Paper – II (Software Engineering)

**Paper - II (Software Engineering)**

**Unit – III**

**3.1 Object Oriented Design**

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

**The different terms related to object design are:**

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.

2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials. 5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

M.Sc. Semester – III (Computer Science) 1

Paper – II (Software Engineering)

7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

**3.2 Performing User Interface Design**

The blueprint for a house (its architectural design) is not complete without a representation of doors, windows, and utility connections for water , electricity, and telephone (not to mention cable TV). The doors, windows, and utility connections" for computer software make up the interface design of a system. Interface design focuses on three areas of concern:

1) the design of interfaces between software components,

2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and

3) the design of the interface between a human (i.e., the user) and the computer. We focus on the third interface design category—user interface design. The problems to which Shneiderman alludes are real. It is true that graphical user interfaces, windows, icons, and mouse picks have eliminated many of the most horrific interface problems. But even in a "Windows world," we all have encountered user interfaces that are difficult to learn, hard to use, confusing, counterintuitive, un - forgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely. User interface design has as much to do with the study of people as it does with technology issues. Who is the user? How does the user learn to interact with a new computer-based system? How does the user interpret information produced by the system? What will the user expect of the system? These are only a few of the many questions that must be asked and answered as part of user interface design.

**3.2.1 The Golden Rules**

Interface design, Theo Mandel coins three "golden rules”:

1. Place the user in control.

2. Reduce the user's memory load.

3. Make the interface consistent.

M.Sc. Semester – III (Computer Science) 2

Paper – II (Software Engineering)

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

**3.2.1.1 Place the User in Control**

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the windows-oriented graphical interface. "What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before 1 need to do it and makes it very easy for me to get it done. That's all, just that."

My first reaction was to shake my head and smile, but 1 paused for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom? In many cases, the designer might introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use.

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 inter- face. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

**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, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard commands (or voice input) to draw a complex shape.

**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.

M.Sc. Semester – III (Computer Science) 3

Paper – II (Software Engineering)

**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. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

**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. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

**3.2.1.2 Reduce the User’s Memory Load**

The more a user has to remember, the more error-prone 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 [MAN97] 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 and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

**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 default values.

**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 be invoked).

M.Sc. Semester – III (Computer Science) 4

Paper – II (Software Engineering)

**The visual layout of the interface should be based on a real world metaphor.** For example, a bill payment system should use a check book 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 mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a text style menu. However, every underlining capability is not listed. The user must pick underlining, and then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

**3.2.1.3. 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 a design standard that is 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 and implemented. 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. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

**Maintain consistency across a family of applications.** A set of applications (or products) 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 she encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion. The interface

M.Sc. Semester – III (Computer Science) 5

Paper – II (Software Engineering)

design principles discussed in this and the preceding sections provide basic guidance for a software engineer.

**3.3 User Interface Design and Analysis**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). The human- and computer-oriented tasks that are required to achieve system function are then delineated; design issues that apply to all interface designs are considered; tools are used to prototype and ultimately implement the design model; and the result is evaluated by end-users for quality. Interface Analysis and Design Models

**3.3.1 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 a implementation model. Unfortunately, each of these models may differ significantly. The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile offend-users of the system. To build an effective user interface, "ail design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality", In addition, users can be categorized as Novices. No syntactic knowledge 1 of the system and little semantic knowledge 2 of the application or computer usage in general. 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 of- ten leads to the "power-user syndrome," that is, individuals who look for shortcuts and abbreviated modes of interaction. A design model of the entire system incorporates data, architectural, interface, and procedural representations of the software. The requirements specification may establish certain constraints that help define the user of the system, but the interface design is often only incidental to the design model.3 The user's mental model (system perception) is the image of the system that endusers carry in their heads. For example, if the user of a particular page layout system The

M.Sc. Semester – III (Computer Science) 6

Paper – II (Software Engineering)

implementation model combines the outward manifestation of the compute based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system 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 To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface. The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive system", In essence, 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.

**3.3.2 The Process**

The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed, the user interface analysis and design process encompasses four distinct framework activities:

1 . User, task, and environment analysis and modeling.

2. interface design.

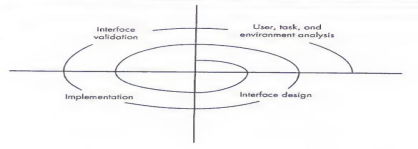
3. Interface construction (implementation).

4. interface validation.

The spiral shown figure 3.1, in implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed. 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. In essence, the software engineer attempts to understand the system perception for each class of users.

M.Sc. Semester – III (Computer Science) 7

Paper – II (Software Engineering)



**Fig. 3.1 The user interface design process**

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). The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

Where will the interface be located physically?

• Will the user be sitting, standing, or performing other tasks unrelated to the interface? • Does the interface hardware accommodate space, light, or noise constraints? • Are there special human factors considerations driven by environmental factors? The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. The construction activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, user interface development tools may be used to complete the construction of the interface. Validation focuses on

l) 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 theii work.

As we have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model)

M.Sc. Semester – III (Computer Science) 8

Paper – II (Software Engineering)

on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

**3.4 Interface Analysis**

A key tenet of all software engineering process models is this: you better 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.

In the sections that follow, we examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow. **3.4.1 User Analysis**

Earlier we noted that each user has a mental image or system perception of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that a designer can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

**User interviews:** The most direct approach, interviews involve representatives from the software team who 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 focus groups.

**Sales input:** Sales people meet with customers and 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 while providing an understanding of how each segment might use the software in subtly different ways.

**Support input:** Support staff talk with users on a daily basis, making them the most likely source of information on what works and what doesn't, what users like and what they dislike,

M.Sc. Semester – III (Computer Science) 9

Paper – II (Software Engineering)

what features generate questions, and what features are easy to use. The following set of questions will help the interface de signer better understand the users of a system: • Are users trained professionals, technicians, clerical or manufacturing workers? • What level of formal education does the average user have?

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

• Are users expert typists or keyboard phobic?

• What is the age range of the user community?

• Will the users be represented predominately by one gender?

• How are users compensated for the work they perform?

• Do users work normal office hours, or do they work until the job is done? • 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 among users?

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

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

• Do users want to know about the technology that sits behind the interface? The answers to these and similar questions will allow the designer to understand 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.

**3.4.2 Task Analysis and Modeling**

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

• What work will the user perform in specific circumstances?

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

• 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 of tasks?

To answer these questions, the software engineer must draw upon analysis techniques discussed, but in this instance, these techniques are applied to the user interface. **Use-cases** we noted that the use-case describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of

M.Sc. Semester – III (Computer Science) 10

Paper – II (Software Engineering)

task analysis, the use-case is developed to show how an end-user performs some specific work related task. In most instances, the use-case is written in an informal style (a simple paragraph) in the first-person. For example, assume that a small software company wants to build a computer aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe specific design functions. When asked "How do you decide where to put furniture in a room?" an interior designer writes the following informal use-case:

This use-case provides a basic description of one important work task for the computer aided design system. From it, the software engineer can extract tasks, objects, and the overall flow of the interaction. In addition, additional features of the system that would please the interior designer can also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through the each window.

**Task elaboration:** we discussed stepwise elaboration (also called Junctional decomposition or stepwise refinement) as a mechanism for refining the pro- cessing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate. Task analysis can be applied in two ways. As we have already noted, an interactive, computer-based system is often used to replace a manual or semi-automated activity. To understand the tasks that must be performed to accomplish the goal of the activity, a human engineer5 must understand the tasks that humans currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, the human engineer can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. . Regardless of the overall approach to task analysis, a human engineer must first define and classify tasks. We have already noted that one approach is stepwise elaboration. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. By observing an interior designer at work, the engineer notices that interior design comprises a number of major activities: furniture layout (note the use-case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the

M.Sc. Semester – III (Computer Science) 11

Paper – II (Software Engineering)

customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use-case, furniture layout can be refined into the following tasks:

1) draw a floor plan based on room dimensions;

2) place windows and doors at appropriate locations;

3a) use furniture templates to draw scaled furniture outlines on floor plan; 3b) use accent templates to draw scaled accents on floor plan.

4) move furniture outlines and accent outlines to get best placement;

5) label all furniture and accent outlines;

6) draw dimensions to show location;

7) draw perspective rendering view for customer.

