UNIT- V

User Interface Design: *The Golden Rules, User Interface Analysis and Design, Interface Analysis, Interface Design Steps, WebApp Interface Design, Design Evaluation, Elements of* Software Quality Assurance (SQA),*SQA Tasks, Goals& Metrics, Statistical SQA,S/W Reliability.*

Software Testing Strategies: *A strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Test Strategies for Object-Oriented Software, Test Strategies for WebApps, Validation Testing, System Testing, The Art of Debugging.*

Testing Conventional Applications: *Software Testing Fundamentals, Internal and External Views of Testing, White-Box Testing, basic Path testing, Control Structure Testing, Black-Box Testing, Model-based Testing, Testing for Specialized Environments, Architectures and Applications, Patterns for Software Testing.*

User Interface Design

User Interface Design creates an effective communication medium between human and computer.

THE GOLDEN RULES

Theo Mandel coins **three** *golden rules:*

1. **Place the user incontrol.**
2. **Reduce the user's memoryload.**
3. **Make the interfaceconsistent.**

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

Place the User in Control

Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing

the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

Reduce the User's Memory Load

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original defaultvalues.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to beinvoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized

hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mousepick.

Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined andimplemented.

Mandel defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.

Maintain consistency across a family of applications. A set of applications should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

USER INTERFACE ANALYSIS AND DESIGN

Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer **(**or the software engineer) establishes a ***user model***, the software engineer creates a ***design model***, the end user develops a mental image that is often called the user's ***mental model*** or the ***system perception***, and the implementers of the system create an ***implementation model.***

The user model establishes the profile of end users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality". Users can be categorized as:

Novices. No syntactic knowledge1 of the system and little semantic knowledge of theapplication or computer usage ingeneral.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's ***mental model*** (system perception) is the image of the system that end users carry in their heads.

The ***implementation model*** combines the outward manifestation of the computer based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively.

These models enable the interface designer to satisfy a key element of the most important principle of user interface design: "**Know the user, know the tasks**."

The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model Referring to the following figure, the user interface analysis and design process begins at the interior of the spiral and encompasses **four** distinct framework activities

(1) interface analysis andmodeling,

(2) interfacedesign,

(3) interface construction,and

(4) interfacevalidation.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. Once general requirements have been defined, a more detailed ***task analysis*** is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

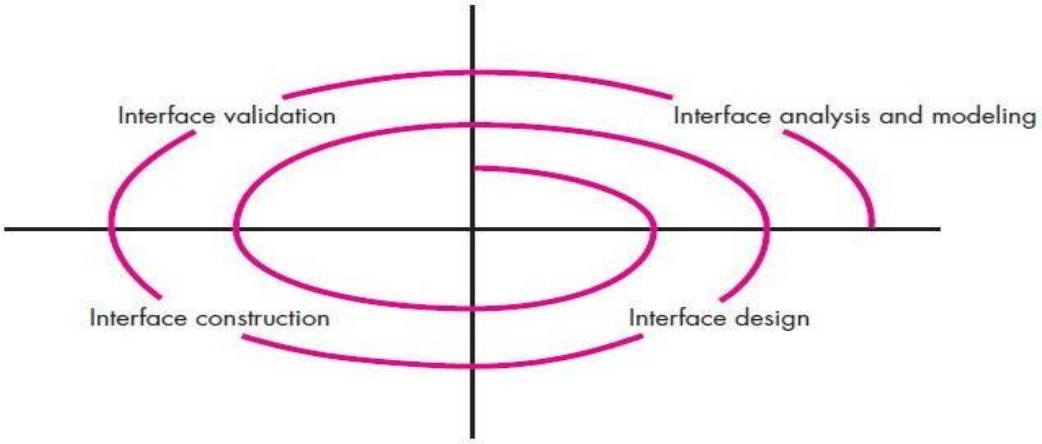


Fig : The user interface design process

Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be locatedphysically?
- Will the user be sitting, standing, or performing other tasks unrelated to theinterface?
- Does the interface hardware accommodate space, light, or noiseconstraints?
- Are there special human factors considerations driven by environmentalfactors?

The goal of ***interface design*** is to define a set of interface objects and actions that enable a user to perform all defined tasks in a manner that meets every usability goal defined for thesystem.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in theirwork.

INTERFACE ANALYSIS

A key tenet of all software engineering process models is this: ***understand the problem before you attempt to design a solution***. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system

through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. These elements of interface analysis with the intent of establishing a solid foundation for the design tasks thatfollow.

User Analysis

The phrase "user interface" is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Information from a broad array of sources can be used to accomplish this:

- **User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focusgroups.
- **Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.
- **Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy touse.

The following set of questions will help you to better understand the users of a system:

• Are users trained professionals, technicians, clerical, or manufacturingworkers?

• What level of formal education does the average userhave?

• Are the users capable of learning from written materials or have they expressed a desire for classroomtraining?

• Are users expert typists or keyboardphobic?

• What is the age range of the usercommunity?

• Will the users be represented predominately by onegender?

• How are users compensated for the work they perform? • Do users work normaloffice hours or do they work until the job isdone?

• Is the software to be an integral part of the work users do or will it be used only

occasionally?

• What is the primary spoken language amongusers?

• What are the consequences if a user makes a mistake using thesystem?

• Are users experts in the subject matter that is addressed by thesystem?

• Do users want to know about the technology that sits behind theinterface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

Task Analysis and Modeling

The goal of task analysis is to answer the following questions:

• What work will the user perform in specificcircumstances?

• What tasks and subtasks will be performed as the user does thework?

• What specific problem domain objects will the user manipulate as work is performed?

• What is the sequence of work tasks—the workflow?

• What is the hierarchy oftasks?

Analysis of Display Content

Analysis of display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). These data objects may be (1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content are considered. Among the questions that are asked and answered are:

• Are different types of data assigned to consistent geographic locations on thescreen (e.g., photos always appear in the upper right-handcorner)?

• Can the user customize the screen location forcontent?

• Is proper on-screen identification assigned to allcontent?

• If a large report is to be presented, how should it be partitioned for ease of understanding?

• Will mechanisms be available for moving directly to summary information forlarge collections ofdata?

• Will graphical output be scaled to fit within the bounds of the display device that is used?

• How will color be used to enhanceunderstanding?

