)

Threat

The means through which the ability or intent of a threat agent to adversely affect an automated system, facility, or operation can be manifest. A potential violation of security. [\[7\]](#)

Any circumstance or event with the potential to cause harm to a system in the form of destruction, disclosure, modification of data, and/or denial of service. [\[8\]](#)

Threat analysis

The examination of all actions and events that might adversely affect a system or operation. [\[8\]](#)

Tiger team

[U.S. military jargon] 1. Originally, a team (of sneakers) whose purpose is to penetrate security, and thus test security measures. ... Serious successes of tiger teams sometimes lead to early retirement for base commanders and security officers. 2. Recently, and more generally, any official inspection team or special firefighting group called in to look at a problem. A subset of tiger teams are professional crackers, testing the security of military computer installations by attempting remote attacks via networks or supposedly 'secure' comm channels. The term has been adopted in commercial computer-security circles in this more specific sense. [\[27\]](#)

Government and industry - sponsored teams of computer experts who attempt to break down the defenses of computer systems in an effort to uncover, and eventually patch, security holes. [\[7\]](#)

Trojan Horse

An apparently useful and innocent program containing additional hidden code which allows the unauthorized collection, exploitation, falsification, or destruction of data. [\[7\]](#)

Underflow

(ISO) The state in which a calculator shows a zero indicator for the most significant part of a number while the least significant part of the number is dropped. For example, if the calculator output capacity is four digits, the number .0000432 will be shown as .0000. See: arithmetic underflow. [\[10\]](#)

Unit

The smallest piece of software that can be independently tested (i.e., compiled or assembled, loaded, and tested). Usually the work of one programmer consisting of a few hundred lines of source code. [\[5\]](#)

Validation

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. [\[6\]](#)

(1) (FDA) Establishing documented evidence which provides a high degree of assurance that a specific process will consistently produce a product meeting its predetermined specifications and quality attributes. Contrast with data validation. [\[10\]](#)

Vendor

A person or an organization that provides software and/or hardware and/or firmware and/or documentation to the user for a fee or in exchange for services. Such a firm could be a medical device manufacturer. [\[10\]](#)

A 'vendor' is any entity that produces networking or computing technology, and is responsible for the technical content of that technology. Examples of 'technology' include hardware

(desktop computers, routers, switches, etc.), and software (operating systems, mail forwarding systems, etc.). Note that the supplier of a technology is not necessarily the 'vendor' of that technology. As an example, an Internet Service Provider (ISP) might supply routers to each of its customers, but the 'vendor' is the manufacturer, since the manufacturer, rather than the ISP, is the entity responsible for the technical content of the router. [\[28\]](#)

Vulnerability

Hardware, firmware, or software flaw that leaves an AIS open for potential exploitation. A weakness in automated system security procedures, administrative controls, physical layout, internal controls, and so forth, that could be exploited by a threat to gain unauthorized access to information or disrupt critical processing. [\[7\]](#)

A weakness in system security procedures, system design, implementation, internal controls, etc., that could be exploited to violate system security policy. [\[8\]](#)

(I) A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [\[2\]](#)

(C) Most systems have vulnerabilities of some sort, but this does not mean that the systems are too flawed to use. Not every threat results in an attack, and not every attack succeeds. Success depends on the degree of vulnerability, the strength of attacks, and the effectiveness of any countermeasures in use. If the attacks needed to exploit a vulnerability are very difficult to carry out, then the vulnerability may be tolerable. If the perceived benefit to an attacker is small, then even an easily exploited vulnerability may be tolerable. However, if the attacks are well understood and easily made, and if the vulnerable system is employed by a wide range of users, then it is likely that there will be enough benefit for someone to make an attack. [\[2\]](#)

"A state-space vulnerability is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states. If generic, the vulnerability may characterize many vulnerable states; if specific, it may characterize only one..." [Bishop and Bailey 1996] [\[11\]](#)

The Data & Computer Security Dictionary of Standards, Concepts, and Terms [Longley and Shain 1990] defines computer vulnerability as: 1) In computer security, a weakness in automated systems security procedures, administrative controls, internal controls, etc., that could be exploited by a threat to gain unauthorized access to information or to disrupt critical processing. 2) In computer security, a weakness in the physical layout, organization, procedures, personnel, management, administration, hardware or software that may be exploited to cause harm to the ADP system or activity. The presence of a vulnerability does not itself cause harm. A vulnerability is merely a condition or set of conditions that may allow the ADP system or activity to be harmed by an attack. 3) In computer security, any weakness or flaw existing in a system. The attack or harmful event, or the opportunity available to threat agent to mount that attack. [\[11\]](#)

[Amoroso 1994] defines a vulnerability as an unfortunate characteristic that allows a threat to potentially occur. A threat is any potential occurrence, malicious or otherwise, that can have an undesirable effect on these assets and resources associated with a computer system. [\[11\]](#)

...a fuzzy vulnerability is a violation of the expectations of users, administrators, and designers. Particularly when the violation of these expectations is triggered by an external object. [\[11\]](#)

Software can be vulnerable because of an error in its specification, development, or configuration. A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy. [\[11\]](#)

A feature or a combination of features of a system that allows an adversary to place the system in a state that is both contrary to the desires of the people responsible for the system and increases the risk (probability or consequence) of undesirable behavior in or of the system. A feature or a combination of features of a system that prevents the successful

implementation of a particular security policy for that system. A program with a buffer that can be overflowed with data supplied by the invoker will usually be considered a vulnerability. A telephone procedure that provides private information about the caller without prior authentication will usually be considered to have a vulnerability. [\[14\]](#)

A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [\[29\]](#)

A 'vulnerability' is a characteristic of a piece of technology which can be exploited to perpetrate a security incident. For example, if a program unintentionally allowed ordinary users to execute arbitrary operating system commands in privileged mode, this "feature" would be a vulnerability. [\[28\]](#)

Vulnerability analysis

Systematic examination of an AIS or product to determine the adequacy of security measures, identify security deficiencies, provide data from which to predict the effectiveness of proposed security measures, and confirm the adequacy of such measures after implementation. [\[7\]](#)

The systematic examination of systems in order to determine the adequacy of security measures, identify security deficiencies, and provide data from which to predict the effectiveness of proposed security measures. [\[8\]](#)

Vulnerability assessment

A measurement of vulnerability which includes the susceptibility of a particular system to a specific attack and the opportunities available to a threat agent to mount that attack. [\[8\]](#)

Vulnerability case

(missing definition)

Worm

Independent program that replicates from machine to machine across network connections often clogging networks and information systems as it spreads. [\[7\]](#)

(I) A computer program that can run independently, can propagate a complete working version of itself onto other hosts on a network, and may consume computer resources destructively. (See: Morris Worm, virus.) [\[2\]](#)

A computer program which replicates itself and is self-propagating. Worms, as opposed to viruses, are meant to spawn in network environments. Network worms were first defined by Shoch & Hupp of Xerox in ACM Communications (March 1982). The Internet worm of November 1988 is perhaps the most famous; it successfully propagated itself on over 6,000 systems across the Internet. See also: Trojan Horse, virus

TESTING GLOSSARY

Acceptance Testing: Testing conducted to enable a user/customer to determine whether to accept a software product. Normally performed to validate the software meets a set of agreed acceptance criteria.

Accessibility Testing: Verifying a product is accessible to the people having disabilities (deaf, blind, mentally disabled etc.).

Ad Hoc Testing: Similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.

Agile Testing: Testing practice for projects using agile methodologies, treating development as the customer of testing and emphasizing a test-first design paradigm.

Alpha testing: Testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.

Application Binary Interface (ABI): A specification defining requirements for portability of applications in binary forms across different system platforms and environments.

Application Programming Interface (API): A formalized set of software calls and routines that can be referenced by an application program in order to access supporting system or network services.

Automated Software Quality (ASQ): The use of software tools, such as automated testing tools, to improve software quality.

Automated Testing:

Testing employing software tools which execute tests without manual intervention. Can be applied in GUI, performance, API, etc. testing.

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.

Backus-Naur Form: A Meta language used to formally describe the syntax of a language.

Basic Block: A sequence of one or more consecutive, executable statements containing no branches.

Basis Path Testing: A white box test case design technique that uses the algorithmic flow of the program to design tests.

Basis Set: The set of tests derived using basis path testing.

Baseline: The point at which some deliverable produced during the software engineering process is put under formal change control.

Beta Testing: Testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.

Binary Portability Testing: Testing an executable application for portability across system platforms and environments, usually for conformation to an ABI specification.

Black Box Testing: Testing based on an analysis of the specification of a piece of software without reference to its internal workings. The goal is to test how well the component conforms to the published requirements for the component.

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

Boundary Testing: Test which focus on the boundary or limit conditions of the software being tested. (Some of these tests are stress tests).

Bug: A fault in a program which causes the program to perform in an unintended or unanticipated manner.

Boundary Value Analysis: BVA is similar to Equivalence Partitioning but focuses on "corner cases" or values that are usually out of range as defined by the specification. It means that if a function expects all values in range of negative 100 to positive 1000, test inputs would include negative 101 and positive 1001.

Branch Testing: Testing in which all branches in the program source code are tested at least once.

Breadth Testing: A test suite that exercises the full functionality of a product but does not test features in detail.

CAST: Computer Aided Software Testing.

Capture/Replay Tool: A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. Most commonly applied to GUI test tools.

CMM: The Capability Maturity Model for Software (CMM or SW-CMM) is a model for judging the maturity of the software processes of an organization and for identifying the key practices that are required to increase the maturity of these processes.

Cause Effect Graph: A graphical representation of inputs and the associated outputs effects which can be used to design test cases.

Code Complete: Phase of development where functionality is implemented in entirety; bug fixes are all that are left. All functions found in the Functional Specifications have been implemented.

Code Coverage: An analysis method that determines which parts of the software have been executed (covered) by the test case suite and which parts have not been executed and therefore may require additional attention.

Code Inspection: A formal testing technique where the programmer reviews source code with 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.

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 or 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.

End-to-End testing: Testing a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Equivalence Class: A portion of a component's input or output domains for which the component's behavior is assumed to be the same from the component's specification.

Equivalence Partitioning: A test case design technique for a component in which test cases are designed to execute representatives from equivalence classes.

Exhaustive Testing: Testing which covers all combinations of input values and preconditions for an element of the software under test.

Exploratory testing: Often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.

Failover Tests: Failover Tests verify of redundancy mechanisms while under load. For example, such testing determines what will happen if multiple web servers are being used under peak anticipate load, and one of them dies. Does the load balancer react quickly enough? Can the other web servers handle the sudden dumping of extra load? This sort of testing allows technicians to address problems in advance, in the comfort of a testing situation, rather than in the heat of a production outage.

Functional Decomposition: A technique used during planning, analysis and design; creates a functional hierarchy for the software.

Functional Specification: A document that describes in detail the characteristics of the product with regard to its intended features.

Functional Testing: See also Black Box Testing.

Testing the features and operational behavior of a product to ensure they correspond to its specifications.

Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

Glass Box Testing: A synonym for White Box Testing.

Gorilla Testing: Testing one particular module, functionality heavily.

Gray Box Testing: A combination of Black Box and White Box testing methodologies: testing a piece of software against its specification but using some knowledge of its internal workings.

High Order Tests: Black-box tests conducted once the software has been integrated.

Incremental Integration testing: Continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.

Independent Test Group (ITG): A group of people whose primary responsibility is software testing,

Inspection: A group review quality improvement process for written material. It consists of two aspects; product (document itself) improvement and process improvement (of both document production and inspection).

Integration Testing: Testing of combined parts of an application to determine if they function together correctly. Usually performed after unit and functional testing. This type of testing is especially relevant to client/server and distributed systems.