A similar approach could be used for each of the other major tasks. Subtasks 1-7 can each be refined further. Subtasks 1-6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction. 6 The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

**Object elaboration:** Rather than focusing on the tasks that a user must perform, the software engineer examines the use-case and other information obtained from -the user and extracts the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide the designer with a list of operations. For example, the furniture template might translate into a class called Furniture with attributes that might include size, shape, location and others. The interior designer would select the object from the Furniture class, move it to a position on the floor plan (another object in this context), draw the furniture outline, and so forth. The tasks select, move, and draw are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

**Workflow analysis:** When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration

M.Sc. Semester – III (Computer Science) 12

Paper – II (Software Engineering)

and apply workflow analysis. This technique allows a software engineer to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process7 will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram). We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 3.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted above. This information may have been elicited via interview or from use-cases written by each actor. Regardless, the flow of events (shown in the figure) enable the interface designer to recognize three key interface characteristics:

1 . Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different from the one defined for pharmacists or physicians.

2 . The interface design for pharmacists and physicians must accommodate ac- cess to and display of information from secondary information sources (e g., access to inventory for the pharmacist and access to information about alter- native medications for the physician). 3 . Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g ..fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

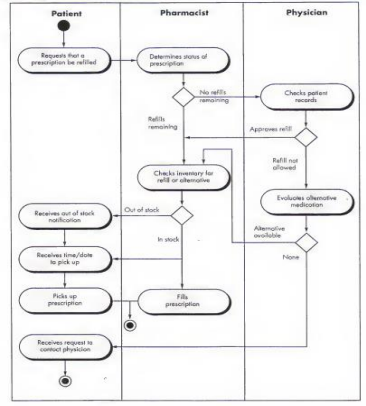
**Hierarchical representation:** As the interface is analyzed, a process of elaboration occurs. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the user task requests that a prescription be refilled. The following task hierarchy is developed: Request that a prescription be refilled

• Provide identifying information

• Specif/ name

M.Sc. Semester – III (Computer Science) 13

Paper – II (Software Engineering)



**Fig 3.2**

• Specify userid

• Specify PIN and password

• Specify prescription number

• Specify date refill is required To complete the request that a prescription be refilled tasks, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

**3.4.3 Analysis of Display Content**

The user tasks identified in the preceding section lead to the presentation of a variety of different types of content. For modern applications, display content can range from character based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture

M.Sc. Semester – III (Computer Science) 14

Paper – II (Software Engineering)

of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques identify the output data objects that are produced by an application. These data objects maybe

1) generated by components (unrelated to the interface) in other parts of the 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 (as it is displayed by the interface) are considered. Among the questions that are asked and answered are: • Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?

• Can the user customize the screen location for content?

• Is proper on-screen identification assigned to all content?

• How is a large report partitioned for ease of understanding?

• Will mechanisms be available for moving directly to summary information for large collections of data.

• Will graphical output be scaled to fit within the bounds of the display device that is used? • How will color be used to enhance understanding?

• How will error messages and warnings be presented to the user?

As each of these (and other) questions are answered, the requirements for content presentation are established.

**3.4.4 Analysis of the Work Environment**

Hackos and Redish discuss the importance of work environment analysis when they state: People 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 some applications the user interface for a computer-based system is placed in a "user-friendly location” (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit) lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use. In addition to physical environmental factors, the work place

M.Sc. Semester – III (Computer Science) 15

Paper – II (Software Engineering)

culture also comes into play. Will system interaction be measured in some manner (e.g., time per trans- action 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 design commences.

**3.5 Interface Design Steps**

Once interface analysis has been completed, all tasks (or objects and actions) required by the end-user have been identified in detail, and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, each elaborating and refining information developed in the preceding step. Although many different user interface design models have been proposed, all suggest some combination of the following steps:

1 . Using information developed during interface analysis, define interface objects and actions (operations).

2 . Define events (user actions) that will cause the state of the user interface to change. Model this behavior.

3. Depict each interface state as it will actually look to the end-user.

4. indicate how the user interprets the state of the system from information pro- vided through the interface.

In some cases, the interface designer may begin with sketches of each interface state (i.e„ what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, the designer must

1) always follow the golden rules,

2) model how the interface will be implemented, and

3) consider the environment (e.g., display technology, operating system, development tools) that will be used

**3.5.1 Applying Interface Design Steps**

An important step in interface design is the definition of interface objects and the actions that are applied to them. To accomplish this, use-cases are parsed in much the same way as described. That is, a description of 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

M.Sc. Semester – III (Computer Science) 16

Paper – II (Software Engineering)

and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon). The implication of this action is to create a hard-copy report. An application object represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction. When the designer is satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items is conducted. If a real world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor. To provide a brief illustration of the design steps noted previously, we consider a user scenario for the SafeHome system. A preliminary use-case (written by the homeowner) for the interface follows:

Based on this use-case, the following homeowner tasks, objects, and data items are identified:

• accesses the SafeHome system

• enters an ID and password to allow remote access

• checks system status

• arms or disarms SafeHome system

• displays floor plan and sensor locations

• displays zones on floor plan

• changes zones on floor plan

• displays video camera locations on floor plan

• selects video camera for viewing

• view's video images

• pans or zooms the video camera

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, video camera location (a source object) is dragged and dropped onto video camera (a target object) to create a video image (a

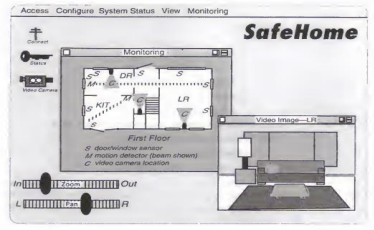
M.Sc. Semester – III (Computer Science) 17

Paper – II (Software Engineering)

window that contains the video display). A preliminary sketch of the screen layout for video monitoring is created. To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window, is selected. In this case, a camera location in the living room, UR, is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the living room (LR). The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left hand corner of the screen. The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

**3.5.2 User Interface Design Patterns**

Sophisticated graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As we noted earlier in this book, design pattern is an abstraction that prescribes a design solution to a specific, well- bounded design problem. Each of the example patterns (and all patterns within each category) presented in the sidebar would also have a complete component-level design, including design classes, attributes, operations, and interfaces.

**Fig. 3.3**

M.Sc. Semester – III (Computer Science) 18

Paper – II (Software Engineering)

**3.5.3 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 labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and customer frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

**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 (e.g., hits the return key or clicks a mouse) until the software responds with the desired output or action. System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress is the inevitable result. 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 relatively long. For example, a 1 -second response to a command will often be preferable to a response that varies from 0. 1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something '‘different" has occurred behind the scenes.

**Help facilities:** Almost every user of an interactive, computer-based system re- quires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of user manuals" may be the only option. In most cases, however, modern software provides on-line help facilities that enable a user to get a question answered or re- solve a problem without leaving the interface. 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 all functions. • How will the user request help? Options include a help menu, a special function key, or a HELP command.

• 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 screen location.

M.Sc. Semester – III (Computer Science) 19

Paper – II (Software Engineering)

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

• How will help information be structured’ Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext. **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 in- crease user frustration. There are few computer users who have not encountered an error of the form : 'Application XXX has been forced to quit because an error of type i 023 has been encountered. " Somewhere, an explanation for error 1 023 must exist;, otherwise. why would the designers have added the identification? Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system should have the following characteristics;

• The message should describe the problem in language the user can understand. . The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

• 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 the user. Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

**Menu and command labeling:** The typed command was once the most common mode of interaction between users and system software and was commonly used for applications of every

M.Sc. Semester – III (Computer Science) 20

Paper – II (Software Engineering)

type. Today, the use of window-oriented, point and pick interfaces has reduced reliance on typed commands, but many power-users continue to prefer a command-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 typed word.

• How difficult will it be to learn and remember the commands?

What can be done if a command is forgotten?

• Can commands be customized or abbreviated by the user?

• Are menu labels self-explanatory within the context of the interface?