• How will error messages and warnings be presented to theuser?

The answers to these (and other) questions will help you to establish requirements

Analysis of the Work Environment

Hackos and Redish discuss the importance of work environment analysis when they state: *eople do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.*

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface designcommences.

INTERFACE DESIGN STEPS

Although many different user interface design models have been proposed, all suggest some combination of the following steps:

1. Define interface objectsand actions(operations).
2. Define events (user actions) that will cause the state of the user interface tochange. Model thisbehavior.
3. Show each interface state as it will actually look to the enduser.
4. Indicate how the user interprets the state of the system from informationprovided through theinterface.

Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important

step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions. Once the objects and actions have been defined and elaborated iteratively, they arecategorized by type. Target, source, and application objects areidentified.

User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem**.**

Design Issues

As the design of a user interface evolves, **four** common design issues almost always surface: **system response time, user help facilities, error information handling, and command labelin**g.

- **Response time.** System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action until the software responds with desired output or action. System response time has **two** important characteristics: **length and variability**. If system response is too **long**, user frustration and stress are inevitable. ***Variability*** refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relativelylong.
- **Help facilities.** Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick.

  A number of design issues must be addressed when a help facility is considered:

  • Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for allfunctions.

  • How will the user request help? Options include a help menu, a special function key, or a HELPcommand.

  • How will help be represented? Options include a separate window, a reference

to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screenlocation.

• How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or controlsequence.

• How will help information bestructured?

- **Error handling.** Error messages and warnings are **"bad news**" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. In general, every error message or warning produced by an interactive system should have the followingcharacteristics:

  • The message should describe the problem in jargon that the user canunderstand.

  • The message should provide constructive advice for recovering from theerror.

  • The message should indicate any negative consequences of the error so that the user can check to ensure that they have notoccurred

  • The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."

  • The message should be "nonjudgmental." That is, the wording should never place blame on theuser.

- **Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer acommand-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

  • Will every menu option have a corresponding command?

  • What form will commands take? Options include a control sequence (e.g.,alt-P), function keys, or a typedword.

  • How difficult will it be to learn and remember the commands? What can be done if a command isforgotten?

- Can commands be customized or abbreviated by theuser?
- Are menu labels self-explanatory within the context of theinterface?
- Are submenus consistent with the function implied by a master menu item?

Application accessibility.: *.Accessibility*for users who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.

Internationalization. Interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software**.** ***Localization*** features enable the interface to be customized for a specificmarket.

A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

WEBAPP INTERFACE DESIGN

Dix argues that you should design a WebApp interface so that it answers **three** primary questions or the end user:

Where am I? The interface should (1) provide an indication of the WebApp that has been accessed and (2) inform the user of her location in the content hierarchy.

What can I do now? The interface should always help the user understand his current options like what functions are available, what links are live, what content is relevant?

Where have I been, where am I going? The interface must facilitate navigation. Hence, it mustprovide a "map" of where the user has been and what paths may be taken to move elsewhere within the WebApp.

An effective WebApp interface must provide answers for each of these questions as the end user navigates through content and functionality.

Interface Design Principles and Guidelines

A good WebApp interface is understandable and forgiving, providing the user with a sense of control. Bruce Tognozzi defines a set of fundamental characteristics that all interfaces should exhibit and in doing so, establishes a philosophy that should be followed by every WebApp interface designer.

Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.

Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time. Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

In order to design WebApp interfaces that exhibit these characteristics, Tognozzi identifies a set of overriding design principles:

- **Anticipation.** *A WebApp should be designed so that it anticipates the user's nextmove.*
- **Communication.** *The interface should communicate the status of any activity initiated by the user.* Communication can be obvious or subtle. The interface should also communicate user status (e.g., the user's identification) and her location within the WebApp contenthierarchy.
- **Consistency.** *The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout theWebApp.*
- **Controlled autonomy.** *The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for theapplication.*
- **Efficiency.** *The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client server environment that executesit.*
- **Flexibility.** *The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion.* In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigationpaths.

- **Focus.** *The WebApp interface (and the content it presents) should stay focused on the user task(s) athand.*
- **Fitt's law.** *"The time to acquire a target is a function of the distance to and size of the target".* Fitt'slaw "is an effective method of modeling rapid, aimed movements, where one appendage starts at rest at a specific start position, and moves to rest within a target area". If a sequence of selections or standardized inputs is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the nextselection.
- **Human interface objects.** *A vast library of reusable human interface objects has been developed for WebApps. Usethem.*
- **Latency reduction.** *Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed.* In addition to reducing latency, delays must be acknowledged so that the userunderstands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the WebApp, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providingsome entertainment (e.g., an animation or text presentation) while lengthy processingoccurs.
- **Learnability.** *A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.* In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to theuser.
- **Metaphors.** *An interface that uses an interaction metaphor is easier to learn and easier touse,aslongasthemetaphorisappropriatefortheapplicationandtheuser.*

  Metaphors are an excellent idea because they mirror real-world experience. Just be sure that the metaphor you choose is well known to end users.
- **Maintain work product integrity.** *A work product (e.g., a form completed by the user, a user-specified list) must be automatically saved so that it will not be lost if an error occurs.*
- **Readability.** *All information presented through the interface should be readable by young and old.* The interface designer should emphasize readable type styles, font sizes,

and color background choices that enhancecontrast.

- **Track state.** *When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she leftoff.*
- **Visible navigation.** *A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought tothem"*

Nielsen and Wagner suggest a few pragmatic interface design guidelines that provide anice complement to the principles suggested earlier in thissection:

• Reading speed on a computer monitor is approximately 25 percent slower than reading speed for hardcopy. Therefore, do not force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists innavigation.

• Avoid "under construction" signs—an unnecessary link is sure todisappoint.

• Users prefer not to scroll. Important information should be placed within the dimensions of a typical browserwindow.

• Navigation menus and head bars should be designed consistently and should be available on all pages that are available to the user. The design should not rely on browser functions to assist innavigation.