Installation Testing: Confirms that the application under test recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Load Testing: Load Tests are end to end performance tests under anticipated production load. The primary objective of this test is to determine the response times for various time critical transactions and business processes and that they are within documented expectations (or Service Level Agreements - SLAs). The test also measures the capability of the application to function correctly under load, by measuring transaction pass/fail/error rates.

This is a major test, requiring substantial input from the business, so that anticipated activity can be accurately simulated in a test situation. If the project has a pilot in production then logs from the pilot can be used to generate ‘usage profiles’ that can be used as part of the testing process, and can even be used to ‘drive’ large portions of the Load Test.

Load testing must be executed on “today’s” production size database, and optionally with a “projected” database. If some database tables will be much larger in some months time, then Load testing should also be conducted against a projected database. It is important that such tests are repeatable as they may need to be executed several times in the first year of wide scale deployment, to ensure that new releases and changes in database size do not push response times beyond prescribed SLAs.

Localization Testing: This term refers to making software specifically designed for a specific locality.

Loop Testing: A white box testing technique that exercises program loops.

Metric: A standard of measurement. Software metrics are the statistics describing the structure or content of a program. A metric should be a real objective measurement of something such as number of bugs per lines of code.

Monkey Testing: Testing a system or an Application on the fly, i.e. just few tests here and there to ensure the system or an application does not crash out.

Mutation testing: A method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources.

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.

Quality Audit: A systematic and independent examination to determine whether quality activities and related results comply with planned arrangements and whether these arrangements are implemented effectively and are suitable to achieve objectives.

Quality Circle: A group of individuals with related interests that meet at regular intervals to consider problems or other matters related to the quality of outputs of a process and to the correction of problems or to the improvement of quality.

Quality Control: The operational techniques and the activities used to fulfill and verify requirements of quality.

Quality Management: That aspect of the overall management function that determines and implements the quality policy.

Quality Policy: The overall intentions and direction of an organization as regards quality as formally expressed by top management.

Quality System: The organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

Race Condition: A cause of concurrency problems. Multiple accesses to a shared resource, at least one of which is a write, with no mechanism used by either to moderate simultaneous access.

Ramp Testing: Continuously raising an input signal until the system breaks down.

Recovery Testing: Confirms that the program recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Regression Testing: Retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Release Candidate: A pre-release version, which contains the desired functionality of the final version, but which needs to be tested for bugs (which ideally should be removed before the final version is released).

Sanity Testing: Brief test of major functional elements of a piece of software to determine if it's basically operational. See also Smoke Testing.

Scalability Testing: Performance testing focused on ensuring the application under test gracefully handles increases in work load.

Security Testing: Testing which confirms that the program can restrict access to authorized personnel and that the authorized personnel can access the functions available to their security level.

Smoke Testing: Typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.

Soak Testing: Soak testing is running a system at high levels of load for prolonged periods of time. A soak test would normally execute several times more transactions in an entire day (or night) than would be expected in a busy day, to identify and performance problems that appear after a large number of transactions have been executed. Also, due to memory leaks and other defects, it is possible that a system may 'stop' working after a certain number of transactions have been processed. It is important to identify such situations in a test environment.

Sociability (sensitivity) Tests: Sensitivity analysis testing can determine impact of activities in one system on another related system. Such testing involves a mathematical approach to determine the impact that one system will have on another system. For example, web enabling a customer 'order status' facility may impact on performance of telemarketing screens that interrogate the same tables in the same database. The issue of web enabling can be that it is more successful than anticipated and can result in many more enquiries than originally envisioned, which loads the IT systems with more work than had been planned.

Static Analysis: Analysis of a program carried out without executing the program.

Static Analyzer: A tool that carries out static analysis.

Static Testing: Analysis of a program carried out without executing the program.

Storage Testing: Testing that verifies the program under test stores data files in the correct directories and that it reserves sufficient space to prevent unexpected termination resulting from lack of space. This is external storage as opposed to internal storage.

Stress Testing: Stress Tests determine the load under which a system fails, and how it fails. This is in contrast to Load Testing, which attempts to simulate anticipated load. It is important to know in advance if a 'stress' situation will result in a catastrophic system failure, or if everything just "goes really slow". There are various varieties of Stress Tests, including spike, stepped and gradual ramp-up tests. Catastrophic failures require restarting various infrastructures and contribute to downtime, a stress-full environment for support staff and managers, as well as possible financial losses. This test is one of the most fundamental load and performance tests.

Structural Testing: Testing based on an analysis of internal workings and structure of a piece of software. See also White Box Testing.

System Testing: Testing that attempts to discover defects that are properties of the entire system rather than of its individual components. It's a black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.

Targeted Infrastructure Test: Targeted Infrastructure Tests are isolated tests of each layer and/or component in an end to end application configuration. It includes communications infrastructure, Load Balancers, Web Servers, Application Servers, Crypto cards, Citrix Servers, Database... allowing for identification of any performance issues that would fundamentally limit the overall ability of a system to deliver at a given performance level.

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 Plan: A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.

Test Procedure: A document providing detailed instructions for the execution of one or more test cases.

Test Script: Commonly used to refer to the instructions for a particular test that will be carried out by an automated test tool.

Test Specification: A document specifying the test approach for a software feature or combination of features and the inputs, predicted results and execution conditions for the associated tests.

Test Suite: A collection of tests used to validate the behavior of a product. The scope of a Test Suite varies from organization to organization. There may be several Test Suites for a particular product for example. In most cases however a Test Suite is a high level concept, grouping together hundreds or thousands of tests related by what they are intended to test.

Test Tools: Computer programs used in the testing of a system, a component of the system, or its documentation.

Thread Testing: A variation of top-down testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by successively lower levels.

Thick Client Application Tests: A Thick Client (also referred to as a fat client) is a purpose built piece of software that has been developed to work as a client with a server. It often has substantial business logic embedded within it, beyond the simple validation that is able to be achieved through a web browser. A thick client is often able to be very efficient with the amount of data that is transferred between it and its server, but is also often sensitive to any poor communications links. Testing tools such as Win Runner are able to be used to drive a Thick Client, so that response time can be measured under a variety of circumstances within a testing regime.

Developing a load test based on thick client activity usually requires significantly more effort for the coding stage of testing, as VUGen must be used to simulate the protocol between the client and the server. That protocol may be database connection based, COM/DCOM based, a proprietary communications protocol or even a combination of protocols.

Thin Client Application Tests: An internet browser that is used to run an application is said to be a thin client. But even thin clients can consume substantial amounts of CPU time on the computer that they are running on. This is particularly the case with complex web pages that utilize many recently introduced features to liven up a web page. Rendering a page after hitting a SUBMIT button may take several seconds even though the server may have responded to the request in less than one second. Testing tools such as WinRunner are able to be used to drive a Thin

Client, so that response time can be measured from a user's perspective, rather than from a protocol level.

Top Down Testing: An approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested.

Traceability Matrix: A document showing the relationship between Test Requirements and Test Cases.

Tuning Cycle Tests: A series of test cycles can be executed with a primary purpose of identifying tuning opportunities. Tests can be refined and re-targeted 'on the fly' to allow technology support staff to make configuration changes so that the impact of those changes can be immediately measured.

Usability Testing: Testing the ease with which users can learn and use a product.

Use Case: The specification of tests that are conducted from the end-user perspective. Use cases tend to focus on operating software as an end-user would conduct their day-to-day activities.

User Acceptance Testing: A formal product evaluation performed by a customer as a condition of purchase.

Unit Testing: The most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.

Validation: The process of evaluating software at the end of the software development process to ensure compliance with software requirements. The techniques for validation are testing, inspection and reviewing.

Verification: The process of determining whether or not the products of a given phase of the software development cycle meet the implementation steps and can be traced to the incoming objectives established during the previous phase. The techniques for verification are testing, inspection and reviewing.

Volume Testing: Testing which confirms that any values that may become large over time (such as accumulated counts, logs, and data files), can be accommodated by the program and will not cause the program to stop working or degrade its operation in any manner.

Volume Tests are often most appropriate to Messaging, Batch and Conversion processing type situations. In a Volume Test, there is often no such measure as Response time. Instead, there is usually a concept of Throughput.

A key to effective volume testing is the identification of the relevant capacity drivers. A capacity driver is something that directly impacts on the total processing capacity. For a messaging system, a capacity driver may well be the size of messages being processed. For batch processing, the type of records in the batch as well as the size of the database that the batch process interfaces with will have an impact on the number of batch records that can be processed per second.

Walkthrough: A review of requirements, designs or code characterized by the author of the material under review guiding the progression of the review.

White Box Testing: Testing based on an analysis of internal workings and structure of a piece of software. Includes techniques such as Branch Testing and Path Testing. Also known as Structural Testing and Glass Box Testing. Contrast with Black Box Testing.

Workflow Testing: Scripted end-to-end testing which duplicates specific workflows which are expected to be utilized by the end-user.

Different approaches for automated testing tools

A common type of automated tool is the 'record/playback' type. For example, a tester could click through all combinations of menu choices, dialog box choices, buttons, etc. in an application GUI and have them 'recorded' and the results logged by a tool. The 'recording' is typically in the form of text based on a scripting language that is interpretable by the testing tool. If new buttons are added, or some underlying code in the application is changed, etc. the application might then be retested by just 'playing back' the 'recorded' actions, and comparing the logging results to check effects of the changes. The problem with such tools is that if there are continual changes to the system being tested, the 'recordings' may have to be changed so much that it becomes very time-consuming to continuously update the scripts. Additionally, interpretation and analysis of results (screens, data, logs, etc.) can be a difficult task. Note that there are record/playback tools for text-based interfaces also, and for all types of platforms.

Another common type of approach for automation of functional testing is 'data-driven' or 'keyword-driven' automated testing, in which the test drivers are separated from the data and/or actions utilized in testing (an 'action' would be something like 'enter a value in a text box'). Test drivers can be in the form of automated test tools or custom-written testing software. The data and actions can be more easily maintained - such as via a spreadsheet - since they are separate from the test drivers. The test drivers 'read' the data/action information to perform specified tests. This approach can enable more efficient control, development, documentation, and maintenance of automated tests/test cases.



TEST AUTOMATION EFFORT ESTIMATION

Babu Narayanan



1. Candidates for test automation.

One of the classical mistakes of the test automation team is: 'NOT choosing right test cases for automation'.

For any smart customer, the test automation scripts are **only a support device to manual testing, NOT to bump off the later**. In that case, the customer will be more focused on the **return on investment (ROI)** for each of test automation script built (as the initial investment is high!). So choose the tangible test cases to automate against each phases of development (and demonstrate the same to the customer).

[How to find good test case candidates?](#)

<i>Sl. No</i>	<i>Test Case Complexity</i>	<i>Number of actions</i>	<i>Number of verifications</i>	<i>Good candidates (~ No of executions)</i>
1	Simple	< 5	< 5	> 15 executions
2	Medium	> 5 - < 15	> 5 - < 10	> 8 - 10 executions
3	Complex	> 15 - < 25	> 10 - < 15	> 5 - 8 executions

Please be aware that step comprises of actions and verification points (or expected results as some spell). Mostly actions are direct method or function calls to test tool scripting language but the verification points are not of that kind.

[Why do you need phase-wise test automation?](#)

Most of the test automation projects fail which is mostly due to application rapid change, unsuitable test cases, shaky frameworks and/or scripting issues. Also it summarizes that test automation projects catch fewer defects than it is supposed to do.

**** Mostly budget overrun than estimated.**

The root casual analysis showed us the necessity of phase-wise test automation than one-go test automation. So advice that test automation needs to kick off with critical test cases that are of good candidate type and then slowly branching out to other mediums as required. This solution helps the customer by lesser maintenance costs and more business for you.