• Are submenus consistent with the function implied by a master menu item? Conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

**Application accessibility:** As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users (and software engineers) who may be physically challenged is an imperative for moral, 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. Others provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

**Internationalization:** Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then juiy 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 specific market. A variety of internationalization guidelines are

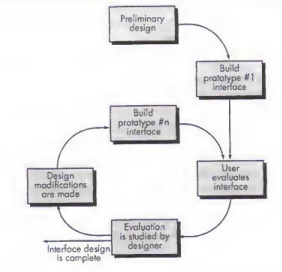
M.Sc. Semester – III (Computer Science) 21

Paper – II (Software Engineering)

available to soft- ware engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different al phabets may create specialized labeling and spacing requirements). The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundred of characters and symbols.

**3.6 Design Evaluation**

Once an operational user interface prototype has been created, 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 end-users. The user interface evaluation cycle takes the form shown in Figure 3.4. After the design model has been completed, a first-level prototype is created. The prototype is



**Fig. 3.5 The Interface design evaluation cycle**

evaluated by the user," who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further

M.Sc. Semester – III (Computer Science) 22

Paper – II (Software Engineering)

modifications to the interface design are necessary. The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If potential problems can be uncovered and corrected early, the number of loops through the evaluation cycle will be re- duced and development time will shorten. 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 written specification of the system and its interface provide an indication of the amount of learning required by users of the system.

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 the system.

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

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 the user. Once the first prototype is built, the designer 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. If quantitative data are desired, a form of time study analysis can be conducted. Users are observed during interaction, and data such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

**3.7 Testing Strategies**

**Introduction**

A strategy for software testing integrates software test case design methods into a well planned series of steps that result in the successful construction of software. The strategy

M.Sc. Semester – III (Computer Science) 23

Paper – II (Software Engineering)

provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation. A software testing strategy should be flexible enough to promote a customized testing approach.

**3.7.1 A Strategic Approach to Software Testing**

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test case design techniques and testing methods—should be defined for the software process. A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics: • Testing begins at the component level2 and works "outward" toward the integration of the entire computer-based system.

• Different testing techniques are appropriate at different points in time.

• Testing is conducted by the developer of the software and (for large projects) an independent test group.

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

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when dead- line pressure begins to rise, progress must be measurable and problems must surface as early as possible.

**3.7.1 Verification and Validation**

Software testing is one element of a broader topic that is often referred to as *verification and validation*. *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

*Verification:* "Are we building the product right?"

M.Sc. Semester – III (Computer Science) 24

Paper – II (Software Engineering)

*Validation:* "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as *software quality assurance* (SQA).

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective formal technical reviews, and solid management and measurement all lead to quality that is confirmed during testing. **3.7.1.2 Organizing for Software Testing**

There are often a number of misconceptions that can be erroneously inferred from the preceding discussion:

1) The developer of software should do no testing at all,

2) The software should be "tossed over the wall" to strangers who will test it mercilessly, 3) The testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

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. After all, personnel in the independent group team are paid to find errors. However, the software engineer doesn't turn the program over to ITG and walk away. 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.

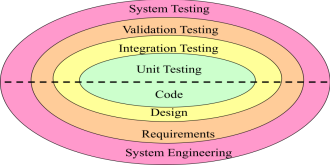
**3.7.1.3 A Software Testing Strategy**

The software engineering process may be viewed as the spiral illustrated in Figure 3.5 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, we come to

M.Sc. Semester – III (Computer Science) 25

Paper – II (Software Engineering)

design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.



**Fig. 3.5 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 (i.e., component) of the software as implemented in source code.

spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter

*validation testing,* where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally,

*system testing,* where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

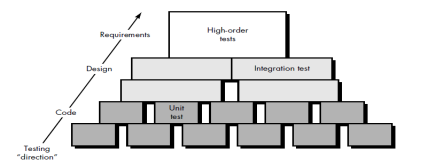
Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 3.6. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module'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. Black-box test case design techniques are the most prevalent during

M.Sc. Semester – III (Computer Science) 26

Paper – II (Software Engineering)

integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioral, and

performance requirements. Black-box testing techniques are used exclusively during validation. **Fig. 3.6 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 is achieved.

**3.8 Strategic Issues**

We explore a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb argues that the following issues must be addressed if a successful software testing strategy is to be implemented: **Specify product requirements in a quantifiable manner long before testing commences:** Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability. 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 measurable terms. For example, test effectiveness, test coverage, mean time to failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours per regression test all should be stated within the test plan.

M.Sc. Semester – III (Computer Science) 27

Paper – II (Software Engineering)

**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 the product.

**Develop a testing plan that emphasizes “rapid cycle testing.”:** Gilb recommends that a software engineering team “learn to test in rapid cycles (2 percent of project effort) of customer useful, at least field ‘trialable,’ increments of functionality and/or quality improvement.” The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

**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 regression testing.

**Use effective formal technical reviews as a filter prior to testing:** Formal technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.

**Conduct formal technical reviews to assess the test strategy and test cases themselves:** Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

**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 software testing.

**3.9 Test Strategies for Conventional Software**

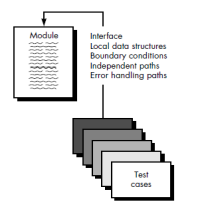
**3.9.1 Unit Testing**

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components. Unit Test Considerations The tests that occur as part of unit tests are illustrated schematically in Figure 3.6.

M.Sc. Semester – III (Computer Science) 28

Paper – II (Software Engineering)

The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.



**Fig. 3.5 Unit Test**

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing. 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. Basis path and loop testing are effective techniques for uncovering a broad array of path errors. Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence,

(2) mixed mode operations,

(3) incorrect initialization,

(4) precision inaccuracy,

(5) incorrect symbolic representation of an expression.

M.Sc. Semester – III (Computer Science) 29

Paper – II (Software Engineering)

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as (1) comparison of different data types,

(2) incorrect logical operators or precedence,

(3)expectation of equality when precision error makes equality unlikely,

(4) incorrect comparison of variables,

(5) improper or nonexistent loop termination,

(6) failure to exit when divergent iteration is encountered, and

(7) improperly modified loop variables.

Good design dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach *antibugging.* Unfortunately, there is a tendency to incorporate error handling into software and then never test it.

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.

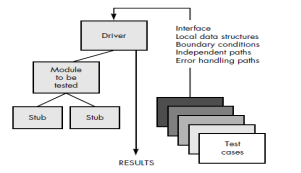
4. Exception-condition processing is incorrect.

5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. 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. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

M.Sc. Semester – III (Computer Science) 30

Paper – II (Software Engineering)



**Fig. 3.6 Unit Test Environment**

**3.9.2 Integration Testing**

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

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. **3.9.2.1 Top-down Integration**

*Top-down integration testing* is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth first manner. Referring to Figure 3.6, *depth-first integration* would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

For example, selecting the left hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) 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 (a replacement for stub S4) would be integrated first.

M.Sc. Semester – III (Computer Science) 31

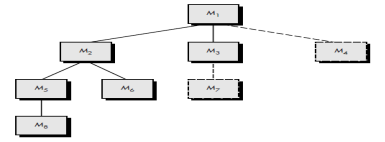
Paper – II (Software Engineering)

The next control level, M5, M6, and so on, follows. 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 control module.

**2.** Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

**3.** Tests are conducted as each component is integrated.

**4.** On completion of each set of tests, another stub is replaced with the real component. **5.** Regression testing may be conducted to ensure that new errors have not been introduced. The process continues from step 2 until the entire program structure is built. The top down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path. **Fig. 3.7 Top down Integration**

The incoming path may be integrated in a top-down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of functional capability is a confidence builder for both the developer and the customer. Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise.

The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low level modules at the

M.Sc. Semester – III (Computer Science) 32

Paper – II (Software Engineering)

beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices:

(1) delay many tests until stubs are replaced with actual modules,

(2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward. The first approach (delay tests until stubs are replaced by actual modules) causes us to loose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex.

**3.9.2.2 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, processing required for 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:

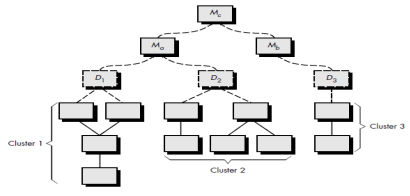
**1.** Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

**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 Figure 3.7. 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 Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

M.Sc. Semester – III (Computer Science) 33

Paper – II (Software Engineering)



**Fig. 3.8 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.

**3.9.3 Regression Testing**

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors. Regression testing may be conducted manually, by re-executing 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 software functions.