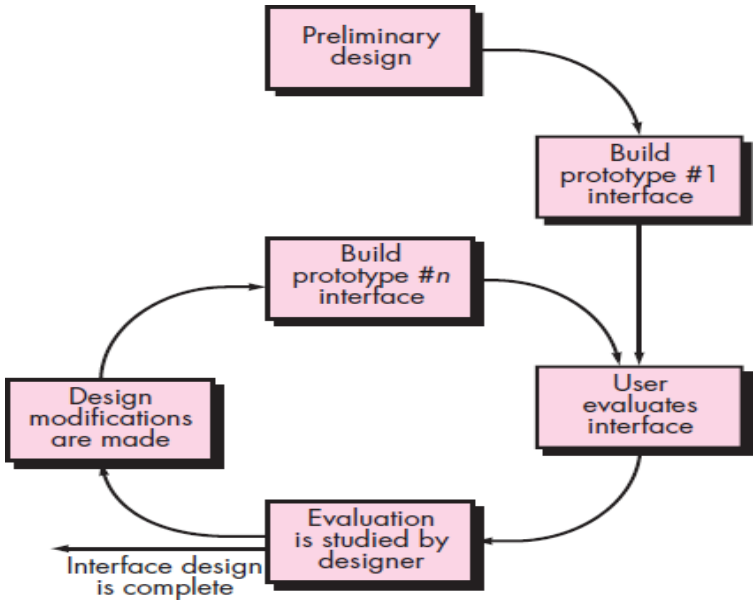
• Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent isunclear.

• Navigation options should be obvious, even to the casual user. The user should not have to search the screen to determine how to link to other content orservices.

DESIGN EVALUATION

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of endusers.

Fig : The interface design evaluation cycle



The user interface evaluation cycle takes the form shown in above figure. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface.In addition, if formal evaluation techniques are used, you can extract information from thesedata.

Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of thesystem.

2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of thesystem.

3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of thesystem.

4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by theuser.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be: (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert scales (e.g., strongly agree, somewhat agree), (5) percentage (subjective) response, or (6) open-ended.

Software Quality Assurance − Software Quality Assurance (SQA) is a set of activities to ensure the quality in software engineering processes that ultimately result in quality software products. The activities establish and evaluate the processes that produce products. It involves process-focused action.

SQA practices are implemented in most types of software development, regardless of the underlying software development model being used. SQA incorporates and implements software testing methodologies to test the software. Rather than checking for quality after completion, SQA processes test for quality in each phase of development, until the software is complete. With SQA, the software development process moves into the next phase only once the current/previous phase complies with the required quality standards. SQA generally works on one or more industry standards that help in building software quality guidelines and implementation strategies.

It includes the following activities −

- Process definition and implementation
- Auditing
- Training

Processes could be −

- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing, etc.

Once the processes have been defined and implemented, Quality Assurance has the following responsibilities −

- Identify the weaknesses in the processes
- Correct those weaknesses to continually improve the process

Components of SQA System

An SQA system always combines a wide range of SQA components. These components can be classified into the following six classes −

Pre-project components

This assures that the project commitments have been clearly defined considering the resources required, the schedule and budget; and the development and quality plans have been correctly determined.

Components of project life cycle activities assessment

The project life cycle is composed of two stages: the development life cycle stage and the operation–maintenance stage.

The development life cycle stage components detect design and programming errors. Its components are divided into the following sub-classes: Reviews, Expert opinions, and Software testing.

The SQA components used during the operation–maintenance phase include specialized maintenance components as well as development life cycle components, which are applied mainly for functionality to improve the maintenance tasks.

Components of infrastructure error prevention and improvement

The main objective of these components, which is applied throughout the entire organization, is to eliminate or at least reduce the rate of errors, based on the organization's accumulated SQA experience.

Components of software quality management

This class of components deal with several goals, such as the control of development and maintenance activities, and the introduction of early managerial support actions that mainly prevent or minimize schedule and budget failures and their outcomes.

Components of standardization, certification, and SQA system assessment

These components implement international professional and managerial standards within the organization. The main objectives of this class are utilization of international professional knowledge, improvement of coordination of the organizational quality systems with other organizations, and assessment of the achievements of quality systems according to a common scale. The various standards may be classified into two main groups: quality management standards and project process standards.

Organizing for SQA – the human components

The SQA organizational base includes managers, testing personnel, the SQA unit and the persons interested in software quality such as SQA trustees, SQA committee members, and SQA forum members. Their main objectives are to initiate and support the implementation of SQA components, detect deviations from SQA procedures and methodology, and suggest improvements.

Pre-project Software Quality Components

These components help to improve the preliminary steps taken before starting a project. It includes −

- Contract Review
- Development and Quality Plans

Contract Review

Normally, a software is developed for a contract negotiated with a customer or for an internal order to develop a firmware to be embedded within a hardware product. In all these cases, the development unit is committed to an agreed-upon functional specification, budget and schedule. Hence, contract review activities must include a detailed examination of the project proposal draft and the contract drafts.

Specifically, contract review activities include −

- Clarification of the customer's requirements
- Review of the project's schedule and resource requirement estimates
- Evaluation of the professional staff's capacity to carry out the proposed project
- Evaluation of the customer's capacity to fulfil his obligations

- Evaluation of development risks

Development and Quality Plans

After signing the software development contract with an organization or an internal department of the same organization, a development plan of the project and its integrated quality assurance activities are prepared. These plans include additional details and needed revisions based on prior plans that provided the basis for the current proposal and contract.

Most of the time, it takes several months between the tender submission and the signing of the contract. During these period, resources such as staff availability, professional capabilities may get changed. The plans are then revised to reflect the changes that occurred in the interim.

The main issues treated in the project development plan are −

- Schedules
- Required manpower and hardware resources
- Risk evaluations
- Organizational issues: team members, subcontractors and partnerships
- Project methodology, development tools, etc.
- Software reuse plans

The main issues treated in the project's quality plan are −

- Quality goals, expressed in the appropriate measurable terms
- Criteria for starting and ending each project stage
- Lists of reviews, tests, and other scheduled verification and validation activities

SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

• To perform effective testing, you should conduct effective technical reviews. Bydoing this, many errors will be eliminated before testingcommences.

• Testing begins at the component level and works "outward" toward the integrationof the entire computer-basedsystem.

• Different testing techniques are appropriate for different softwareengineering approaches and at different points intime.

• Testing is conducted by the developer of the software and (for

large projects) an independent testgroup.