Also remember that framework needs constant updates along with script development process and thereby it becomes harder to maintain the framework incase if you have many scripts to develop in parallel.

[What test type to be automated?](#)

It is always good to script 'integration and/or system' functional test cases as they mostly bundle component level test cases within them. This would reduce your effort further and find good defects (Yep! Of course, this is primary objective of any test automation project) too. Please note that if you have good framework, then you can change the test scope and/or test type using configuration files.



2. Factors that affects test automation estimation.

The following factors may have varying impact on the test automation effort calculation exercise.

Sl. No	Type	Factor	Impact	Remarks
1	Framework	Availability	High	Good framework makes your scripting, debugging and maintenance easier. Do understand that framework needs continuous updating across the script development.
2	Application	Repeat functionality	High	It is quite easier to automate incase the functionality repeats across the application (Recommend to use keyword driven in such cases, as you do not end up in writing more action/verification based methods). If NOT, then the effort of building libraries and/or scripts is more linear in nature.
3	Project	Test Scope	High	If the complexity of application as well as the test scope is complex in nature, then it would consume huge efforts to automate each test case.
4	Test tool	Support to AUT	Medium	The test tool to be used may not support some application functionality and may cause overhead. You may find it more difficult to get started with open source scripting and/or tools.
5	Scripter	Skill	Medium	This costs project. The right skill packages of the scripter are very essential for any good test automation. If the customer NOT willing to provide the leverage on the estimate for this factor, do NOT forget to add the learning curve cost to the overall time.
6	Application	Custom Objects	Medium	The number of custom objects in the automation scope matters as it becomes overhead for the test automation team to built and maintain the libraries for them.
7	Application	Type (Web/ Client-Server / Mainframe)	Medium	For web application, any commercial test tool has amazing utilities and support. Otherwise, there are good possibilities that you need to spend huge effort in building libraries.
8	Application	Developed Language & Medium	Low	It matters as selected test tool does not support specific verification checkpoints.



3. Grouping steps to determine complexity.

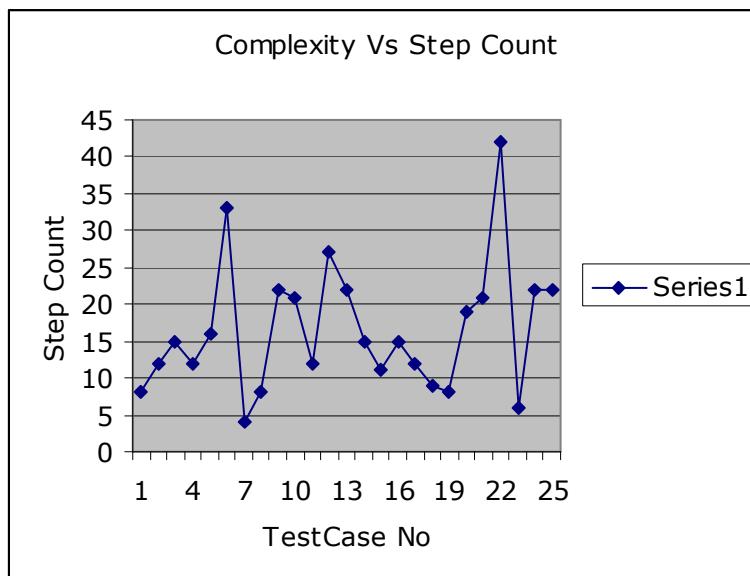
This is an important exercise as it may draw wrong overall effort in spite of depth application analysis.

STEP 1:

Suggest finding number of actions and verifications points for each test case (that are in automation scope) and then draw a chart to find the average actions, verification points and then the control limits for them. So that the complexity derivation will be based on the AUT not based on the generic industry standards.

Example,

TC	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
STEPS	8	12	15	12	16	30	4	8	22	21	12	27	22	15	11	15	12	9	8	19	21	42	6	22	22



STEP 2:

Based on the data chart,

Average step count = 16

Lower control limit = 08

Upper control limit = 25

So the complexity can be grouped as:-

Simple \leq 7 steps

Medium \geq 8 steps -- \leq 16 steps

Complex \geq 17 steps -- \leq 25 steps



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.



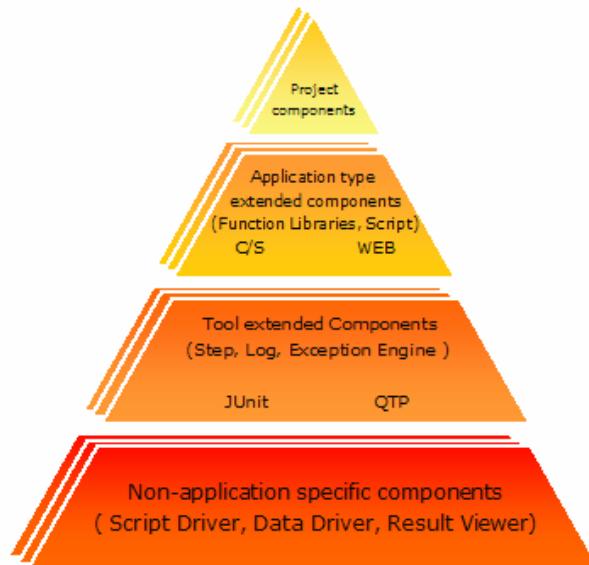
4. Framework design & estimation.

"We have experienced a significant increase in software reusability and an overall improvement in software quality due to the application programming concepts in the development and (re)use of semi finished software architectures rather than just single components and the bond between them, that is, their interaction." - **Wolfgang Pree [Pree94]**

There are many frameworks that are available commercially & as open-source which are specific to a test tool or wide-open. Also you find homebrew test automation frameworks too specific to test tools. These frameworks saves a lot of scripting, debugging and maintenance efforts but aware that the customization of framework (based on the application) are very essential.

Characteristics of any framework:

- Portable, extendable and reusable across and within projects.
- Ease of functionality Plug-ins/outs based on application version changes.
- Loosely coupled with the test tool wherever possible.
- Extended recovery system and exception handling to capture the unhandled errors and to run smoothly.
- Step, Log and Error Information provide easier debugging and customized reporting facilities of scripts.
- Ease of test data driven to the scripts and they need to be loosely coupled.
- Easily controllable and configurable test scope for every test run.
- Simple and easy integration of test automation components with test management, defect tracking and configuration management tools.



Please note that these efforts have wide range as the framework size and scope purely depends on application nature, size and complexity.

It is always a good practice to create and/or customize the framework for initial needs and then add/ update components/ features and fine tune them as project goes.

Be aware that wrong framework choice may cost your project.



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 required 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	Add error/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).

Test Cases: Assignment and Execution Strategies

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 code. The real worth of testing, as many say, is derived when test cases are written, executed, and defects logged. There are numerous techniques on writing effective test cases. However, just as significant as writing effective test cases, is who writes and executes them.

Let us consider the following scenario:

You are a test manager of a distributed test team. Both the test and development team consist of onsite and offshore members. The team follows an iterative cycle of product development. Your product development schedule is divided into phases and approximately twenty application functionalities are supposed to be developed and tested in each phase.

You need to assign the use cases (or applicable documentation) to test team members for test case creation. Once the test cases are ready, you need to decide on the most efficient test execution strategy.

This article discusses few strategies which might help in formulating an effective test case creation and execution assignments.

Test Case Creation: Assignment Strategy

There are many different factors to consider when assigning use cases to specific team members for test case creation. They might be distributed randomly or the assignments might be done according to some specific policy, as needed.

Following are some the questions which one could consider when deciding an assignment strategy for test case creation:

- a) Which use cases require significant interaction with business analysts, architects and other team members?

Assignments should consider ease of communication with the team members. This saves time and facilitates quick problem resolution in case of issues or concerns. If some use cases require frequent communication with the off-shore analyst and development teams, they should be assigned to the offshore test team members. If frequent interaction is required with onsite team members, assigning them to onsite test team members might be a good idea. This assumes greater importance for organizations with distributed team structure. In case of questions or issues, a significant amount of time might be wasted waiting for a response because of time difference between two locations and other such factors.

b) Is any use case an extension of the existing use case?

If yes, then it might be a better idea to assign it to the same team member who worked on the related use case in earlier iteration. He might be best suited to understand the dependencies and updates to the use case with respect to the earlier iteration.

c) How will the use case assignment enhance team members' knowledge of different functionalities of the application?

Use cases should be well distributed throughout the team with the aim of making team members aware of various application functionalities. They should not be assigned such that a team member is just aware of one module of application with limited knowledge about the others.

Test Case Execution: Assignment Strategy

Like test case creation, there are many different factors to consider while making test case execution assignments. They might be distributed randomly or the assignments might be done according to some specific policy.

Test execution strategy, typically, involves one of the following:

- a) Let the owners of the test cases execute the scripts written by them.
- b) Distribute the test cases among various team members so that the owners of the test cases are not assigned to execute the scripts written by them .

For example: If John wrote test cases for use case X and Peter did for use case M, then John and Peter might be assigned to execute use cases M and X respectively. John is not assigned to execute test cases X and Peter is not assigned to execute test cases for use case M.

A strategy best suited to the situation should be adopted and followed. Even a mixed strategy, a combination of two strategies above, might be used in some situations.

Strategy 1: Test Case Owners Execute Test Cases Written by Them

Below are some of the factors which favor the use of this strategy:

- a) Saves time

Team members executing the test cases know what needs to be tested as they are the owner of the test cases. This saves time from issues or concerns which might arise while executing test cases written by someone else.

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.

If a team member is aware that he will be executing the scripts written by him, it might lead to a very casual effort in documenting scripts. Even though all possible scenarios might be tested, there might not be documentation for the same, as was the case with John in above conversation.

Strategy 2: Test Case Owners Execute Test Cases Written by Other Team Members

Below are some of the factors which favor the strategy of distributing use cases among team members for test execution:

a) Error detection

People executing others scripts might find errors which might have been overlooked by the owner. This includes missing test cases, test cases based on misunderstood requirements, and incorrect test case design(e.g., incorrect test case naming conventions). It is very easy to overlook one's errors which can be easily seen by others.

b) Broadens the scope of knowledge of the application

Team members become familiar with additional application functionalities than writing and executing their own test cases.

As a coin, this strategy too has two sides. Below are some of the factors which also need to be considered together with the ones mentioned above:

a) Incomplete testing

Test case owner might have missed few scenarios while creating scripts. Since these are not covered in test scripts, they might go untested. A person executing someone else's test cases might not delve into details about missing test cases. He might assume the test cases to be complete in all aspects.

b) Waste time

If the team is distributed (onsite and offshore) and test case assignments are done such that a person executing test cases is in different location than the development and analyst team for the same use case, it might waste time in case of issues and concerns to contact the appropriate person because of time difference and other factors. The wait for a response might range from few hours to days.

c) Frustration

Understanding someone else's scripts can be quite challenging. It can become be very frustrating if the scripts are not well written and ambiguous.

For example: if during the test script execution one doesn't understand the intent of the owner, one might execute the scripts with whatever limited understanding they have. They might not raise issues and concerns because of tight deadlines and rush to finish the testing. Frustration to understand scripts might lead to poor testing.

Test case assignment and execution strategy should be an important consideration in any testing initiative. This not only ensures a smooth testing cycle but also improves the overall team efficiency. This article covered few of the considerations which might help when making such decisions.

Test Data preparation

TABLE OF CONTENTS

Test Data preparation	1
1 Introduction	3
2 Practical cases	4
3 Test case Generation from Test Data table	11
4 Summary	12

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 Practical cases

Case 1

The image shows a standard Windows-style login dialog box. The title bar reads "Enter your ID and password". Inside the box, there are two text input fields: one for "USER ID" and one for "PASSWORD". Below these fields is a checkbox labeled "Remember my ID on this computer". At the bottom right of the dialog is a large, prominent "Sign In" button.