M.Sc. Semester – III (Computer Science) 34

Paper – II (Software Engineering)

• Additional tests that focus on software functions that are likely to be affected by the change. • Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred. **3.9.4 Smoke Testing**

*Smoke testing* is an integration testing approach that is commonly used when “shrink wrapped” software products are being developed. It is designed as a pacing mechanism for time critical projects, allowing the software team to assess its project on a frequent basis. 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 product functions.

**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 “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.

**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 bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time critical software engineering 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 are uncovered.

M.Sc. Semester – III (Computer Science) 35

Paper – II (Software Engineering)

• *The quality of the end-product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.

• *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 discovered error.

• *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 being made.

**3.9.5 Integration Test Documentation**

An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification.* This document contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.

The test plan describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for a CAD system might be divided into the following test phases: • User interaction (command selection, drawing creation, display representation, error processing and representation).

• Data manipulation and analysis (symbol creation, dimensioning; rotation, computation of physical properties).

• Display processing and generation (two-dimensional displays, three dimensional displays, graphs and charts).

• Database management (access, update, integrity, performance).

Each of these phases and sub phases (denoted in parentheses) delineates a broad functional category within the software and can generally be related to a specific domain of the program structure.

M.Sc. Semester – III (Computer Science) 36

Paper – II (Software Engineering)

Therefore, program builds (groups of modules) are created to correspond to each phase. The following criteria and corresponding tests are applied for all test phases: **Interface integrity** Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

**Functional validity** Tests designed to uncover functional errors are conducted. **Information content** Tests designed to uncover errors associated with local or global data structures are conducted.

**Performance** Tests designed to verify performance bounds established during software design are conducted.

**3.10 Black-Box and White-Box Testing**

White-box testing, sometimes called *glass-box testing,* is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

• *Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.* Errors tend to creep into our work when we design and implement function, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.

• *We often believe that a logical path is not likely to be executed when, in fact, It may be executed on a regular basis.* The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.

• *Typographical errors are random.* When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path. Each of these reasons provides an argument for conducting white-box tests.

M.Sc. Semester – III (Computer Science) 37

Paper – II (Software Engineering)

Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White-box testing is far more likely to uncover them.

**3.10.1 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.

**3.10.2 Flow Graph Notation**

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 3.8. Each structured construct has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design representation in Figure 3.10A. Here, a flowchart is used to depict program control structure.

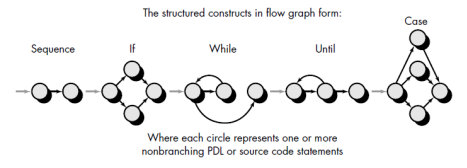
**Fig. 3.9 Flow Graph Notation**

Figure 3.10B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 3.10B, 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 (e.g., see the symbol for the if-then-else construct). Areas bounded by

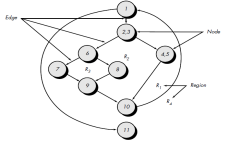
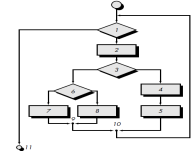
M.Sc. Semester – III (Computer Science) 38

Paper – II (Software Engineering)

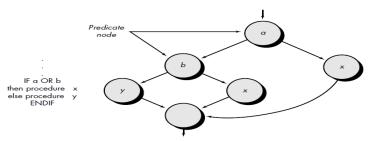
edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region. When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 3.9, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

**3.10.3 Cyclomatic Complexity**

*Cyclomatic complexity* is 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 us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once. An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow



**Fig. 3.10 Flowchart, (A) and flow graph (B)**

****M.Sc. Semester – III (Computer Science) 39

Paper – II (Software Engineering)

**Fig. 3.11 Compound logic**

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 3.10B 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 new edges. Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 2.8B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique.

In fact, a number of different basis sets can be derived for a given procedural design. How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer. Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways: **1.** The number of regions of the flow graph correspond to the cyclomatic complexity. **2.** Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is defined as

*V*(*G*) = *E* - *N* + 2

**FIGURE 3.9** Compound logic *Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. It can be used for test planning as well as test case design.*

**How is cyclomatic complexity computed?**

where *E* is the number of flow graph edges, *N* is the number of flow graph nodes. **3.** Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is also defined as

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

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

M.Sc. Semester – III (Computer Science) 40

Paper – II (Software Engineering)

Referring once more to the flow graph in Figure 3.10B, 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 + 2 = 4.

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

Therefore, the cyclomatic complexity of the flow graph in Figure 3.10B is 4. More important, the value for *V*(*G*) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

**3.10.4 Deriving Test Cases**

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 3.11, will be used as an example to illustrate each step in the test case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

**1. Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure 3.11, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 3.12.

**2. Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, *V*(*G*), is determined by applying the algorithms. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. Referring to Figure 3.12,

*V*(*G*) = 6 regions

*V*(*G*) = 17 edges \_ 13 nodes + 2 = 6

*V*(*G*) = 5 predicate nodes + 1 = 6

**3. Determine a basis set of linearly independent paths.** The value of *V*(*G*) provides the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

path 1: 1-2-10-11-13

M.Sc. Semester – III (Computer Science) 41

Paper – II (Software Engineering)

path 2: 1-2-10-12-13

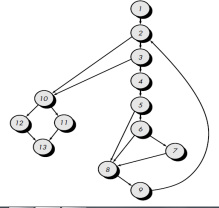
path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes. **4. Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are



**Fig. 3.12 Flow graph for the procedure average**

**Path 1 test case:**

value(*k*) = valid input, where *k* < *i* for 2 ≤ *i* ≤ 100

value(*i*) = \_999 where 2 ≤ *i* ≤ 100

*Expected results:* Correct average based on *k* values and proper totals.

*Note:* Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests. **Path 2 test case:**

value(1) = \_999

*Expected results:* Average = \_999; other totals at initial values.

**Path 3 test case:**

M.Sc. Semester – III (Computer Science) 42

Paper – II (Software Engineering)

Attempt to process 101 or more values.

First 100 values should be valid.

*Expected results:* Same as test case 1.

**Path 4 test case:**

value(*i*) = valid input where i < 100

value(*k*) < minimum where *k* < *i*

*Expected results:* Correct average based on *k* values and proper totals.

**Path 5 test case:**

value(*i*) = valid input where *i* < 100

value(*k*) > maximum where *k* <= *i*

*Expected results:* Correct average based on *n* values and proper totals.

**Path 6 test case:**

value(*i*) = valid input where *i* < 100

*Expected results:* Correct average based on *n* values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once. It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

**3.10.5 Graph Matrices**

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

A *graph matrix* is a square matrix whose size (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 Figure 3.13.

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.*

M.Sc. Semester – III (Computer Science) 43

Paper – II (Software Engineering)

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 during testing. 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 be executed.

• The processing time expended during traversal of a link.

• The memory required during traversal of a link.

• The resources required during traversal of a link.

**Fig. 3.13 Graph Matrix**

**Fig. 3.14 Connection Matrix**

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure 3.13 is redrawn as shown in Figure 3.14. Each letter has been replaced with a 1,

M.Sc. Semester – III (Computer Science) 44

Paper – II (Software Engineering)

indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a *connection matrix.* Referring to Figure 3.14 each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity .

Beizer provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

**3.10.a Black-Box Testing**

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

(4) behavior or performance errors, and

(5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

• How is functional validity tested?

• How is system behavior and performance tested?

• What classes of input will make good test cases?

• Is the system particularly sensitive to certain input values?

• How are the boundaries of a data class isolated?

• What data rates and data volume can the system tolerate?

• What effect will specific combinations of data have on system operation?

M.Sc. Semester – III (Computer Science) 45

Paper – II (Software Engineering)

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria (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 us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

**3.10.a.1 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 are uncovered.

To accomplish these steps, the software engineer begins by creating a *graph*—a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and *link weights* that describe some characteristic of a link.