• Testing and debugging are different activities, but debugging must be accommodated in any testingstrategy.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as **verification** and **validation** (V&V). ***Verification*** refers to the set of tasks that ensure that software correctly implements a specific function. ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Boehm states this another way:

> **Verification: "Are we building the productright?" Validation: "Are we building the rightproduct?"**

Verification and validation includes a wide array of **SQA** activities: **technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing**.

Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software.

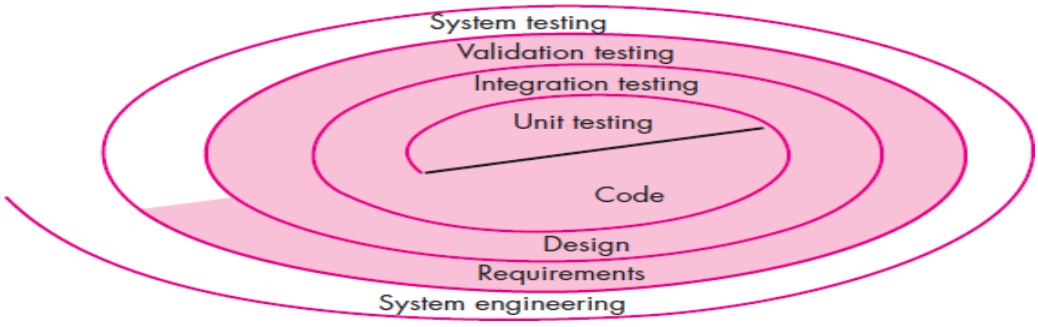
The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group becomeinvolved.

The role of an ***independent test group*** **(ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the

developer must be available to correct errors that are uncovered.

Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in following figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counter clockwise)



along streamlines that decrease the level of abstraction on each turn.

Fig : Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code. Testing progresses by moving outward along the spiral to ***integration testing***, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter ***validation testing***, where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at ***system testing***, where the software and other system elements are tested as awhole.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of **four** steps that are implemented sequentially. The steps are shown in following figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete

software package. ***Integration testing*** addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

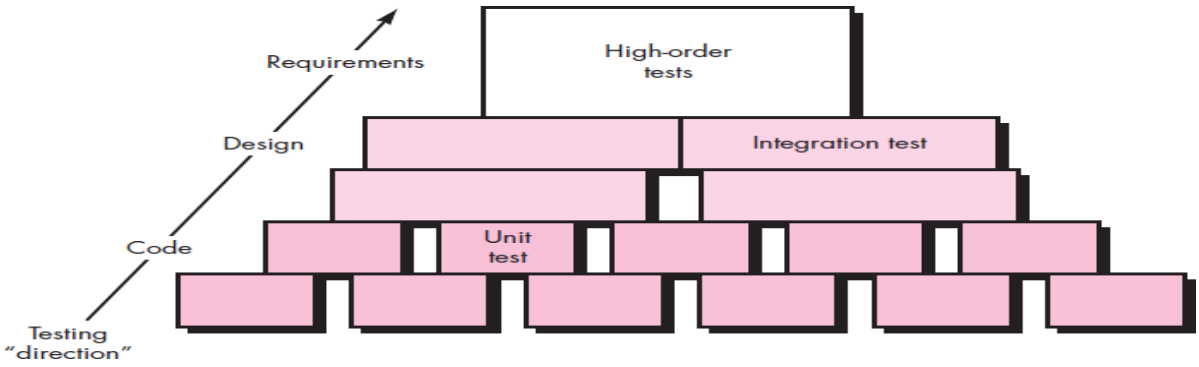


Fig : Software testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). ***System testing*** verifies that all elements mesh properly and that overall system function/performance isachieved.

Criteria for Completion of Testing

"When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the user executes a computer program, the program is beingtested.

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The ***clean room software engineering*** approach suggests statistical

use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

. By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we donetesting?"

STRATEGIC ISSUES

Tom Gilb argues that a software testing strategy will succeed when software testers:

- ***Specify product requirements in a quantifiable manner long beforetestingcommences***. Although the overriding objective of testing is to find errors, a good testingstrategyalso assessesotherqualitycharacteristicssuchasportability,maintainability,andusability..These should be specified in a way that is measurable so that testing results are unambiguous.
- ***State testing objectives explicitly***. The specific objectives of testing should be stated in measurableterms.
- ***Understand the users of the software and develop a profile for each user category***. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of theproduct.
- ***Develop a testing plan that emphasizes "rapid cycle testing."*** Gilb recommends that a software team "learn to test in rapid cycles The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding teststrategies.
- ***Build "robust" software that is designed to test itself***. Software should be designed in a manner that uses anti bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regressiontesting.
- ***Use effective technical reviews as a filter prior to testing***. Technical reviews can be as effective as testing in uncoveringerrors.
- ***Conduct technical reviews to assess the test strategy and test cases themselves***. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves productquality.

- ***Develop a continuous improvement approach for the testing process***. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for softwaretesting.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit-test considerations. Unit tests are illustrated schematically in following figure. The module**interface** is tested to ensure that information properly flows into and out of the program unit under test. **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All **independent paths** through the control structure are exercised to ensure that all statements in a module have been executed at least once. **Boundary conditions** are tested to ensure that the moduleoperates properly at boundaries established to limit or restrict processing. And finally, all **error-handling paths** are tested.

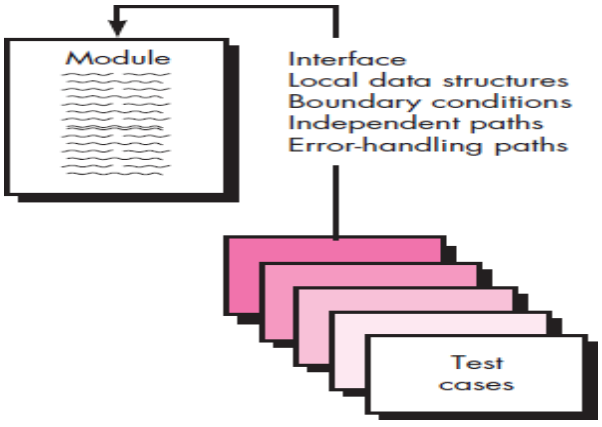


Fig : Unit Test

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i*passes is invoked, when the maximum or minimum allowable value is encountered.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach ***antibugging***.

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated.