Let us start with a simple LOGIN window assuming that the valid user id and password to be XYZ

The test data that could be generated for this are as follows:

Sl no.	User id	Password	Action Pass/Fail	Comments
1	Blank User id	Blank Password	Fail	
2	Blank User id	Valid Password	Fail	
3	Blank User id	Invalid Password	Fail	
4	Valid User id	Blank	Fail	

		Password		
5	Valid User id	Valid Password	Pass	
6	Valid User id	Invalid Password	Fail	
7	Invalid User id	Blank Password	Fail	
8	Invalid User id	Valid Password	Fail	
9	Invalid User id	Invalid Password	Fail	

From the above table it is clear that Tester ought to run the condition no. 5 to certify the functionality of login window. (Positive test case)

Of course, even other conditions are required to check the error handling capability of the application. (Negative test cases)

Let us drill down into the Invalid data

Type of data	Example	Comments		
Invalid	xyz	All Lower case letters X,Y,Z		
Invalid	123	Numerals		
Invalid	<@#%#%#\$+_>	Special characters		

Now, the expanded data table would be as follows :

Sl no.	User id	Password	Action Pass/Fail	Comments
1	Blank User id	Blank Password	Fail	
2	Blank User id	Valid Password XYZ	Fail	
3	Blank User id	Invalid Password xyz	Fail	
4	Blank User id	Invalid Password 123	Fail	
5	Blank User id	Invalid Password	Fail	

		&%^%6		
6	Valid User id XYZ	Blank Password	Fail	
7	Valid User id XYZ	Valid Password XYZ	Pass	
8	Valid User id XYZ	Invalid Password xyz	Fail	
9	Valid User id XYZ	Invalid Password 123	Fail	
10	Valid User id XYZ	Invalid Password ()^&^\$#	Fail	
11	Invalid User id xyz	Blank Password	Fail	
12	Invalid User id xyz	Valid Password XYZ	Fail	
13	Invalid User id xyz	Invalid Password xyz	Fail	
14	Invalid User id xyz	Invalid Password 123	Fail	
15	Invalid User id xyz	Invalid Password &%^%6	Fail	
16	Invalid User id 123	Blank Password	Fail	
17	Invalid User id 123	Valid Password XYZ	Fail	
18	Invalid User id 123	Invalid Password xyz	Fail	
19	Invalid User id 123	Invalid Password 123	Fail	
20	Invalid User id 123	Invalid Password &%^%6	Fail	
21	Invalid User id *&%\$	Blank Password	Fail	
22	Invalid User id	Valid Password XYZ	Fail	

	*&%\$			
23	Invalid User id *&%\$	Invalid Password xyz	Fail	
24	Invalid User id *&%\$	Invalid Password 123	Fail	
25	Invalid User id *&%\$	Invalid Password 0^&^&\$#	Fail	

Thus, one would arrive at more data combinations (25) rather than generic 9 combinations. This data could be passed onto automation team for Data Driven Test implementation.

Case 2



Let us assume the following :

1. The “Flights” button would function only if the values are selected from the dropdown lists.
2. “Fly From” dropdown list would have 3 values for instance Frankfurt, Colombo, Austin and a blank value.
3. “Fly To” dropdown list would have 3 values for instance Paris, London, Delhi and a blank value.

The test data that could be generated for this are as follows:

Sl no.	Fly From	Fly To	Action Pass/Fail	Comments

1	Not Selected	Not Selected	Fail	
2	Not Selected	Selected Paris	Fail	
3	Not Selected	Selected London	Fail	
4	Not Selected	Selected Delhi	Fail	
5	Selected Frankfurt	Not Selected	Fail	
6	Selected Frankfurt	Selected Paris	Pass	
7	Selected Frankfurt	Selected London	Pass	
8	Selected Frankfurt	Selected Delhi	Pass	
9	Selected Colombo	Not Selected	Fail	
10	Selected Colombo	Selected Paris	Fail	
11	Selected Colombo	Selected London	Pass	
12	Selected Colombo	Selected Delhi	Pass	
13	Selected Austin	Not Selected	Fail	
14	Selected Austin	Selected Paris	Pass	
15	Selected Austin	Selected London	Pass	
16	Selected Austin	Selected Delhi	Pass	

Case 3

Name:	<input type="text" value="krish"/>	Tickets:	<input type="text" value="99"/>
Class:	<input type="radio"/> First <input type="radio"/> Business <input checked="" type="radio"/> Economy	Price:	<input type="text" value="\$112.20"/>
		Total:	<input type="text" value="\$11107.80"/>
<input type="button" value="Update Order"/> <input type="button" value="Delete Order"/>		<input type="button" value="Insert Order"/>	

Let us assume the following :

1. Name is populated with the default value as “krish”.

2. “Insert Order” button would function only if the “Tickets” edit field is filled with a number and any one of the “Class” radio button is selected.

3. “Tickets” field would accept any value ranging from 1-99 only.

The test data that could be generated for this are as follows:

Sl no.	Tickets	Class	Action Pass/Fail	Comments
1	Blank	First	Fail	
2	Blank	Business	Fail	
3	Blank	Economy	Fail	
4	1	First	Pass	BVA
5	1	Business	Pass	BVA
6	1	Economy	Pass	BVA
7	2	First	Pass	BVA
8	2	Business	Pass	BVA
9	2	Economy	Pass	BVA
10	0	First	Fail	BVA
11	0	Business	Fail	BVA
12	0	Economy	Pass	BVA
13	98	First	Pass	BVA
14	98	Business	Pass	BVA
15	98	Economy	Pass	BVA, valid Equivalence class
16	99	First	Pass	BVA
17	99	Business	Pass	BVA
18	99	Economy	Pass	BVA
19	100	First	Fail	BVA
20	100	Business	Fail	BVA
21	100	Economy	Fail	BVA

22	two	First	Fail	
23	two	Business	Fail	
24	two	Economy	Fail	Invalid Equivalence class
25	@#@#	First	Fail	
26	@#@#	Business	Fail	
27	@#@#	Economy	Fail	Invalid Equivalence class

3 Test case Generation from Test Data table

	A	B	C	D	E	F	G	H	I
Sl no	Ticket	action1	Formula used to get action1	Class	action2	Formula used to get action2	Test Step	Expected Result	
1	1 Blank	Blank out Tickets	=B2&" out "&B1	First	Select First from Class	="Select &E2&" Erom "&E1	Blank out Tickets, Select First from Class and click insert order button	Error message should appear	
2	2 Blank	Blank out Tickets	=B3&" out "&B1	Business	Select Business from Class	="Select &E3&" Erom "&E1	Blank out Tickets, Select Business from Class and click insert order button	Error message should appear	
3	3 Blank			Economy					Fail
4	4	1 Enter 1 in Tickets	= "Enter "&B5&" in "&B1	First	Select First from Class	="Select &E3&" Erom "&E1	Enter 1 in Tickets, Select First from Class and click insert order button	Navigation should proceed	
5	5	1	BASIC DATA FROM TEST DATA TABLE	Business			READY TESTCASE	Pass	

One needs to use the formulas to concatenate field values and then copy/paste them to different cells wherever applicable. This process of writing test cases is much faster than the traditional one, which involves typing each and every action with the values to arrive at the expected result.

For example:

- Action1=**B2&" out "&B1** would mean concatenate “B2 contents i.e., blank” with “ out ” and “B1 contents i.e., Tickets”, finally resulting in as “Blank out Tickets”.

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., $3^2 = 9$

Now, the Final formula covering overall combinations would be = F1*F2 i.e., 243*9
= 2187)

✓✓M

TESTING WITHOUT REQUIREMENTS SPECIFICATIONS



TABLE OF CONTENTS

TEST CASES: INTRODUCTION, CREATING AND IMPORTANCE	3
TEST CASES WITHOUT REQUIREMENTS SPECIFICATIONS	3
THEORETICAL APPROACH.....	4
USE CASE BASED APPROACH.....	4
BUSINESS RULES TESTING	5
WALKTHROUGH MEETING WITH THE TECHNICAL SPECIFICATIONS DOCUMENT.....	6
DIAGRAMMATIC APPROACH	6
ER DIAGRAM APPROACH	7
ACTIVITY MODEL DIAGRAM /UML.....	9
DFD DIAGRAM-	11
RISKS, ISSUES AND CONTINGENCY PLANNING	12
ALTERNATIVE WAYS OF TESTING.....	13
USER/ACCEPTANCE TESTING	13
RANDOM TESTING	13
CUSTOMER STORIES	13
CONCLUSION.....	ERROR! BOOKMARK NOT DEFINED.
AUTHOR'S PROFILE	ERROR! BOOKMARK NOT DEFINED.
REFERENCES.....	ERROR! BOOKMARK NOT DEFINED.



Test Cases: Introduction, Creating and Importance

Test cases have traditionally been used to test any system - software or otherwise. The test case may transform into a checklist, a comprehensive step by step guideline on information displayed by the system, or a simple black box scenario. In any case, a test case is a pure representation of what the system should and shouldn't do.

A functional specification document on the other hand is documented the way the user would want the system to be or how he perceives the system to be. A typical functional system is dominated by user understandable language, screen shots and behavior. Business rules may also be documented as part of the document.

One can argue that it may be easier to test from a functional/requirements specification document than creating a separate test case document - since the basis of a test case document is the functional specification document.

However there are several known pitfalls with this approach. Some of them are -

1. Test scenario build up is difficult
2. Incomplete documentation of flows
3. Unit level approach rather than system level approach.
4. Business rules may not be apparent, though they are documented.

Typically a functional specification is the basis of developing test cases with user flows, business rules and special conditions thrown in.

Test cases without requirements specifications

On the fly development, pressurized deadlines, changing user requirements, legacy systems, large systems, varied requirements and many more are known influences for many a project manager to discard or not update the functional specification document. While a project manager/requirements manager may be aware of the system requirements, this can cause havoc for an independent test team.

More so, with ever expanding teams, temporary resources, a large system may not be documented well enough or sometimes not at all, for the tester to form complete scenarios from the functional specification (if present) document.

This can pose a great risk to the testing of the system. Requirements will never be clearly understood by the test team and the developers will adopt an 'I am king' attitude, citing incorrectly working functionality as the way the code or design is supposed to work. This can confuse end development and you may have a product on your hand that the business doesn't relate to, the end users don't understand and the testers can't test.



Dealing with a system that needs to be tested without functional specifications requires good planning, smart understanding and good documentation - lack of which caused the problem in the first place.

Before I begin describing some ways in which you can tackle the problem - here are a few questions you could be asking yourself

1. What type of testing will be done on the system- unit / functional / performance / regression?
2. Is it a maintenance project or is it a new product?
3. From where have the developers achieved their understanding of the system?
4. Is there any kind of documentation other than functional specification?
5. Is there a business analyst or domain knowledge expert available?
6. Is an end user available?
7. Is there a priority on any part of the functionality of the system?
8. Are there known high-risk areas?
9. What is the level of expertise of the development team and the testing team?
10. Are there any domain experts within your test team?
11. Are there any known skill sets in your team?
12. Are you considering manual/automated testing?

All of the questions should help you identify the key areas for testing, the ignorable areas for testing, the strong and weak areas of your team. Once you ascertain the level of skills and expertise in your team, you have choices available to choose from. Depending on the comfort level in your team, you can choose one from the options given below.

Theoretical Approach

The theoretical approach will ensure that you have good comprehensive documents for the system at the end of the process.