**Fig. 3.15 (A) Graph notation (B) Simple example**

M.Sc. Semester – III (Computer Science) 46

Paper – II (Software Engineering)

The symbolic representation of a graph is shown in Figure 3.15A. 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 graph nodes. As a simple example, consider a portion of a graph for a word-processing application where *Object #1* = **new file menu select**

*Object #2* = **document window**

*Object #3* = **document text**

Referring to the figure, a menu select on **new file** generates a **document window.** The node weight of **document window** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **new file menu select** and **document text,** and parallel links indicate relationships between **document window** and **document text.** In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer, describes a number of behavioral testing methods that can make use of graphs:

**Transaction flow modeling:** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., **flight.information.input** is followed by *validation/availability.processing*).

The data flow diagram can be used to assist in creating graphs of this type. **Finite state modeling:** The nodes represent different user observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **order-information** is verified during *inventory- availability look-up* and is followed by **customer-billing-information input**). The state transition diagram can be used to assist in creating graphs of this type.

M.Sc. Semester – III (Computer Science) 47

Paper – II (Software Engineering)

**Data flow modeling:** The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node **FICA.tax.withheld (FTW)** is computed from **gross.wages (GW)** using the relationship, FTW = 0.62 \_ GW. **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.

However, to provide a generic outline of the graph-based testing approach. Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes. Once nodes have been identified, links and link weights should be established. In general, links should be named, although links that represent control flow between program objects need not be named. In many cases, the graph model may have loops (i.e., a path through the graph in which one or more nodes is encountered more than one time). Loop testing can also be applied at the behavioral (black-box) level. The graph will assist in identifying those loops that need to be tested.

Each relationship is studied separately so that test cases can be derived. The *transitivity* of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, **X, Y,** and **Z.** Consider the following relationships:

**X** is required to compute **Y**

**Y** is required to compute **Z**

Therefore, a transitive relationship has been established between **X** and **Z**: **X** is required to compute **Z**

Based on this transitive relationship, tests to find errors in the calculation of **Z** must consider a variety of values for both **X** and **Y.**

The *symmetry* of a relationship (graph link) is also an important guide to the design of test cases. If a link is indeed bidirectional (symmetric), it is important to test this feature. The UNDO feature in many personal computer applications implements limited symmetry. That is, UNDO allows an action to be negated after it has been completed. This should be thoroughly

M.Sc. Semester – III (Computer Science) 48

Paper – II (Software Engineering)

tested and all exceptions (i.e., places where UNDO cannot be used) should be noted. Finally, every node in the graph should have a relationship that leads back to itself; in essence, a “no action” or “null action” loop. These *reflexive* relationships should also be tested. As test case design begins, the first objective is to achieve *node coverage.* By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, *link coverage* is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present. A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked.

**3.10.a.2 Equivalence Partitioning**

*Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed. 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. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

**1.** If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.

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

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

**4.** If an input condition is *Boolean,* one valid and one invalid class are defined.

M.Sc. Semester – III (Computer Science) 49

Paper – II (Software Engineering)

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form area code— blank or three-digit number prefix—three-digit number not beginning with 0 or 1 suffix—four digit number password—six digit alphanumeric string commands—check, deposit, bill pay, and the like The input conditions associated with each data element for the banking application can be specified as area code: Input condition, *Boolean*—the area code may or may not be present. Input condition, *range*—values defined between 200 and 999, with specific exceptions. prefix: Input condition, *range*—specified value >200 Input condition, value—four-digit length password: Input condition, *Boolean*—a password may or may not be present. Input condition, *value*—six-character string.

command: Input condition, *set*—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

**3.10.a.3 Boundary Value Analysis**

For reasons that are not completely clear, a greater number of errors tends to occur 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 bounding values. 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*.

M.Sc. Semester – III (Computer Science) 50

Paper – II (Software Engineering)

**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 also tested.

**3.** Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

**4.** If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection. **3.11 Validation Testing**

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests— *validation testing*—may begin. Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the *Software Requirements Specification*—a document that describes all user-visible attributes of the software. The specification contains a section called *Validation Criteria.* Information contained in that section forms the basis for a validation testing approach.

**3.11.1 Validation Test Criteria**

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

M.Sc. Semester – III (Computer Science) 51

Paper – II (Software Engineering)

After each validation test case has been conducted, one of two possible conditions exist: 1) The function or performance characteristics conform to specification and are accepted or 2) A deviation from specification is uncovered and a *deficiency list* is created.

Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

**3.11.2 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 phase of the software life cycle. The configuration review, sometimes called an *audit.*

**3.11.3 Alpha and Beta Testing**

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field. 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. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find. The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally 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 (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta

M.Sc. Semester – III (Computer Science) 52

Paper – II (Software Engineering)

tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

**3.12 System Testing**

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems

1) design error-handling paths that test all information coming from other elements of the system,

2) conduct a series of tests that simulate bad data or other potential errors at the software interface,

3) record the results of tests to use as "evidence" if finger-pointing does occur, and 4) participate in planning and design of system tests to ensure that software is adequately tested. 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 allocated functions. In the sections that follow, we discuss the types of system tests [BEI84] that are worthwhile for software-based systems.

**3.12.1 Recovery Testing**

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

*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

M.Sc. Semester – III (Computer Science) 53

Paper – II (Software Engineering)

itself), re-initialization, check pointing 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.

**3.12.2 Security Testing**

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain. *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer: "The system's security must, of course, be tested for invulnerability from frontal attack— but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, 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 be obtained.

**3.12.3 Stress Testing**

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?" *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,

M.Sc. Semester – III (Computer Science) 54

Paper – II (Software Engineering)

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.

Essentially, the tester attempts to break the program. A variation of stress testing is a technique called *sensitivity testing.* In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

**3.12.4 Performance Testing**

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. *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 white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

M.Sc. Semester – III (Computer Science) 55

Paper – II (Software Engineering)



**Fig. 3.16 The debugging process**

**3.13 The Art of Debugging**

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations. *Debugging* occurs as a consequence of successful testing. 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. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

**3.13.1 The Debugging Process**

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure 3.16, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes:

1) the cause will be found and corrected, or

2) the cause will not be found.

M.Sc. Semester – III (Computer Science) 56

Paper – II (Software Engineering)

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion. Why is debugging so difficult? In all likelihood, human psychology has more to do with an answer than software technology. However, 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 program structures exacerbate this situation.

**2.** The symptom may disappear (temporarily) when another error is corrected. **3.** The symptom may actually be caused by non errors (e.g., round-off inaccuracies). **4.** The symptom may be caused by human error that is not easily traced.

**5.** The symptom may be a result of timing problems, rather than processing problems. **6.** It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

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

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

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g. the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more. 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. Commenting on the human aspects of debugging, Shneiderman states:

Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected. Debugging Approaches

M.Sc. Semester – III (Computer Science) 57

Paper – II (Software Engineering)

Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck. 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. Take a simple non-software example: A lamp in my house does not work. If nothing in the house works, the cause must be in the main circuit breaker or outside; I look around to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test. In general, three categories for debugging approaches may be proposed:

(1) brute force, (2) backtracking, and (3) cause elimination.

The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error.

Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first! *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 site of the cause is found.

Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large. The third approach to debugging—*cause elimination*— is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug. Each of these debugging approaches can be supplemented with debugging tools.

M.Sc. Semester – III (Computer Science) 58

Paper – II (Software Engineering)

We can apply a wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, memory dumps, and cross-reference maps. However, tools are not a substitute for careful evaluation based on a complete software design document and clear source code.

Each of us can recall puzzling for hours or days over a persistent bug. A colleague wanders by and in desperation we explain the problem and throw opens the listing. Instantaneously (it seems), the cause of the error is uncovered. Smiling smugly, our colleague wanders off. A fresh viewpoint, unclouded by hours of frustration, can do wonders. A final maxim for debugging might be: "When all else fails, get help!"

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 every software engineer 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 other errors.

**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 is made. **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 we correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

**3.14 Product Metrics**

We use measures to better understand the attributes of the models that we create and to assess the quality of the engineered products or systems that we build. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Absolute measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Instead, we attempt to derive a set of indirect measures that lead to metrics

M.Sc. Semester – III (Computer Science) 59

Paper – II (Software Engineering)

that provide an indication of the quality of some representation of software. Because software measures and metrics are not absolute But some members of the software community continue to argue that software is unmeasurable or that attempts at measurement should be postponed until we better understand software and the attributes that should be used to describe it. This is a mistake. Although technical metrics for computer software are not absolute, they provide us with a systematic way to assess quality based on a set of clearly defined rules. They also provide the software engineer with on-the-spot, rather than after-the-fact insight. This enables the engineer to discover and correct potential problems before they become catastrophic defects.