The unit test environment is illustrated in following figure.. In most applications a ***driver*** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. ***Stubs*** serve to replace modules that are subordinate (invoked by) the component to be tested.

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Integration Testing

Integration testing is a systematic technique for constructing the software architecture 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 bydesign.

There is often a tendency to attempt non incremental integration; that is,

to construct the program using a "**big bang**" approach. All components are combined in advance. The entire program is tested as a **whole**. If a set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. There are **two** different incremental integration strategies :

Top-down integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a **depth-first or breadth-first** manner. Referring to the following figure, ***depth-first integration*** integrates all components on a major control path of the program structure. For example, selectingthe left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and right-hand control paths are built. ***Breadth-first integration*** incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on,follows.

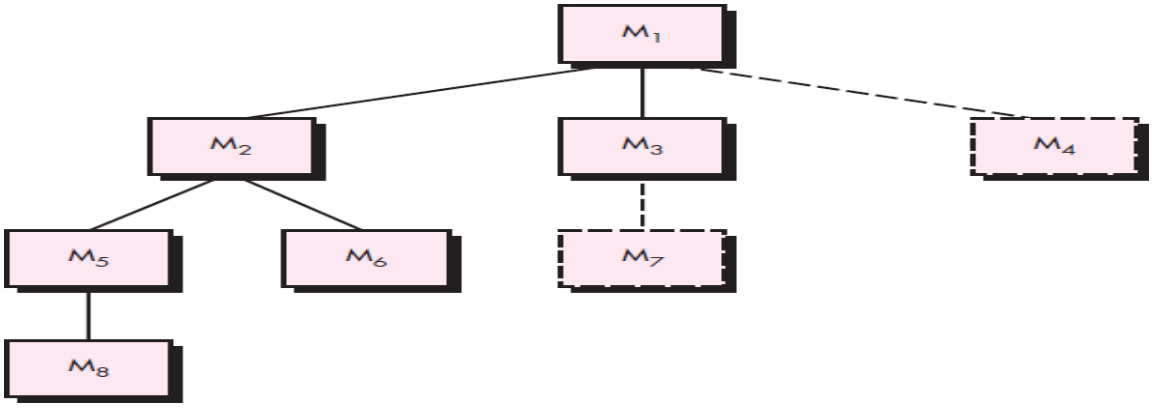


Fig : Top-down integration

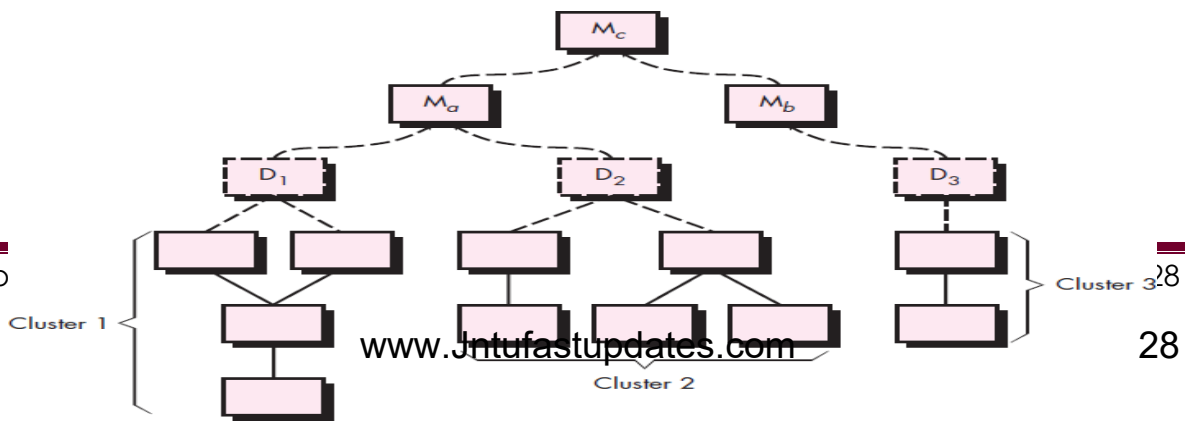
The integration process is performed in a series of **five** steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main controlmodule.
2. Depending on the integration approach selected (i.e., depth or breadthfirst), subordinate stubs are replaced one at a time with actualcomponents.
3. Tests are conducted as each component isintegrated.
4. On completion of each set of tests, another stub is replaced with the realcomponent.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not beenintroduced.

Bottom-up integration. *Bottom-up integration testing,* as its name implies, 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, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in following figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a*. Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3 for cluster 3 is removed prior to integration with



module M*b*. Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.

Fig : Bottom-up integration

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression testing. R*egression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated **capture/playback** tools. ***Capture/playback tools*** enable the software engineer to capture test cases and results for subsequent playback and comparison. The ***regression test suite*** (the subset of tests to be executed) contains **three** different classes of test cases:

- A representative sample of tests that will exercise all softwarefunctions.
- Additional tests that focus on software functions that are likely to be affected bythe change.
- Tests that focus on the software components that have beenchanged.

As integration testing proceeds, the number of regression tests can grow quite large.

Smoke testing. *Smoke testing* is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke- testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more productfunctions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "showstopper" errors that have the highest likelihood of throwing the

software project behindschedule.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottomup.

Smoke testing provides a number of **benefits** when it is applied on complex, time critical software projects:

- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors areuncovered.
- ***The quality of the end product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality willresult.
- ***Error diagnosis and correction are simplified***. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discoverederror.
- ***Progress is easier to assess***. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is beingmade.

TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of **unit testing**.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the

module interface, class testing for OO software isdriven by the operations encapsulated by the class and the state behavior of theclass.

Integration Testing in the OO Context

There are two different strategies for integration testing of OO systems.

The first, ***thread-based testing***, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.

The second integration approach, ***use-based testing***, begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server*classes. After the independent classes are tested, the next layer of classes, called ***dependent classes***, that use the independent classes are tested.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.

TEST STRATEGIES FOR WEBAPPS

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncovererrors.
2. The interface model is reviewed to ensure that all use cases can beaccommodated.
3. The design model for the WebApp is reviewed to uncover navigationerrors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unittested.
6. Navigation throughout the architecture istested.
7. The WebApp is implemented in a variety of different environmentalconfigurations and is tested for compatibility with eachconfiguration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within itsenvironment.
9. Performance tests areconducted.
10. The WebApp is tested by a controlled and monitored population of

end users. The results of their interaction with the system are evaluated for content and navigationerrors, usability concerns, compatibility concerns, and WebApp reliability andperformance.