Use Case Based Approach

Traditionally use cases have been used for requirement gathering. There has been a considerable shift to use them for testing, primarily because they model user role-play, movement, access rights and the interaction of one module to another - which is essentially what a test plan aims to do.

A use case defines the following -

1. Primary actor or actors
2. Goal
3. Scenarios used
4. Conditions under which a scenario occurs
5. Scenario result (goal delivery or failure)
6. Alternative scenarios or flows

The advantage of this approach is that you have two ready things in your hand

1. Documented system functionality
2. Evident module interaction

The development or creation of the use case will not be an easy task. Typical way to develop a use case is to pick up the system screen by screen or module wise and talk to the concerned developers or business analysts. Users can also be brought into the discussion at a later stage. Most users will give you valuable inputs to the type of data they input to the system. That in turn can give you an insight to the flows of the system.



Exemplary questions that you can ask users/developers are:

1. Inputs to a screen
2. User movements/warnings on a screen
3. Access rights to screen(s) or modules.
4. Access to screen(s) and exits from screen(s)
5. Database checks/updates.
6. Outputs from a screen
7. Alternative inputs or exits from the screen(s).

You can screen these questions or add some based on the testing you will be conducting for the system. For e.g. If you are carrying out GUI testing, you could do well if you add questions like length of the text box (this question could be answered well by a developer, rather than a user or a business analyst). If you are doing system testing, you could probably cut out the question on database check or updates. On the other hand, if you are doing system testing as part of a technology migration, you should enforce that database updates are verified.

Business Rules testing

Every system operates on business rules. If one were to document the business rules on which a system is based - it would be an exhaustive list to write down and track. Most business rules become implicit in the design and mind.

Business rules are typically implemented separately from presentation, application and data logic. To add to this, they are under constant change due to market requirements, user needs and software upgrades. Business Rules testing will help validate that your system is performing as expected, thus enforcing correct business practices on the user and the system code.

A typical way to gather business rules for your system is by talking to the business analyst and trying to understand what the analyst expects the system to do or how he/she expects the system to behave. A meeting with a developer will be helpful to understand how he/she thinks the system should behave. You may get conflicting ideas here, and it is important that these are sorted out before you have begun testing.

You could document your learning using a spreadsheet designed as below. I have given some examples to help you understand what the header means.

Screen/Module Name	Component	Business Rule	Affects Flow - if yes, detail how?
XX1	<i>Text Box</i>	<i>Min Char Length</i>	No
XX1	<i>User Rights</i>	<i>Admin</i>	Yes - new menu option
XX1	<i>XX Path selected</i>	<i>Available only if Option A is selected</i>	Yes - takes user to Screen YY

The important point is to document all the business rules and at the same time understand the co-relation it has with various functional scenarios of the system.

You can expand the table above to suit your needs in order to understand the various normal and alternative flows of the system. Again, mould the table and the data that you gather to suit the type of testing that you will do for your system. If you are doing unit testing, you will use more business rules like the first example (min char length), than like the third example (available only if Option A..)

Keep the following in mind:



1. Divide the system into modules as done by development.
2. For each module, take views based on the kind of testing you intend to do.
3. Direct your questions to the right people. E.g. If you are unit testing a product, a developer would be the right person to talk to. If you are system or user testing a product, a business analyst or end user would be able to answer your questions.
4. If you are short on time, document your business rules in order of decreasing priority.
5. Be specific with your questions. Run through the product or the system yourself, and make a list of questions, before you attempt to ask other team members.
6. Know the difference between a business rule and UI design. For e.g. a warning message to indicate that all information may be deleted is not a business rule, but a UI design.
7. Check if the business rules are incorporated in the system as code logic or are stored in the database and are made use of through a rules engine. This will help you organize your questions better.

Walkthrough meeting with the Technical Specifications document

In a typical walkthrough, the author presents a code module or design component to the participants, describing what it does, how it is structured and performs its tasks, the logic flow, and its inputs and outputs.

This can be modulated to suit the needs of developing test cases. You can call for a walkthrough meeting with the business analyst, the head developer or the solution architect and request for a presentation of the system. If a technical specification (TS) document exists, it will do you good to do the walkthrough with the TS. The TS document will focus on the technicalities of the system, while the walkthrough will help you understand the front end of the system.

Document your system in the following areas at the time of walkthrough with the TS -

1. Working/Functionality of each screen
2. Special conditions/functionalities of the screen
3. Backend updates
4. Inputs
5. Outputs
6. Expected scenarios and results
7. Importance of the screen and its scenarios.
8. User interaction and dependency

Again, drill down to the level that your testing requires of you.

DO NOT do the following

1. Convert the walkthrough to a problem solving session.
2. Try to find faults in the design, UI, system flow or layout.
3. Try to find faults in the DB design or architecture
4. Leave doubts till later.
5. Ignore glaring or critical faults. Point them out, but don't attempt to find a solution.

Diagrammatic Approach

Data models are tools used in analysis to describe the data requirements and assumptions in the system from a top-down perspective. Traditionally, data models set the stage for the design of databases later on in the SDLC.



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:

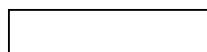
1. **Entities** are the "things" about which we seek information.
2. **Attributes** are the data we collect about the entities.
3. **Relationships** provide the structure needed to draw information from multiple entities.

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

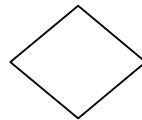
1. What/Who are the entities?
2. Is there more than one entity? Is there any relationship between these entities?
3. Does the entity have relationships other than the existing entities on the screen?
4. What are the qualities of the entities?
5. Cardinality of the entities (many-to-one, one-to-many)
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.)

Update Repair Type

Repair Type:	Insurance
New Repair Type:	<input type="button" value="Select..."/>
Change Reason:	<input type="button" value="Select..."/>
Engineer Notes:	<input type="text"/>

Cancel **Save**



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

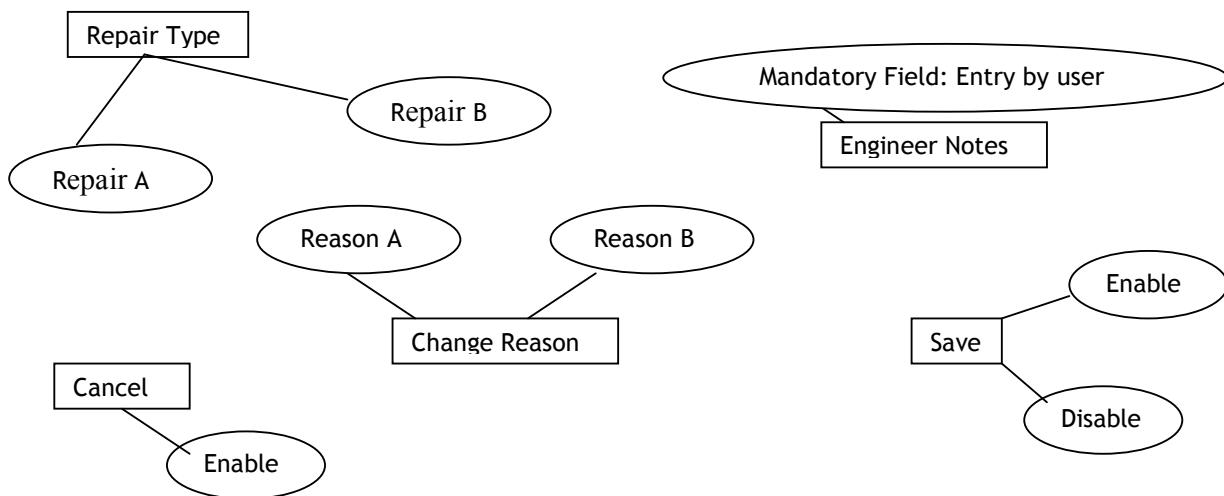
Change Reason - Change Reason A, Change Reason B

Engineer Notes- Entry mandatory by user.

Cancel- Enabled

Save- Enabled and Disabled

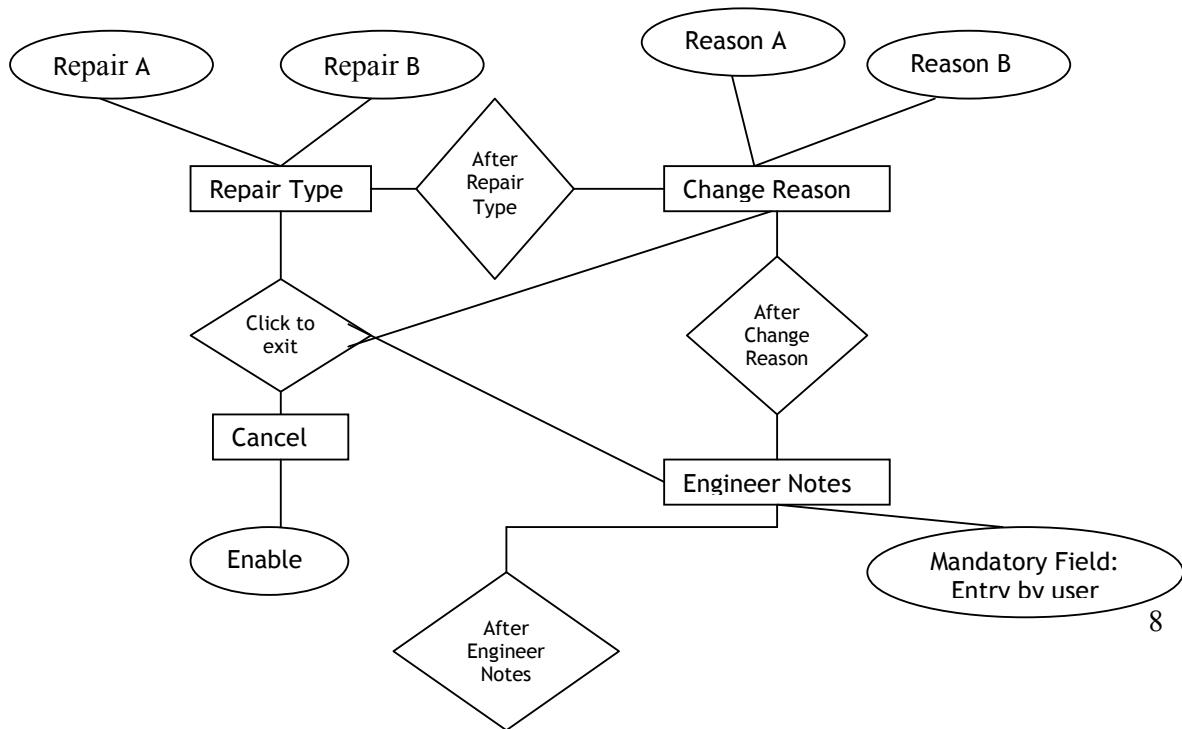
Based on the input I gathered so far, I made the following ER Diagram



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 usages 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.



Deployment diagrams show the physical system's runtime architecture. A deployment diagram can include a description of the hardware and the software that resides on that hardware.

Composite structure diagrams show the internal structure of model elements.

Package diagrams depict the dependencies between packages. (A package is a model element used to group together other model elements.)

Behavior diagrams depict the dynamic behavior of the elements in your system. The various behavior diagrams are as follows:

Activity diagrams show the flow of activities within a system. You often would use them to describe different business processes.

Use case diagrams address the business processes that the system will implement. The use cases describe ways the system will work and who will interact with it. [BOOCH1]

State chart diagrams show the state of an object and how that object transitions between states. A state chart diagram can contain states, transitions, events, and activities. A state chart diagram provides a dynamic view and is quite important when modeling event-driven behavior. For example, you could use a state chart diagram to describe a switch in a telephone routing system. That switch will change states based on different events, and you can model those events in a state chart diagram to understand how the switch behaves.