We discussed software metrics as they are applied at the process and project level. In this chapter, our focus shifts to measures that can be used to assess the quality of the product as it is being engineered. These measures of internal product attributes provide the software engineer with a real-time indication of the efficacy of the analysis, design, and code models; the effectiveness of test cases; and the overall quality of the software to be built.

**3.14.1 Software Quality**

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *quality? W*e proposed a number of different ways to look at software quality and introduced a definition that stressed conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software. There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

**1.** Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.

**2.** There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Software quality is a complex mix of factors that will vary across different applications and the customers who request them. In the sections that follow, software quality factors are identified and the human activities required to achieve them are described.

M.Sc. Semester – III (Computer Science) 60

Paper – II (Software Engineering)

**3.14.1.1 McCall’s Quality Factors**

The factors that affect software quality can be categorized in two broad groups: 1) factors that can be directly measured (e.g., defects per function-point) and 2) factors that can be measured only indirectly (e.g., usability or maintainability).

In each case measurement must occur. We must compare the software (documents, programs, data) to some datum and arrive at an indication of quality. McCall, Richards, and Walters propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 3.17, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments. Referring to the factors noted in Figure 3.18, McCall and his colleagues provide the following descriptions:

*Correctness.* The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

*Reliability.* The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed

**Fig. 3.17 McCall’s software quality factors**

*Efficiency.* The amount of computing resources and code required by a program to perform its function.

*Integrity.* Extent to which access to software or data by unauthorized persons can be controlled. *Usability.* Effort required to learn, operate, prepare input, and interpret output of a program. *Maintainability.* Effort required to locate and fix an error in a program. [This is a very limited definition.]

*Flexibility.* Effort required to modify an operational program.

M.Sc. Semester – III (Computer Science) 61

Paper – II (Software Engineering)

*Testability.* Effort required to test a program to ensure that it performs its intended function. *Portability.* Effort required to transfer the program from one hardware and/or software system environment to another.

*Reusability.* Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs. *Interoperability.* Effort required to couple one system to another. It is difficult, and in some cases impossible, to develop direct measures of these quality factors. Therefore, a set of metrics are defined and used to develop expressions for each of the factors according to the following relationship:

*Fq* = *c*1 \_ *m*1 + *c*2 \_ *m*2 + . . . + *cn* \_ *mn*

where *Fq* is a software quality factor, *cn* are regression coefficients, *mn* are the metrics that affect the quality factor. Unfortunately, many of the metrics defined by McCall et al. can be measured only subjectively. The metrics may be in the form of a checklist that is used to "grade" specific attributes of the software. The grading scheme proposed by McCall et al. is a 0 (low) to 10 (high) scale. The following metrics are used in the grading scheme:

*Auditability.* The ease with which conformance to standards can be checked. *Accuracy.* The precision of computations and control.

*Communication commonality.* The degree to which standard interfaces, protocols, and bandwidth are used.

*Completeness.* The degree to which full implementation of required function has been achieved. *Conciseness*. The compactness of the program in terms of lines of code.

*Consistency.* The use of uniform design and documentation techniques throughout the software development project.

*Data commonality.* The use of standard data structures and types throughout the program. *Error tolerance.* The damage that occurs when the program encounters an error. *Execution efficiency.* The run-time performance of a program.

*Expandability.* The degree to which architectural, data, or procedural design can be extended. *Generality.* The breadth of potential application of program components.

*Hardware independence.* The degree to which the software is decoupled from the hardware on which it operates.

M.Sc. Semester – III (Computer Science) 62

Paper – II (Software Engineering)

*Instrumentation.* The degree to which the program monitors its own operation and identifies errors that do occur.

*Modularity.* The functional independence of program components.

*Operability.* The ease of operation of a program.

*Security.* The availability of mechanisms that control or protect programs and data. *Self-documentation.* The degree to which the source code provides meaningful documentation. *Simplicity.* The degree to which a program can be understood without difficulty. *Software system independence.* The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.

*Traceability.* The ability to trace a design representation or actual program component back to requirements.

*Training.* The degree to which the software assists in enabling new users to apply the system. The relationship between software quality factors and these metrics is shown in Figure 3.2. It should be noted that the weight given to each metric is dependent on local products and concerns.

**3.14.1.2 ISO 9126 Quality Factors**

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

*Functionality.* The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security. *Reliability.* The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

*Usability.* The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

*Efficiency.* The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

*Maintainability.* The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

M.Sc. Semester – III (Computer Science) 63

Paper – II (Software Engineering)

*Portability.* The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

The ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.

**3.14.2 Metrics For The Analysis Model**

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

**Fig. 3.18 Part of the analysis model for SafeHome software**

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics derived for project application (Chapter 4) for use in this context. These metrics examine the analysis model with the intent of predicting the “size” of the resultant system. It is likely that size and design complexity will be directly correlated. **3.14.2.1 Function-Based Metrics**

The function point metric can be used effectively as a means for predicting the size of a system that will be derived from the analysis model. To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 3.3. Referring to the figure, a data flow diagram for a function within the *SafeHome* software is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The

M.Sc. Semester – III (Computer Science) 64

Paper – II (Software Engineering)

function displays a series of prompting messages and sends appropriate control signals to various components of the security system.

The data flow diagram is evaluated to determine the key measures required for computation of the function point metric:

• number of user inputs

• number of user outputs

• number of user inquiries

• number of files

• number of external interfaces

**Fig. 3.19 Computing function points for a SafeHome function**

Three user inputs—**password, panic button,** and **activate/deactivate**—are shown in the figure along with two inquires—**zone inquiry** and **sensor inquiry.** One file (**system configuration file**) is shown. Two user outputs (**messages** and **sensor status**) and four external interfaces (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure 3.19.

The *count total* shown in Figure 3.17 must be adjusted using Equation (4-1): FP = count total - [0.65 + 0.01 -- (*Fi*)]

where count total is the sum of all FP entries obtained from Figure 3.18 and *Fi* (*i* = 1 to 14) are "complexity adjustment values." For the purposes of this example, we assume that \_ (*Fi*) is 46 (a moderately complex product). Therefore,

FP = 50 - [0.65 + (0.01 - 46)] = 56

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the *SafeHome* user interaction function.

M.Sc. Semester – III (Computer Science) 65

Paper – II (Software Engineering)

Assume that past data indicates that one FP translates into 60 lines of code (an object oriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates.

Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

**3.14.2.2 Metrics for Specification Quality**

Davis and his colleagues propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: *specificity* (lack of ambiguity), *completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision,* and *reusability.* In addition, the authors note that high-quality specifications are electronically stored, executable or at least interpretable, annotated by relative importance and stable, versioned, organized, cross-referenced, and specified at the right level of detail. Although many of these characteristics appear to be qualitative in nature, Davis et al. suggest that each can be represented using one or more metrics.

For example, we assume that there are *nr* requirements in a specification, such that *nr* = *nf* + *nnf*

where *nf* is the number of functional requirements and *nnf* is the number of nonfunctional (e.g., performance) requirements.

To determine the *specificity* (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers’ interpretation of each requirement: *Q1* = *nui*/*nr*

where *nui* is the number of requirements for which all reviewers had identical interpretations. The closer the value of *Q* to 1, the lower is the ambiguity of the specification. The *completeness* of functional requirements can be determined by computing the ratio *Q*2 = *nu*/[*ni* \_ *ns*]

where *nu* is the number of unique function requirements, *ni* is the number of inputs (stimuli) defined or implied by the specification, and *ns* is the number of states specified. The *Q*2 ratio

M.Sc. Semester – III (Computer Science) 66

Paper – II (Software Engineering)

measures the percentage of necessary functions that have been specified for a system. However, it does not address nonfunctional requirements. To incorporate these into an overall metric for completeness, we must consider the degree to which requirements have been validated: *Q*3 = *nc*/[*nc* + *nnv*]

where *nc* is the number of requirements that have been validated as correct and *nnv* is the number of requirements that have not yet been validated.