VALIDATION TESTING

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.

Validation can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created.

Configuration Review

An important element of the validation process is a ***configuration review***. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an**audit**

Alpha and Beta Testing

When custom software is built for one customer, a series of **acceptance tests** are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

The ***alpha test*** is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage

problems. Alpha tests are conducted in a controlled environment.

The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software

in an environment that cannot be controlled by the developer. The customer records all problems that are encountered during beta testing and reports these to the developer at regular intervals.

A variation on beta testing, called ***customer acceptance testing***, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from thedeveloper.

SYSTEM TESTING

System testing is actually 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 allocatedfunctions.

Recovery Testing

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), 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.

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, the tester plays the role(s) of the individual who desires to penetrate the system. Good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will beobtained.

Stress Testing

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. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created.

A variation of stress testing is a technique called ***sensitivity testing***. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

Deployment Testing

Deployment testing, sometimes called ***configuration testing***, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to endusers.

THE ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art.

The Debugging Process

Debugging is not testing but often occurs as a consequence of testing. Referring to the following figure, the debugging process begins with the execution of a test case.. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of **two** outcomes:

(1) the cause will be found and correctedor

(2) the cause will not befound.

A few characteristics of bugs provide some clues:

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 components exacerbate thissituation.

2. The symptom may disappear (temporarily) when another error iscorrected.

3. The symptom may actually be caused by non errors (e.g., round-offinaccuracies).

4. The symptom may be caused by human error that is not easilytraced.

5. The symptom may be a result of timing problems, rather than processingproblems.

6. It may be difficult to accurately reproduce inputconditions

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and softwareinextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on differentprocessors.

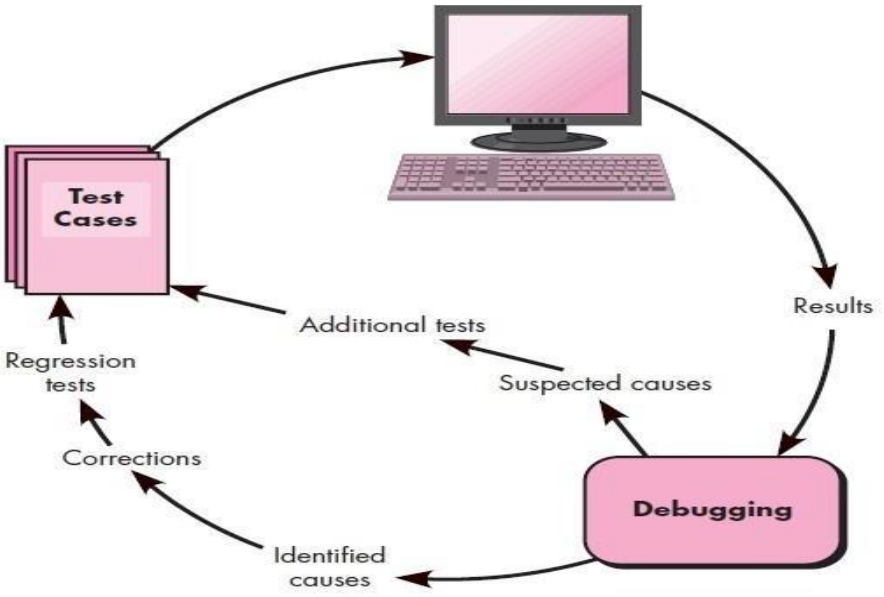


Fig : The Debugging Process

Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same

education and experience.

Debugging Strategies

Bradley describes the debugging approach in this way:

Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem's source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined. In general, **three** debugging strategies have been proposed

(1) bruteforce,

(2) back tracking,and

(3) cause elimination.

Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging tactics.

The ***brute force*** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when **all else fails**.

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageablylarge.

The third approach to debugging is ***cause elimination.*** It is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence

Correcting the Error

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck suggests **three** simple questions that you should ask before making the "correction" that removes the cause of a bug:

1. ***Is the cause of the bug reproduced in another part of the program****?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may

result in the discovery of othererrors.

2. ***What "next bug" might be introduced by the fix I'm about to make?*** Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change ismade.

3. ***What could we have done to prevent this bug in the first place?*** This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all futureprograms.

TESTING CONVENTIONAL APPLICATIONS

SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach provides the following definition for testability: "*Software testability* is simply how easily can be tested." The following **characteristics** lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

Observability. "What you see is what you test."

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity , structural simplicity*, and *codesimplicity*

Stability. "The fewer the changes, the fewer the disruptions totesting."

Understandability. "The more information we have, the smarter we will test."

Test Characteristics. Kaner, Falk, and Nguyen suggest the following attributes of a "good" test: ***A good test has a high probability of finding an error***. To achieve this goal,

the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed.

A good test is not redundant. Testing time and resources are limited. There is no point inconducting a test that has the same purpose as another test. Every test should have a different purpose.

A good test should be "best of breed" In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

INTERNAL AND EXTERNAL VIEWS OF TESTING

Any engineered product can be tested in one of **two** ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product.

The first test approach takes an **external view** and is called **black-box testing.** The second requires an **internal view** and is termed **white-boxtesting**.

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

WHITE-BOX TESTING

White-box testing, sometimes called ***glass-box testing***, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

Using white-box testing methods, you can derive test cases that

1) guarantee that all independent paths within a module have been exercised at leastonce,

2) exercise all logical decisions on their true and falsesides,

3) execute all loops at their boundaries and within their operational bounds,and

4) exercise internal data structures to ensure theirvalidity.

BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test-case designer 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.

Flow Graph Notation

A simple notation for the representation of control flow, called a *flow graph* (or *program graph*). The flow graph depicts logical control flow using the notation illustrated in following figure.