Collaboration diagrams are a type of interaction diagram, as are sequence diagrams. The collaboration diagram emphasizes how objects collaborate and interact.

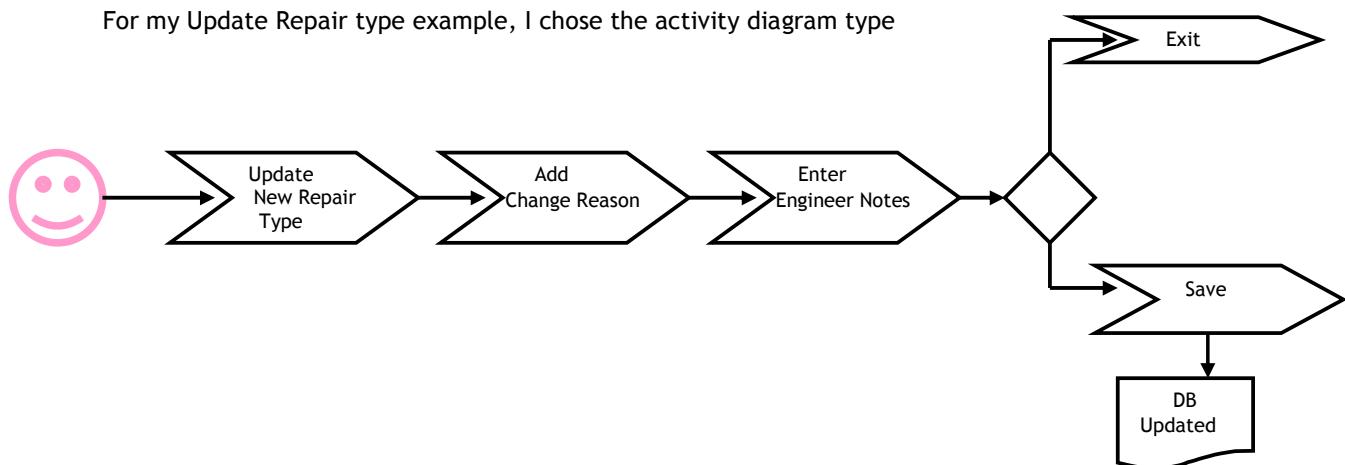
Sequence diagrams are another type of interaction diagram. Sequence diagrams emphasize the time ordering of messages between different elements of a system.

Timing diagrams are another type of interaction diagram. They depict detailed timing information and changes to state or condition information of the interacting elements.

Interaction overview diagrams are high-level diagrams used to show an overview of flow of control between interaction sequences.

As you can see, UML can support almost any kind of need that you might have of depicting a system. You need to decide which one you and your team would find easier to understand and draw test cases from.

For my Update Repair type example, I chose the activity diagram type





As you can see the diagram is immediately readable to the tester and clearly shows two flows in the system. Again, depending on the type of testing you are planning to do, add or delete detail from the diagram.

More importantly, determine if your system demands a structural or behavioral UML to detail it correctly.

DFD Diagram-

Data flow diagrams can be used to provide a clear representation of any business function. They depict the movement and process steps of data and information. The DFD diagram is an ideal representation of a black box kind of scenario. A DFD diagram can be used for analysis of the system at a high level and can then be used to describe the system to the lowest level of detail or as required. Hence, you can have a clear top-down expansion of the system functionality.

A DFD uses the following to depict a business process diagram -

1. External Entity

An external entity is a source or destination of a data flow, which is outside the area of study. Entities, which receive or originate data, are represented in the DFD. Symbol is an oval, which contains a meaningful, unique identifier.



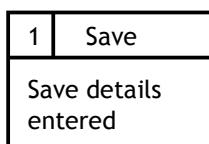
2. Process

A process represents a transformation or manipulation of data flows within a system.

The symbol used is a rectangular box, which contains 3 descriptive elements:

Firstly an identification number appears in the upper left hand corner. This is allocated arbitrarily at the top level and serves as a unique reference. **Secondly**, a location appears to the right of the identifier and describes where in the system the process takes place.

Finally, a descriptive title is placed in the centre of the box. This should be a simple imperative sentence with a specific verb, for example 'maintain customer records' or 'find driver'.



3. Data Flow

A data flow shows the flow of information from its source to its destination. A line represents a data flow, with arrowheads showing the direction of flow. Information always flows to or from a process and may be written, verbal or electronic. Each data flow may be referenced by the processes or data stores at its head and tail, or by a description of its contents.

Select Repair Type
→

4. Data Store

A data store is a holding place for information within the system. It is represented by an open ended narrow rectangle. Data stores may be long-term files such as sales ledgers, or may be short-term accumulations: for example batches of documents that are waiting to be processed. Each data store should be given a reference followed by an arbitrary number.

✓✓M

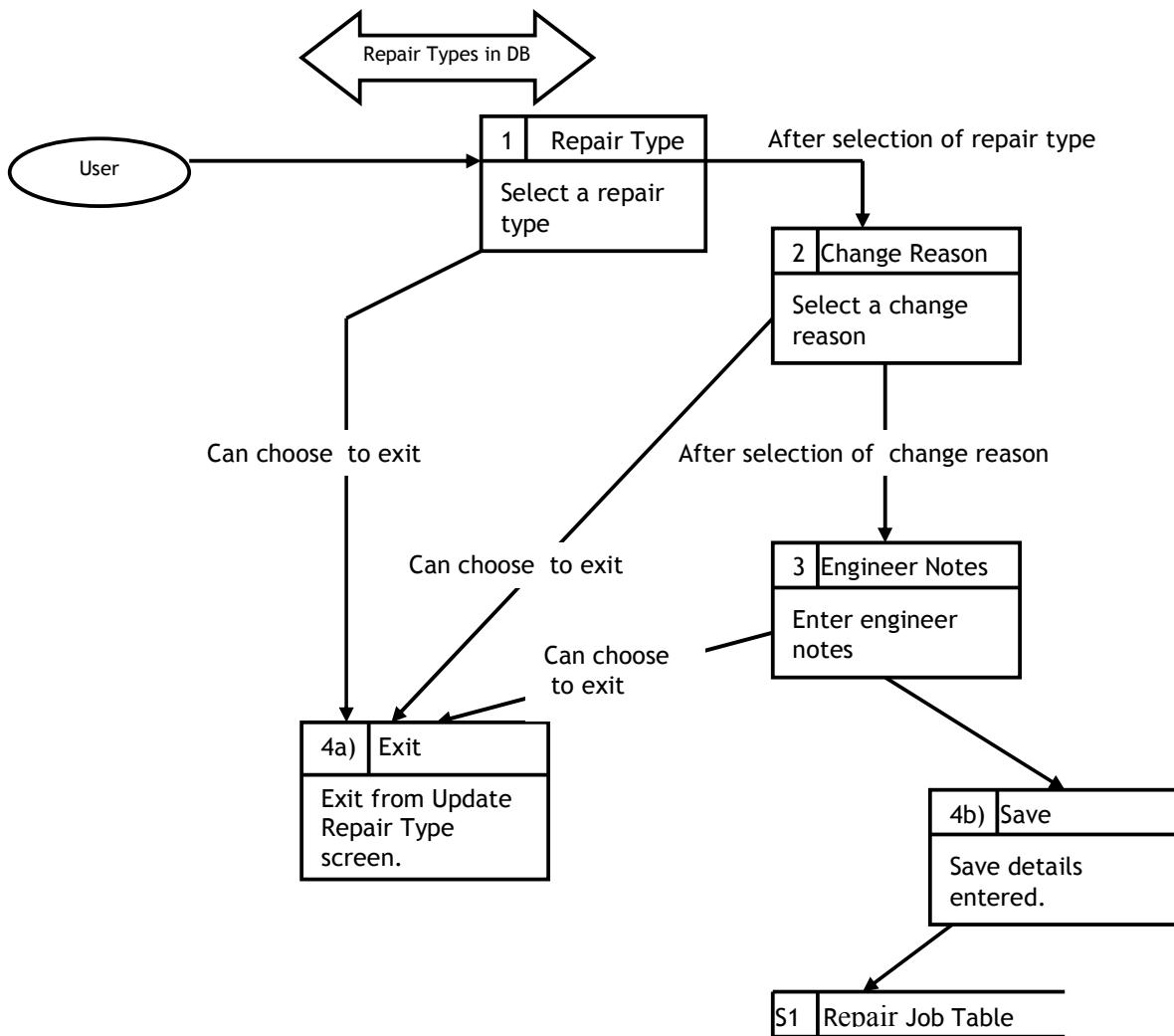
S1 Repair Job Table

5. Resource Flow

A resource flow shows the flow of any physical material from its source to its destination. For this reason they are sometimes referred to as physical flows.

The physical material in question should be given a meaningful name. Resource flows are usually restricted to early, high-level diagrams and are used when a description of the physical flow of materials is considered to be important to help the analysis.

My example would look something like this when I create a DFD for it.



Risks, Issues and Contingency Planning

Some common risks/issues that you might face are: -

1. Half baked development - expected, since your requirements aren't sufficient
 2. Inexperienced resources.
 3. Unavailable functional knowledge in your team.
 4. Ability to comprehend diagrammatic representation in various ways.
 5. Inadequate time to document functionality properly.



6. Unavailability of business analysts, developers, and team architects.
7. Constantly changing requirements.
8. Level of detail required may be too high and time to produce the detail maybe be less.
9. Conflicting diagrams/views arising from ambiguous discussions with business analysts and developers (yes, this is possible!)

And some ways to tackle the risks are: -

1. Know your team strength.
2. Know the technical "modules" as developed by the developers.
3. Collate the modules or decompose them further, depending on the type of testing you plan to do. For e.g. a system test will require you to combine modules while a unit test might require you to break down a module to extensive detail.
4. Assign modules as per person strength and knowledge.
5. Maintain a clean communication line between the team, developers and the business analysts.
6. If the above is done, changing your documentation to match the ever-changing requirements should be possible, although- beware, it could create havoc with your estimates.
7. When conflicts arise, make sure to discuss it out with the parties involved. In case that is not possible, refer to prototypes- if any. If there are no prototypes, validate it from an end user and the desired business approach/need.
8. To avoid ambiguous interpretation of diagrams or documents, make sure to determine standards for your teams to follow. In case of diagrams, choose those approaches, which are clean and analysis free. When a diagram is open for analysis, it is open for interpretation, which may lead to errors. Maintaining error free documentation and tracking of the same can be painful for a large project.
9. If level of detail required is high with limited time on your hands, document the basics - i.e. a very high-level approach and update the test cases as you go ahead with testing.
10. Unfortunately there is little you can do in terms of inexperienced resources or inadequate functional knowledge expertise within your team. Be prepared to hire outside help or build up the knowledge by assigning a team member to the task.

Alternative Ways of Testing

If there is absolutely no time for you to document your system, do consider these other approaches of testing

User/Acceptance Testing

User Testing is used to identify issues with user acceptability of the system. The most trivial to the most complicated defects can be reported here. It is also possibly the best way to determine how well people interact with the system. Resources used are minimal and potentially requires no prior functional or system knowledge. In fact the more unknown a user is to the system, the more preferred he is for user testing. A regressive form of user testing can help uncover some defects in the high-risk areas.

Random Testing

Random Testing has no rules. The tester is free to do what he pleases and as he seems fit. The system will benefit however, if the tester is functionally aware of the domain and has a basic understanding of business requirements. It will prevent reporting of unnecessary defects. A variation of random testing - called as Monte Carlo simulation, which is described as generation of values for uncertain variables over and over to simulate a model can also be used. This can be used for systems, which have large variation in input data of the numbers or data kind.