**3.14.3 Metrics for the Design Model**

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.

We examine some of the more common design metrics for computer software. Each can provide the designer with improved insight and all can help the design to evolve to a higher level of quality.

**3.14.3.1 Architectural Design Metrics**

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component. Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity.

*Structural complexity* of a module *i* is defined in the following manner:

*S(i)* = *f* 2 out*(i)* (3-1)

where *f*out*(i)* is the fan-out7 of module *i*.

*Data complexity* provides an indication of the complexity in the internal interface for a module *i* and is defined as

M.Sc. Semester – III (Computer Science) 67

Paper – II (Software Engineering)

*D(i)* = *v(i)*/[ *f*out*(i)* +1] (3-2)

where *v(i)* is the number of input and output variables that are passed to and from module *i.* Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

*C(i)* = *S(i)* + *D(i)* (19-3)

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

An earlier high-level architectural design metric proposed by Henry and Kafura [HEN81] also makes use the fan-in and fan-out. The authors define a complexity metric (applicable to call and return architectures) of the form

HKM = length*(i)* \_ [ *f*in*(i)* + *f*out*(i)*]2 (19-4)

where length*(i)* is the number of programming language statements in a module *i* and *f*in*(i)* is the fan-in of a module *i.* Henry and Kafura extend the definitions of *fan-in* and *fan-out* presented in this book to include not only the number of module control connections (module calls) but also the number of data structures from which a module *i* retrieves (fan-in) or updates (fan-out) data.

To compute HKM during design, the procedural design may be used to estimate the number of programming language statements for module *i.* Like the Card and Glass metrics noted previously, an increase in the Henry-Kafura metric leads to a greater likelihood that integration and testing effort will also increase for a module.

Fenton suggests a number of simple *morphology* (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to Figure 3.20, the following metrics can be defined:

size = *n* + *a*

**M.Sc. Semester – III (Computer Science) 68

Paper – II (Software Engineering)

**Fig. 3.20 Morphology Metrics**

where *n* is the number of nodes and *a* is the number of arcs. For the architecture shown in Figure 3.4,

size = 17 + 18 = 35

depth = the longest path from the root (top) node to a leaf node. For the architecture shown in Figure 3.20, depth = 4. width = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 3.20, width = 6. arc-to-node ratio, *r* = *a*/*n*, which measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in Figure 3.20, r = 18/17 = 1.06.

The U.S. Air Force Systems Command has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-1988, the Air Force uses information obtained from data and architectural design to derive a *design structure quality index* (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI:

*S*1 = the total number of modules defined in the program architecture.

*S*2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of *S2*).

*S*3 = the number of modules whose correct function depends on prior processing. *S*4 = the number of database items (includes data objects and all attributes that define objects). *S*5 = the total number of unique database items.

*S*6 = the number of database segments (different records or individual objects). *S*7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit).

Once values *S*1 through *S*7 are determined for a computer program, the following intermediate values can be computed:

*Program structure: D*1, where *D*1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then *D*1 = 1, otherwise *D*1 = 0.

*Module independence: D*2 = 1 \_ (*S*2/*S*1)

*Modules not dependent on prior processing: D*3 = 1 \_ (*S*3/*S*1)

M.Sc. Semester – III (Computer Science) 69

Paper – II (Software Engineering)

*Database size: D*4 = 1 \_ (*S*5/*S*4)

Database compartmentalization: *D*5 = 1 \_ (*S*6/*S*4)

Module entrance/exit characteristic: *D*6 = 1 \_ (*S*7/*S*1)

With these intermediate values determined, the DSQI is computed in the following manner: DSQI = \_ *wiDi* (3-5)

where *i* = 1 to 6, *wi* is the relative weighting of the importance of each of the intermediate values, and \_ *wi* = 1 (if all *Di* are weighted equally, then *wi* = 0.167).

The value of DSQI for past designs can be determined and compared to a design that is currently under development. If the DSQI is significantly lower than average, further design work and review are indicated. Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

**3.14.3.2 Component-Level Design Metrics**

Component-level design metrics focus on internal characteristics of a software component and include measures of the “three Cs”—module cohesion, coupling, and complexity. These measures can help a software engineer to judge the quality of a component level design. The metrics presented in this section are glass box in the sense that they require knowledge of the inner working of the module under consideration. Component-level design metrics may be applied once a procedural design has been developed. Alternatively, they may be delayed until source code is available.

**Cohesion metrics** Bieman and Ott define a collection of metrics that provide an indication of the cohesiveness of a module. The metrics are defined in terms of five concepts and measures: *Data slice.* Stated simply, a data slice is a backward walk through a module that looks for data values that affect the module location at which the walk began. It should be noted that both program slices (which focus on statements and conditions) and data slices can be defined. *Data tokens.* The variables defined for a module can be defined as data tokens for the module. *Glue tokens.* This set of data tokens lies on one or more data slice.

*Superglue tokens.* These data tokens are common to every data slice in a module. *Stickiness.* The relative stickiness of a glue token is directly proportional to the number of data slices that it binds. Bieman and Ott develop metrics for *strong functional cohesion* (SFC), *weak functional cohesion* (WFC), and *adhesiveness* (the relative degree to which glue tokens bind data slices together). These metrics can be interpreted in the following manner:

M.Sc. Semester – III (Computer Science) 70

Paper – II (Software Engineering)

All of these cohesion metrics range in value between 0 and 1. They have a value of 0 when a procedure has more than one output and exhibits none of the cohesion attribute indicated by a particular metric. A procedure with no superglue tokens, no tokens that are common to all data slices, has zero strong functional cohesion—there are no data tokens that contribute to all outputs. A procedure with no glue tokens, that is no tokens common to more than one data slice (in procedures with more than one data slice), exhibits zero weak functional cohesion and zero adhesiveness—there are no data tokens that contribute to more than one output. Strong functional cohesion and adhesiveness are encountered when the Bieman and Ott metrics take on a maximum value of 1.

A detailed discussion of the Bieman and Ott metrics is best left to the authors. However, to illustrate the character of these metrics, consider the metric for strong functional cohesion: SFC(*i*) = SG [SA(*i*))/(tokens(*i*)] (3-6)

where SG[SA(*i*)] denotes superglue tokens—the set of data tokens that lie on all data slices for a module *i.* As the ratio of superglue tokens to the total number of tokens in a module *i* increases toward a maximum value of 1, the functional cohesiveness of the module also increases.

It is possible to compute measures of the functional independence— coupling and cohesion—of a component and to use these to assess the quality of the design. **Coupling metrics** Module coupling provides an indication of the “connectedness” of a module to other modules, global data, and the outside environment. Coupling was discussed in qualitative terms. Dhama has proposed a metric for module coupling that encompasses data and control flow coupling, global coupling, and environmental coupling. The

measures required to compute module coupling are defined in terms of each of the three coupling types noted previously. *For data and control flow coupling,*

*di* = number of input data parameters

*ci* = number of input control parameters

*do* = number of output data parameters

*co* = number of output control parameters *For global coupling,*

*gd* = number of global variables used as data

*gc* = number of global variables used as control *For environmental coupling, w* = number of modules called (fan-out)

*r* = number of modules calling the module under consideration (fan-in)

M.Sc. Semester – III (Computer Science) 71

Paper – II (Software Engineering)

Using these measures, a module coupling indicator, *mc ,* is defined in the following way: *mc* = *k*/*M*

where k = 1, a proportionality constant8 and

*M* = *di* + (*a* \_ *ci*) + *do* + (*b* \_ *co*) + *gd* + (*c* \_ *gc*) + *w* + *r*

where *a* = *b* = *c* = 2.

The higher the value of *mc*, the lower is the overall module coupling. For example, if a module has single input and output data parameters, accesses no global data, and is called by a single module,

*mc* = 1/(1 + 0 + 1+ 0 + 0 + + 0 + 1 + 0) = 1/3 = 0.33

We would expect that such a module exhibits low coupling. Hence, a value of *mc* = 0.33 implies low coupling. Alternatively, if a module has five input and five output data parameters, an equal number of control parameters, accesses ten items of global data, has a fan-in of 3 and a fan-out of 4,

*mc* = 1/[5 + (2 \_ 5) + 5 + (2 \_ 5) + 10 + 0 + 3 + 4] = 0.02

and the implied coupling would be high. In order to have the coupling metric move upward as the degree of coupling increases, a revised coupling metric may be defined as *C* = 1 \_ *mc*

where the degree of coupling increases nonlinearly between a minimum value in the range 0.66 to a maximum value that approaches 1.0.