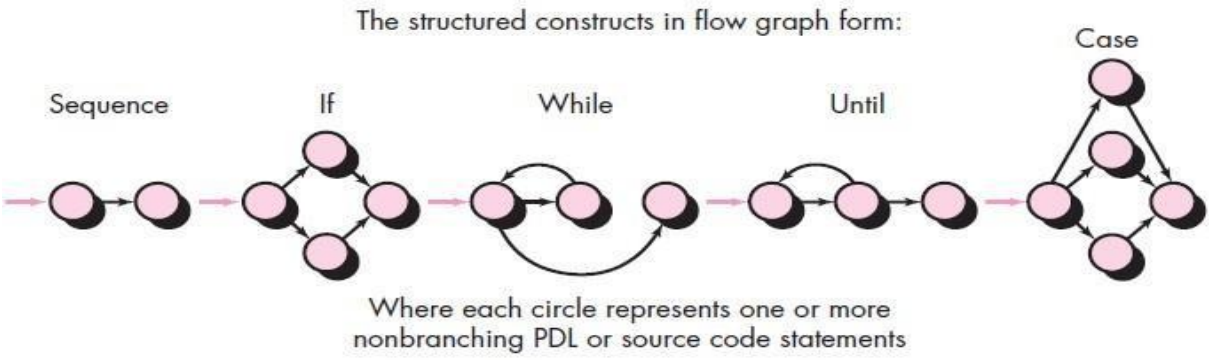


Fig : Flow Graph Notation

To illustrate the use of a flow graph, consider the procedural design representation in following figure (a). Here, a flowchart is used to depict program control structure. Figure (b) maps the flowchart into a corresponding flow graph.

Referring to figure (b), each circle, called a ***flow graph node***, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called ***edges* or *links***, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements. Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region Each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it

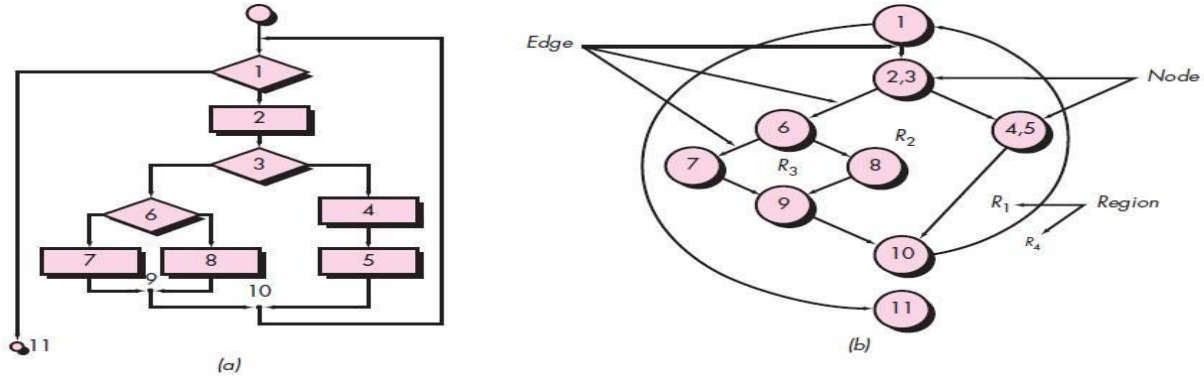


Fig : (a) Flowchart and (b) flow graph

Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in figure (b) is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3:1-2-3-6-8-9-10-1-11

Path 4:1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any newedges.

How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer. ***Cyclomatic complexity*** is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the 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 you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of **three** ways:

1. The number of regions of the flow graph corresponds to the cyclomaticcomplexity.

2. Cyclomatic complexity $V(G)$ for a flow graph G is definedas

$V(G) = E - N + 2$;

where *E* is the number of flow graph edges and *N* is the number of flow graph nodes.

3. Cyclomatic complexity *V*(*G*) for a flow graph *G* is also definedas

V(*G*) = *P*+1

where *P* is the number of predicate nodes contained in the flow graph *G*.

Referring once more to the flow graph in figure (b), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has **four**regions.

2. *V*(*G*) = 11 edges - 9 nodes + =4.

3. *V*(*G*) = 3 predicate nodes + 1 =4.

Therefore, the cyclomatic complexity of the flow graph in figure (b) is 4.

Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code.

The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flowgraph.

2. Determine the cyclomatic complexity of the resultant flowgraph.

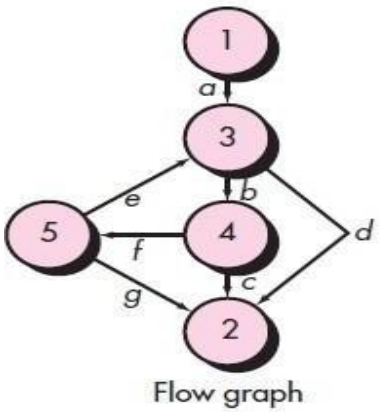
3. Determine a basis set of linearly independentpaths.

4. Prepare test cases that will force execution of each path in the basisset.

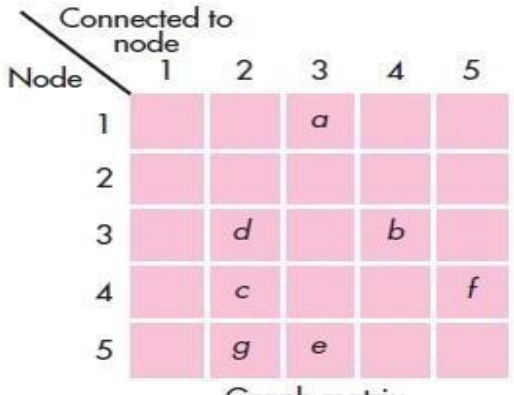
Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths isamenable to mechanization. A data structure, called a ***graph matrix***, can be quite useful for developing a software tool that assists in basis path testing.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) 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 (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix is shown in following figure.



Flow graph



Graph matrix

Fig : Graph Matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*. To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a ***link weight*** to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure duringtesting.

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will beexecute.
- The processing time expended during traversal of alink
- The memory required during traversal of alink
- The resources required during traversal of alink.

CONTROL STRUCTURE TESTING

These broaden control structure testing coverage and improve the quality of white-box testing.

Condition Testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

***E*1 <relational-operator>*E*2**

where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following:

<,<=, =, ≠,> or >=. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing

method focuses on testing each condition in the program to ensure that it does not contain errors.

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. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.

Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software.

Loop testing is a **white-box testing technique** that focuses exclusively on the validity of loop constructs. **Four** different classes of loops can be defined: **simple loops, concatenated loops, nested loops, and unstructured loops** (shown in figure).