Customer Stories

User/Customer stories can be defined as simple, clear, brief descriptions of functionality that will be valuable to real users. Small stories are written about the

✓✓M

desired function. The team writing the stories will include customer, project manager, developer and the tester. Concise stories are hand written on small cards such that the stories are small and testable. Once a start is made available to the tester, available functional and system knowledge can help speed up the process of testing and further test case development.

SOFTWARE TESTING

Testing Life Cycle – Bug Life Cycle

Differences between QA, QC and Testing

- **Quality Assurance:** A set of activities designed to ensure that the development and/or maintenance process is adequate to ensure a system will meet its objectives.
- **Quality Control:** A set of activities designed to evaluate a developed work product.
- **Testing:** The process of executing a system with the intent of finding defects. (Note that the "process of executing a system" includes test planning prior to the execution of the test cases.)

QA activities ensure that the process is defined and appropriate. Methodology and standards development are examples of QA activities. A QA review would focus on the process elements of a project - e.g., are requirements being defined at the proper level of detail.

QC activities focus on finding defects in specific deliverables - e.g., are the defined requirements the right requirements

Testing is one example of a QC activity, but there are others such as inspections

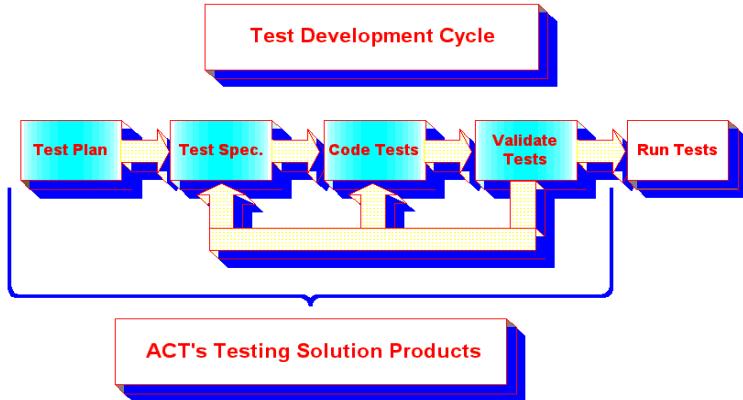
The difference is that **QA is *process* oriented** and **QC is *product* oriented**.

Testing therefore is product oriented and thus is in the QC domain. Testing for quality isn't assuring quality, it's controlling it.

Quality Assurance makes sure you are doing the right things, the right way.

Quality Control makes sure the results of what you've done are what you expected.

Test Development Life Cycle



Test Specifications

The test case specifications should be developed from the test plan and are the second phase of the test development life cycle. The test specification should explain "how" to implement the test cases described in the test plan.

Test Specification Items

Each test specification should contain the following items:

Case No.: The test case number should be a three digit identifier of the following form: c.s.t, where: c- is the chapter number, s- is the section number, and t- is the test case number.

Title: is the title of the test.

ProgName: is the program name containing the test.

Author: is the person who wrote the test specification.

Date: is the date of the last revision to the test case.

Background: (Objectives, Assumptions, References, Success Criteria): Describes in words how to conduct the test.

Expected Error(s): Describes any errors expected

Reference(s): Lists reference documentation used to design the specification.

Data: (Tx Data, Predicted Rx Data): Describes the data flows between the Implementation Under Test (IUT) and the test engine.

Script: (Pseudo Code for Coding Tests): Pseudo code (or real code) used to conduct the test.

Test Results Analysis Report

The Test Results Analysis Report is an analysis of the results of running tests. The results analysis provide management and the development team with a readout of the product quality.

The following sections should be included in the results analysis:

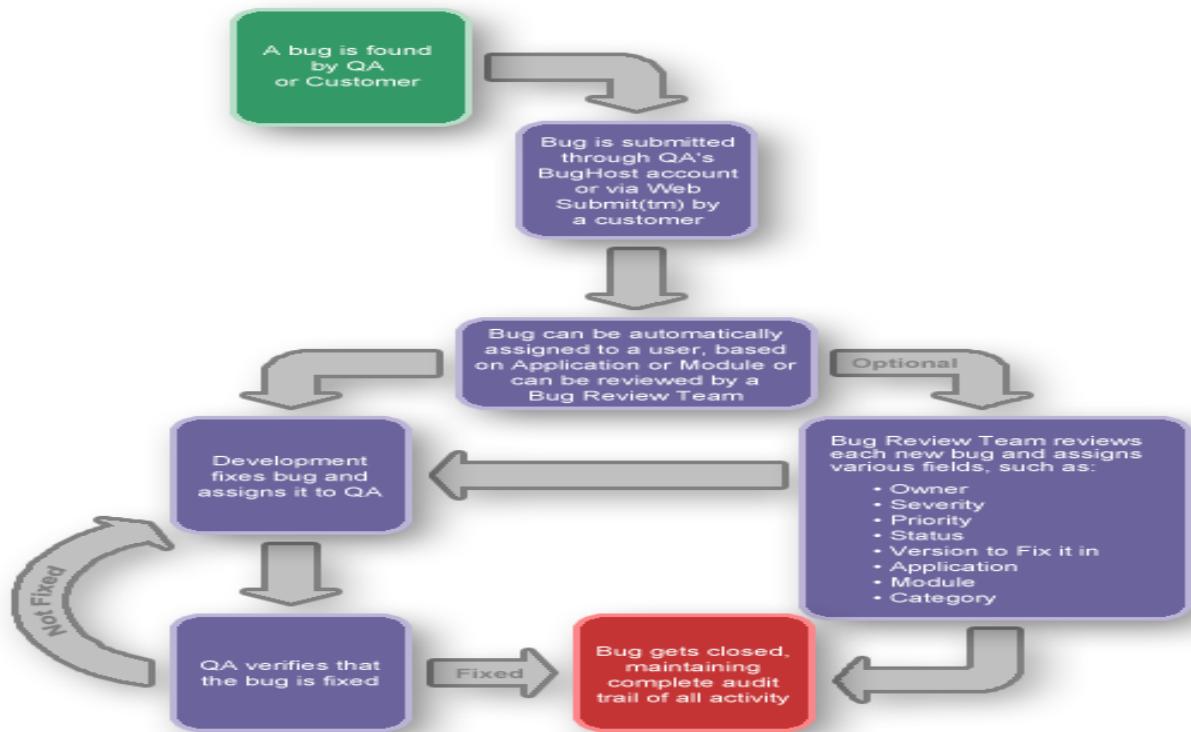
- Management Summary
 - Test Results Analysis
 - Test Logs/Traces
 -
-

Software Quality Concepts

- Quality is conformance to product requirements and should be free.
- Quality is achieved through prevention of defects .
- Quality control is aimed at finding problems as early as possible and fixing them.
- Doing things right the first time is the performance standard which results in zero defects and saves the expenses of doing things over.

- *The expense of quality is nonconformance to product requirements*
 - Quality is what distinguishes a good company from a great one.
 - Quality is meeting or exceeding our customer's needs and requirements.
 - Software Quality is measureable.
 - Quality is continuous improvement.
 - The quality of a software product comes from the quality of the process used to create it.
 - Quality is the Entire Company's Business
 - Our Quality network testing products help make our customers successful.
-
-

Bug Life Cycle



States of a Bug

UNCONFIRMED

This bug has recently been added to the database. Nobody has validated that this bug is true. Users who have the "canconfirm" permission set may confirm this bug, changing its state to NEW. Or, it may be directly resolved and marked RESOLVED.

NEW

This bug has recently been added to the assignee's list of bugs and must be processed. Bugs in this state may be accepted, and become ASSIGNED, passed on to someone else, and remain NEW, or resolved and marked RESOLVED.

ASSIGNED

This bug is not yet resolved, but is assigned to the proper person. From here bugs can be given to another person and become NEW, or resolved and become RESOLVED.

REOPENED

This bug was once resolved, but the resolution was deemed incorrect. For example, a WORKSFORME bug is REOPENED when more information shows up and the bug is now reproducible. From here bugs are either marked ASSIGNED or RESOLVED.

RESOLVED

A resolution has been taken, and it is awaiting verification by QA. From here bugs are either re-opened and become REOPENED, are marked VERIFIED, or are closed for good and marked CLOSED.

VERIFIED

QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. Bugs remain in this state until the product they were reported against actually ships, at which point they become CLOSED.

CLOSED

The bug is considered dead, the resolution is correct. Any zombie bugs who choose to walk the earth again must do so by becoming REOPENED.

Resolution of a Bug

FIXED

A fix for this bug is checked into the tree and tested.

INVALID

The problem described is not a bug

WONTFIX

The problem described is a bug which will never be fixed.

LATER

The problem described is a bug which will not be fixed in this version of the product.

REMIND

The problem described is a bug which will probably not be fixed in this version of the product, but might still be.

DUPLICATE

The problem is a duplicate of an existing bug. Marking a bug duplicate requires the bug# of the duplicating bug and will at least put that bug number in the description field.

WORKSFORME

All attempts at reproducing this bug were futile, reading the code produces no clues as to why this behavior would occur. If more information appears later, please re-assign the bug, for now, file it.

Severity levels of a Bug

This field describes the impact of a bug.

Blocker Blocks development and/or testing work

Critical crashes, loss of data, severe memory leak

Major major loss of function

Minor minor loss of function, or other problem where easy workaround is present

Trivial cosmetic problem like misspelled words or misaligned text

Enhancement Request for enhancement

Priority levels of a Bug

This field describes the importance and order in which a bug should be fixed. This field is utilized by the programmers/engineers to prioritize their work to be done. The available priorities are:

P1 Most important

P2

P3

P4

P5 Least important

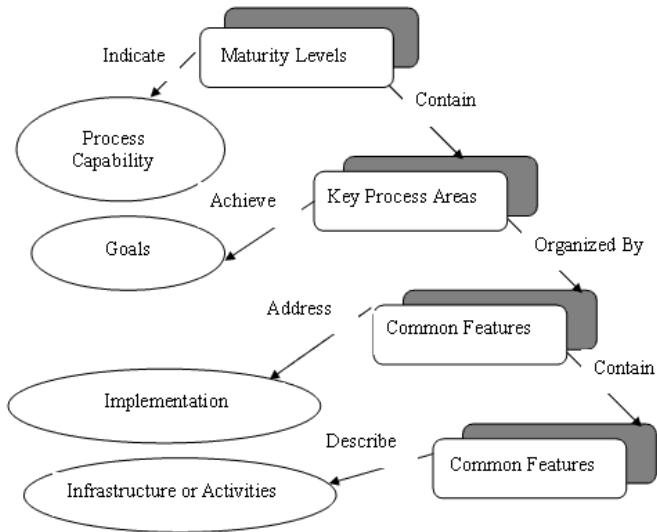


The **Capability Maturity Model for Software (CMM)** is a framework that describes the key elements of an effective software process. There are CMMs for non software processes as well, such as **Business Process Management (BPM)**. The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process. The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality. The *CMM* establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's software process and compare it to the state of the practice of the industry. The CMM can also be used by an organization to plan improvements to its software process. It also reflects the needs of individuals performing software process, improvement, software process assessments, or software capability evaluations; is documented; and is publicly available.

The Five Maturity Levels described by the Capability Maturity Model can be characterized as per their primary process changes made at each level as follows:

- 1) Initial The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
- 2) Repeatable Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
- 3) Defined The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
- 4) Managed Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
- 5) Optimizing Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The structure of the Capability Maturity model is as shown below:



At the Optimizing Level or the CMM level 5, the entire organization is focused on continuous Quantitative feedback from previous projects is used to improve the project management, usually using pilot projects, using the skills shown in level 4. The focus is on continuous process improvement.