**3.14.3.3 Complexity metrics.** A variety of software metrics can be computed to determine the complexity of program control flow. Many of these are based on the flow graph. A graph is a representation composed of nodes and links (also called *edges*). When the links (edges) are *directed,* the flow graph is a directed graph.

McCabe and Watson identify a number of important uses for complexity metrics: Complexity metrics can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of source code [or procedural design information]. Complexity metrics also provide feedback during the software project to help control the [design activity]. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

The most widely used (and debated) complexity metric for computer software is cyclomatic complexity, originally developed by Thomas McCabe.

M.Sc. Semester – III (Computer Science) 72

Paper – II (Software Engineering)

The McCabe metric provides a quantitative measure of testing difficulty and an indication of ultimate reliability. Experimental studies indicate distinct relationships between the McCabe metric and the number of errors existing in source code, as well as time required to find and correct such errors.

McCabe also contends that cyclomatic complexity may be used to provide a quantitative indication of maximum module size. Collecting data from a number of actual programming projects, he has found that cyclomatic complexity = 10 appears to be a practical upper limit for module size. When the cyclomatic complexity of modules exceeded this number, it became extremely difficult to adequately test a module.

Zuse presents an encyclopedic discussion of no fewer that 18 different categories of software complexity metrics. The author presents the basic definitions for metrics in each category (e.g., there are a number of variations on the cyclomatic complexity metric) and then analyzes and critiques each. Zuse’s work is the most comprehensive published to date.

**3.14.3.4 Interface Design Metrics**

Although there is significant literature on the design of human/computer interfaces relatively little information has been published on metrics that would provide insight into the quality and usability of the interface. Sears suggests that *layout appropriateness* (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses *layout entities*— graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the “cost” of the transition from one layout entity to the next all contribute to the appropriateness of the interface. For a specific layout (i.e., a specific GUI design), cost can be assigned to each sequence of actions according to the following relationship:

cost = \_ [frequency of transition(*k*) \_ cost of transition(*k*)] (3-7)

where *k* is a specific transition from one layout entity to the next as a specific task is accomplished. The summation occurs across all transitions for a particular task or set of tasks required to accomplish some application function. Cost may be characterized in terms of time, processing delay, or any other reasonable value, such as the distance that a mouse must travel between layout entities. Layout appropriateness is defined as

LA = 100 \_ [(cost of LA \_ optimal layout)/(cost of proposed layout)] (3-8)

M.Sc. Semester – III (Computer Science) 73

Paper – II (Software Engineering)

where LA = 100 for an optimal layout.

To compute the optimal layout for a GUI, interface real estate is divided into a grid. Each square of the grid represents a possible position for a layout entity. For a grid with *N* possible positions and *K* different layout entities to place, the number of possible layouts is represented in the following manner: number of possible layouts = [*N*!/(*K*! \_ (*N* \_ *K*)!] \_ *K*! (3-9)

As the number of layout positions increases, the number of possible layouts grows very large. To find the optimal (lowest cost) layout, Sears proposes a tree searching algorithm. LA is used to assess different proposed GUI layouts and the sensitivity of a particular layout to changes in task descriptions (i.e., changes in the sequence and/or frequency of transitions). The interface designer can use the change in layout appropriateness, LA, as a guide in choosing the best GUI layout for a particular application. It is important to note that the selection of a GUI design can be guided with metrics such as LA, but the final arbiter should be user input based on GUI prototypes. Nielsen and Levy report that “one has a reasonably large chance of success if one chooses between interface [designs] based solely on users’ opinions. Users’ average task performance and their subjective satisfaction with a GUI are highly correlated.”

**3.15 Metrics for Source Code**

Halstead's theory of software science is one of "the best known and most thoroughly studied . . . composite measures of (software) complexity". Software science proposed the first analytical "laws" for computer software. *Software science* assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

*n*1 = the number of distinct operators that appear in a program.

*n*2 = the number of distinct operands that appear in a program.

*N*1 = the total number of operator occurrences.

*N*2 = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall *program length, potential minimum volume* for an algorithm, the *actual volume* (number of bits required to specify a program), the *program level* (a measure of software complexity), the *language level* (a constant for a given language), and other features such as development effort, development

M.Sc. Semester – III (Computer Science) 74

Paper – II (Software Engineering)

time, and even the projected number of faults in the software. Halstead shows that length *N* can be estimated

*N* = *n*1 log 2 *n*1 + *n*2 log 2 *n*2 (19-10)

and program volume may be defined

*V* = *N* log 2 (*n*1 + *n*2) (19-11)

It should be noted that *V* will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio *L* as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, *L* must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

*L* = 2/*n*1 \_ *n*2/*N2* (19-12)

Halstead's work is amenable to experimental verification and a large body of research has been conducted to investigate software science. A discussion of this work is beyond the scope of this text, but it can be said that good agreement has been found between analytically predicted and experimental results.

**3.15.1 Metrics for Testing**

Although much has been written on software metrics for testing, the majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

Function-based metrics can be used as a predictor for overall testing effort. Various project-level characteristics (e.g., testing effort and time, errors uncovered, number of test cases produced) for past projects can be collected and correlated with the number of FP produced by a project team. The team can then project “expected values” of these characteristics for the current project. The bang metric can provide an indication of the number of test cases required by examining the primitive measures. The number of functional primitives (FuP), data elements (DE), objects (OB), relationships (RE), states (ST), and transitions (TR) can be used to project the number and types of black-box and white-box tests for the software. For example, the number of tests associated with the human/computer interface can be estimated by

M.Sc. Semester – III (Computer Science) 75

Paper – II (Software Engineering)

1) examining the number of transitions (TR) contained in the state transition representation of the HCI and evaluating the tests required to exercise each transition;

2) examining the number of data objects (OB) that move across the interface, and 3) the number of data elements that are input or output.

Architectural design metrics provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing, a test case design method. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, the tester should expend above average effort to uncover errors in such modules before they are integrated in a system. Testing effort can also be estimated using metrics derived from Halstead measures. Using the definitions for program volume, *V*, and program level, PL, software science effort, *e*, can be computed as

PL = 1/[(*n*1/2)•(*N*2/*n*2)] (3-13a)

*e* = *V*/PL (3-13b)

Testing metrics fall into two broad categories:

(1) metrics that attempt to predict the likely number of tests required at various testing levels and (2) metrics that focus on test coverage for a given component.

The percentage of overall testing effort to be allocated to a module *k* can be estimated using the following relationship:

percentage of testing effort (*k*) = e(*k*)/ \_ e(*i*) (3-14)

where e(*k*) is computed for module *k* using Equations (3-13) and the summation in the denominator of Equation (3-14) is the sum of software science effort across all modules of the system. As tests are conducted, three different measures provide an indication of testing completeness. A measure of the breath of testing provides an indication of how many requirements (of the total number of requirements) have been tested. This provides an indication of the completeness of the test plan. Depth of testing is a measure of the percentage of independent basis paths covered by testing versus the total number of basis paths in the program. A reasonably accurate estimate of the number of basis paths can be computed by adding the cyclomatic complexity of all program modules. Finally, as tests are conducted and error data are

M.Sc. Semester – III (Computer Science) 76

Paper – II (Software Engineering)

collected, fault profiles may be used to rank and categorize errors uncovered. Priority indicates the severity of the problem. Fault categories provide a description of an error so that statistical error analysis can be conducted.

**3.16 Metrics for Maintenance**

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed. IEEE Std. 982.1-1988 suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

*MT* = the number of modules in the current release

*Fc* = the number of modules in the current release that have been changed *Fa* = the number of modules in the current release that have been added

*Fd* = the number of modules from the preceding release that were deleted in the current release The software maturity index is computed in the following manner:

SMI = [*MT* \_ (*Fa* + *Fc* + *Fd*)]/*MT* (19-15)

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI and empirical models for maintenance effort can be developed.

---------------------------------------------------------------------------------------------------------------------

M.Sc. Semester – III (Computer Science) 77