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loopentirely.
2. Only one pass through theloop.
3. Two passes through theloop.
4. m passes through the loop where m <n.
5. n - 1, n, n + 1 passes through theloop.

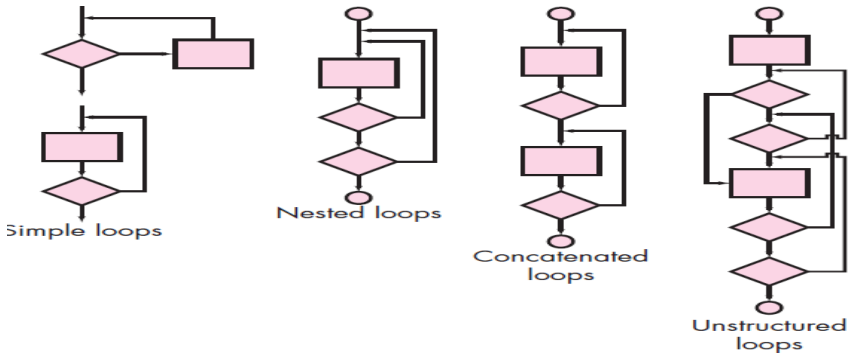


Fig : Classes of Loops

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer suggests an approach that will help to reduce the number oftests:

1. Start at the innermost loop. Set all other loops to minimumvalues.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excludedvalues.

3. Work outward, conducting tests for the next loop, but keeping all other outer loopsat minimum values and other nested loops to "typical"values.

4. Continue until all loops have beentested.

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. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

BLACK-BOX TESTING

Black-box testing, also called ***behavioral testing***, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. 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) behavior or performance errors, and (5) initialization and termination errors.

Tests are designed to answer the following questions:

- How is functional validitytested?
- How are system behavior and performancetested?
- What classes of input will make good testcases?
- Is the system particularly sensitive to certain inputvalues?
- How are the boundaries of a data classisolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on systemoperation?

By applying black-box techniques, you derive a set of test cases that satisfy the

followingcriteria

(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, and (2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test athand.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another". Stated in another way, 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 object and relationship is exercised and errors areuncovered.

To accomplish these steps, you begin by creating a ***graph***, it is a collection of ***nodes*** that represent objects, ***links*** that represent the relationships between objects, ***node weights*** that describe the properties of a node, and ***link weights*** that describe some characteristic of a link.

The symbolic representation of a graph is shown in following figure. Nodes are represented as circles connected by links that take a number of different forms.

A ***directed link*** (represented by an arrow) indicates that a relationship moves in only one direction. A ***bidirectional link***, also called a ***symmetric link***, implies that the relationship applies in both directions. ***Parallel links*** are used when a number of different relationships are established between graphnodes.

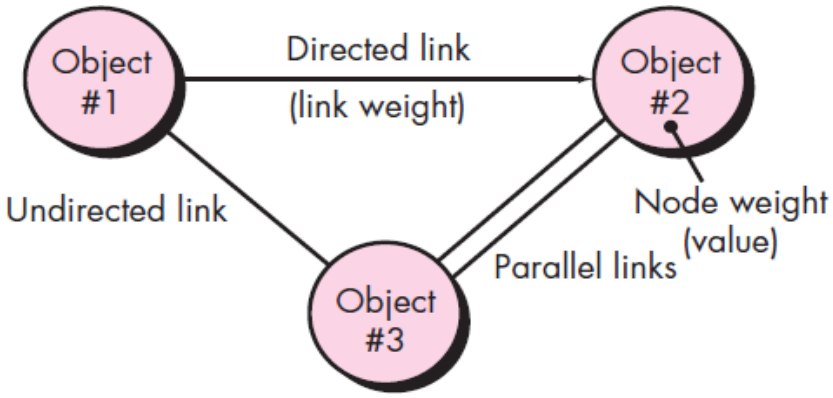


Fig : Graph Notation

Beizer describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction, and the links represent the logical connection between steps .

Finite state modeling. The nodes represent different user-observable states of the software, and the links represent the transitions that occur to move from state to state. The state diagram can be used to assist in creating graphs of thistype.

Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another.

Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

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. Test-case design for equivalence partitioning is based on an evaluation of ***equivalence classes*** for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalenceclasses aredefined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes aredefined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class aredefined.

4. If an input condition is Boolean, one valid and one invalid class aredefined.

Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that ***boundary value analysis*** **(BVA)** has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise boundingvalues.

Boundary value analysis 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:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and*b.*

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are alsotested.

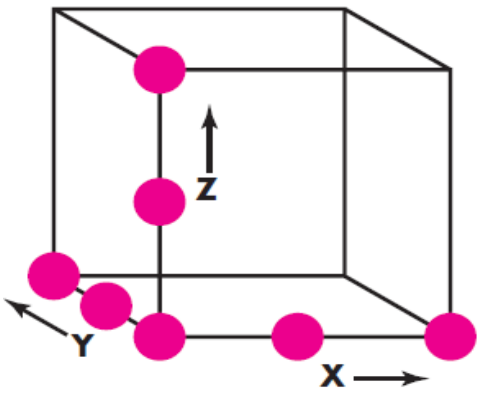
3. Apply guidelines 1 and 2 to outputconditions.

4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree.

Orthogonal Array Testing

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding ***region faults***—an error category associated with faulty logic within a softwarecomponent.

Orthogonal array testing enables you to design test cases that provide maximumtest coverage with a reasonable number of testcases



One input item at a time

MODEL-BASED TESTING

Model-based testing **(MBT)** is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases. In many cases, the model- based testing technique uses UML state diagrams, an element of the

behavioral model, as the basis for the design of test cases.

The MBT technique requires **five** steps:

1. **Analyze an existing behavioral model for the software or create one.** Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the steps (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the systemand (5) review the behavioral model to verify accuracy and consistency.
2. **Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.** The inputs will trigger events that will cause the transition tooccur.
3. **Review the behavioral model and note the expected outputs as the software makes the transition from state to state.** Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created.
4. **Execute the test cases.** Tests can be executed manually or a test script can be created and executed using a testingtool.
5. **Compare actual and expected results and take corrective action as required.**

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-drivenapplications.