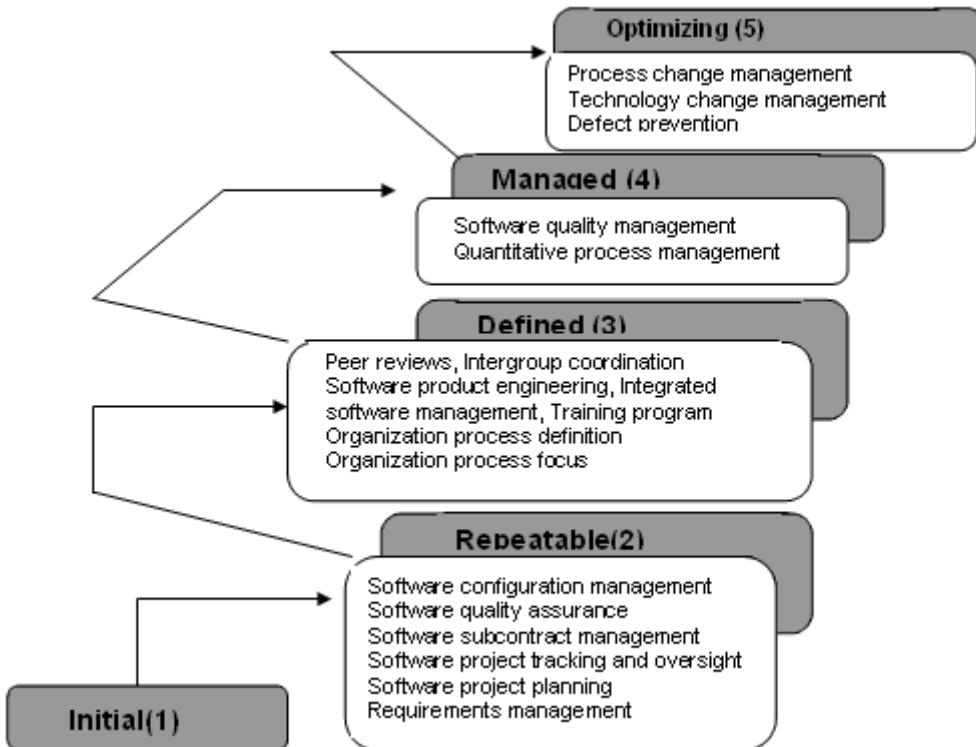
Software project teams in SEI CMM Level 5 organizations analyze defects to determine their causes. Software processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects. The software process capability of Level 5 organizations can be characterized as continuously improving because Level 5 organizations are continuously striving to improve the range of their process capability, thereby improving the process performance of their projects. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

Previously known as Key Process Area (KPA)

A process area (PA) contains the goals that must be reached in order to improve a software process. A PA is said to be satisfied when procedures are in place to reach the corresponding goals.

A software organization has achieved a specific maturity level once all the corresponding PAs are satisfied. The process areas (PA's) have the following features:

- 1) Identify a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing process capability.
- 2) Defined to reside at a single maturity level.
- 3) Identify the issues that must be addressed to achieve a maturity level.



The different maturity levels have different process areas pre-defined as shown in the figure above. The SEI CMMI Level 5 has 3 PA's defined:

- Process change management: To identify the causes of defects and prevent them from recurring.
- Technology change management: To identify beneficial new technology and transfer them in an orderly manner
- Defect prevention: To continually improve the process to improve quality, increase productivity, and decrease development time.

The purpose of Process Change Management is to continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development. When software process improvements are approved for normal practice, the organization's standard software process and the projects' defined software processes are revised as suited.

The goals sought to achieve are as follows:

- Continuous process improvement is planned.
- Participation in the organization's software process improvement activities is organization wide.



- 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.



- Software managers and technical staff receive feedback on the status and results of the software process improvement activities on an event-driven basis.

The primary purpose of periodic reviews by senior management is to provide awareness of, and insight into, software process activities at an appropriate level of abstraction and in a timely manner. The time between reviews should meet the needs of the organization and may be lengthy, as long as adequate mechanisms for exception reporting are available.

The *Capability Maturity Model* has been criticized in that, that it does not describe how to create an effective software development organization. The traits it measures are in practice very hard to develop in an organization, even though they are very easy to recognize. However, it cannot be denied that the Capability Maturity Model reliably assesses an organization's sophistication about software development.

The CMM was invented to give military officers a quick way to assess and describe contractors' abilities to provide correct software on time. It has been a roaring success in this role. So much so that it caused panic-stricken salespeople to clamor for their engineering organizations to "implement CMM" with CMM level 5 being the ultimate.



TRACEABILITY MATRIX



TABLE OF CONTENT

1.INTRODUCTION	<u>1</u>
1.1 Overview	<u>1</u>
1.2 Executive Summary	<u>1</u>
1.3 Disadvantages of not using Traceability Matrix	<u>2</u>
1.4 Where can a Traceability Matrix be used?	<u>2</u>
1.5 Test Coverage	<u>3</u>
2. TRACEABILITY FROM THE TESTING PERSPECTIVE	<u>3</u>
2.1 Traceability Matrix in testing	<u>3</u>
2.2 Developing a Traceability Matrix	<u>3</u>
3 Creating a Traceability Matrix	<u>4</u>



1.INTRODUCTION

This paper is prepared with the intention of providing an insight into the important concept in the life cycle of a Software called Traceability Matrix.

1.1 Overview

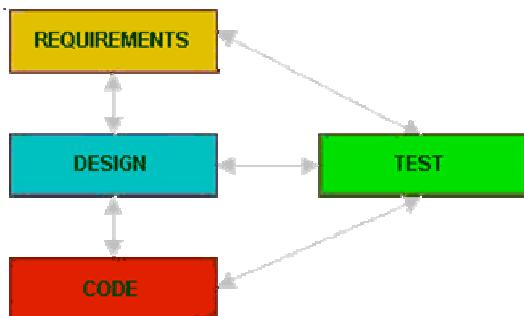
Automation requirement in an organization initiates it to go for a custom built Software. The client who had ordered for the product specifies his requirements to the development Team and the process of Software Development gets started. In addition to the requirements specified by the client, the development team may also propose various value added suggestions that could be added on to the software. But maintaining a track of all the requirements specified in the requirement document and checking whether all the requirements have been met by the end product is a cumbersome and a laborious process. But if high priority is not provided to this aspect of Software development cycle, it may result in a lot of confusion and arguments between the development team and the client once the product is built.

The remedy for this problem is the Traceability Matrix.

1.2 Executive Summary

What is Traceability Matrix?

Requirements tracing is the process of documenting the links between the user requirements for the system you're building and the work products developed to implement and verify those requirements. These work products include Software requirements, design specifications, Software code, test plans and other artifacts of the systems development process. Requirements tracing helps the project team to understand which parts of the design and code implement the user's requirements, and which tests are necessary to verify that the user's requirements have been implemented correctly.





In the diagram, based on the requirements the design is carried out and based on the design, the codes are developed. Finally, based on these the tests are created. At any point of time , there is always the provision for checking which test case was developed for which design and for which requirement that design was carried out. Such a kind of Traceability in the form of a matrix is the Traceability Matrix.

1.3 Disadvantages of not using Traceability Matrix

What happens if the Traceability factor is not considered while developing the software?

- *The system that is built may not have the necessary functionality to meet the customers and users needs and expectations*
- *If there are modifications in the design specifications, there is no means of tracking the changes*
- *If there is no mapping of test cases to the requirements, it may result in missing a major defect in the system*
- *The completed system may have “Extra” functionality that may have not been specified in the design specification , resulting in wastage of manpower, time and effort.*
- *If the code component that constitutes the customer’s high priority requirements is not known, then the areas that need to be worked first may not be known thereby decreasing the chances of shipping a useful product on schedule*
- *A seemingly simple request might involve changes to several parts of the system and if proper Traceability process is not followed, the evaluation of the work that may be needed to satisfy the request may not be correctly evaluated*

1.4 Where can a Traceability Matrix be used?

Is the Traceability Matrix applicable only for big projects?

The Traceability Matrix is an essential part of any Software development process, and hence irrespective of the size of the project, whenever there is a requirement to build a Software this concept comes into focus.

The biggest advantage of Traceability Matrix is backward and forward traceability. (i.e) At any point of time in the development life cycle the status of the project and the modules that have been tested could be easily determined thereby reducing the possibility of speculations about the status of the project.



1.5 Test coverage

Some of the persistent questions that get repeated often with regard to Testing are, “Did we test that the account numbers are valid?”, “Is the pin number field accepting null values?”. These are very difficult question to answer, but with the help of Traceability Matrix, it is possible to relate all possible functional requirements to a test case. The Matrix indicates which test case does that test.

2. TRACEABILITY FROM THE TESTING PERSPECTIVE

The concept of Traceability Matrix is very important from the Testing perspective. This chapter discusses about the point where Traceability Matrix gets involved in Testing and the means of developing it.

2.1 Traceability Matrix in testing

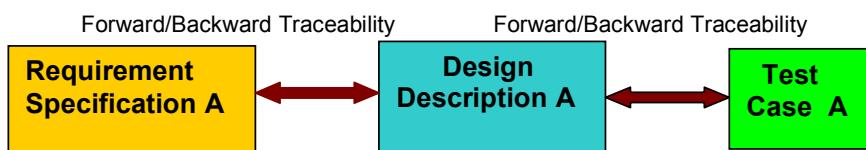
Where exactly does the Traceability Matrix gets involved in the broader picture of Testing?

The Traceability Matrix is created even before any test cases are written, because it is a complete list indicating what has to be tested. Sometimes there is one test case for each requirement or several requirements can be validated by one test scenario. This purely depends on the kind of application that is available for testing.

2.2 Developing a Traceability Matrix

How is the Traceability Matrix developed?

In the design document, if there is a Design description A, which can be traced back to the Requirement Specification A, implying that the design A takes care of Requirement A. Similarly in the test plan, Test Case A takes care of testing the Design A, which in turn takes care of Requirement A and so on.





There has to be references from Design document back to Requirement document, from Test plan back to Design document and so on.

Usually Unit test cases will have Traceability to Design Specification and System test cases /Acceptance Test cases will have Traceability to Requirement Specification. This helps to ensure that no requirement is left uncovered (either un-designed / un-tested).

Requirements Traceability enhances project control and quality. It is a process of documenting the links between user requirements for a system and the work products developed to implement and verify those requirements. It is a technique to support an objective of requirements management, to make certain that the application will meet end-users needs.

3. Creating a Traceability Matrix

Traceability Matrix gives a cross-reference between a test case document and the Functional/design specification document. This document helps to identify, if the Test case document contains tests for all the identified unit functions from the design specification. From this matrix we can collect the percentage of test coverage taking into account the percentage of functionalities to the total tested and not tested.

Requirement	Function Specification	Design Specification	Source Code Files	Test Cases
Modification of Auto Load Process	BRD section 6.5.8	LLD section 3.1	SCW2FORCE.PRG	Test Case No.: 1 to 10 (Reference: 1099_Reportin g_Test_Record.Doc)
Modification of Force Balance Process	BRD section 6.5.8	LLD section 3.2	W2FORCE.PRG	Test Case No.: 1 to 10 (Reference: 1099_Reportin g_Test_Record.Doc)

According to the above table, the requirement specifications are clearly spelt out in the requirement column. The functional specification notified as BRD section 6.5.8 tells about the requirement that has been specified in the Requirement document (i.e it tells about the requirement to which a test case is designed). With the help of the Design Specification, it is possible to drill down to the level of identifying the Low Level Design for the Requirement specified.



Based on the requirements, code is developed. At any point of time, the program corresponding to a particular requirement could be easily traced back. The test cases corresponding to the requirements are available in the Test Case column.

Comment [j1]: code is