Software engineering

3rd Class

University of Technology
Computer Science Department
Software Engineering 3rd Class
Lecturer Yossra Hussain

An introduction to Software Engendering

Topics:

1.1 The Computer Software

It is the product that software engineers design and build. It encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text but also includes representations of pictorial, video, and audio information.

Software engineers built it, and virtually everyone in the industrialized world uses it either directly or indirectly.

When you built computer software like you built any successful product, by applying a process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

The software might take the following forms:

1. Instructions: Computer programs, that when executed provide desired function and performance.
2. Data structured: That enable the programs to adequately manipulate information.
3. Documents: That describes the operation and use of programs.

1.2 Software Engineering

As software engineers, we use our knowledge of computer-and computing to help solve problems. Often the problem with which we are dealing is related to a computer or an existing computer system, but sometimes the difficulties underlying the problem have nothing to do with computers. Therefore, it is essential that we first understand the nature of the problem. In particular, we must be very careful not to impose computing machinery on every problem that comes our way. We must solve the problem first. Then, if need be, we can use technology as a tool to implement our solution.

Solving problems:

Most problems are large and sometimes tricky to handle, especially if they represent something new that has never been solved before. So we must begin investigating it by :

a) Analyzing: Breaking the problem into pieces that we can understand and try to deal with. We can thus describe the larger problem as a collection of small problems and their interrelationships.

Figure (1.1) illustrates how analysis works. It is important to remember that the relationships (the arrows in the figure, and the relative position of the subproblems) are as essential as the subproblems themselves.

b) Synthesis: construct our solution from components that address the problem's various aspects, (putting together of a large structure from small building blocks). Figure (1.2) illustrates this reverse process.

Software engineers use's tools techniques, procedures and paradigms to enhance the quality of their software products. Figure (1.3) illustrates the relationship between computer science and software engineering.

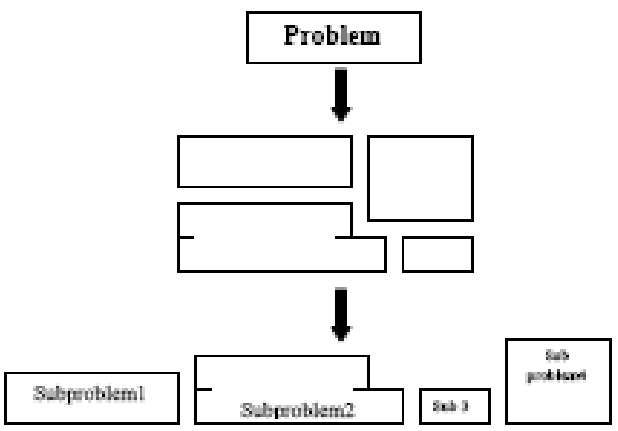


Figure (1.1): The process of analysis

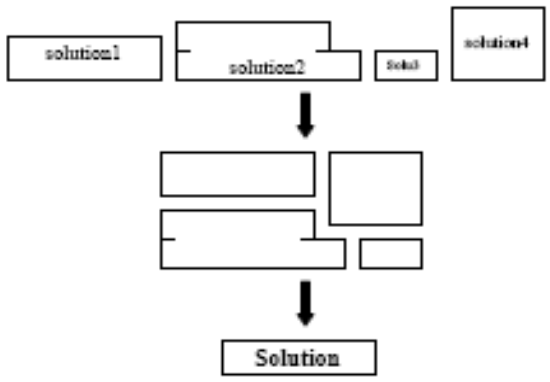


Figure (1.2): The process of synthesis

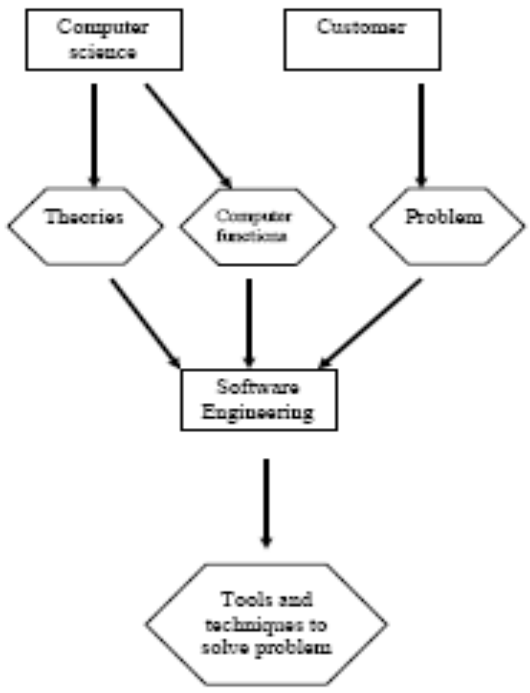


Figure (1.3): The relationship between computer science and software engineering

1.3 The characteristic of software engineer

1- Good programmer and fluent in one or more programming language.
2- Well versed data structure and approaches.
3- Familiar with several designs approaches.
4- Be able to translate vague (not clear) requirements and desires into precise specification.
5- Be able to converse with the user of the system in terms of application not in “computer”.
6- Able to a build a model. The model is used to answer questions about the system behavior and its performance.
7- Communication skills and interpersonal skills.

1.4 The Evolving Role of Software

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product.

1- As a product: it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer (producing, managing, acquiring, modifying, displaying, or transmitting) information that can be as simple as a single bit or as complex as a multimedia presentation.

2- As the vehicle used to deliver the product: software acts as the basis for the :

a. control of the computer (operating systems).

b. The communication of information (networks).

c. The creation and control of other programs (software tools and environments).

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can

produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

1.5 Software Characteristics

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

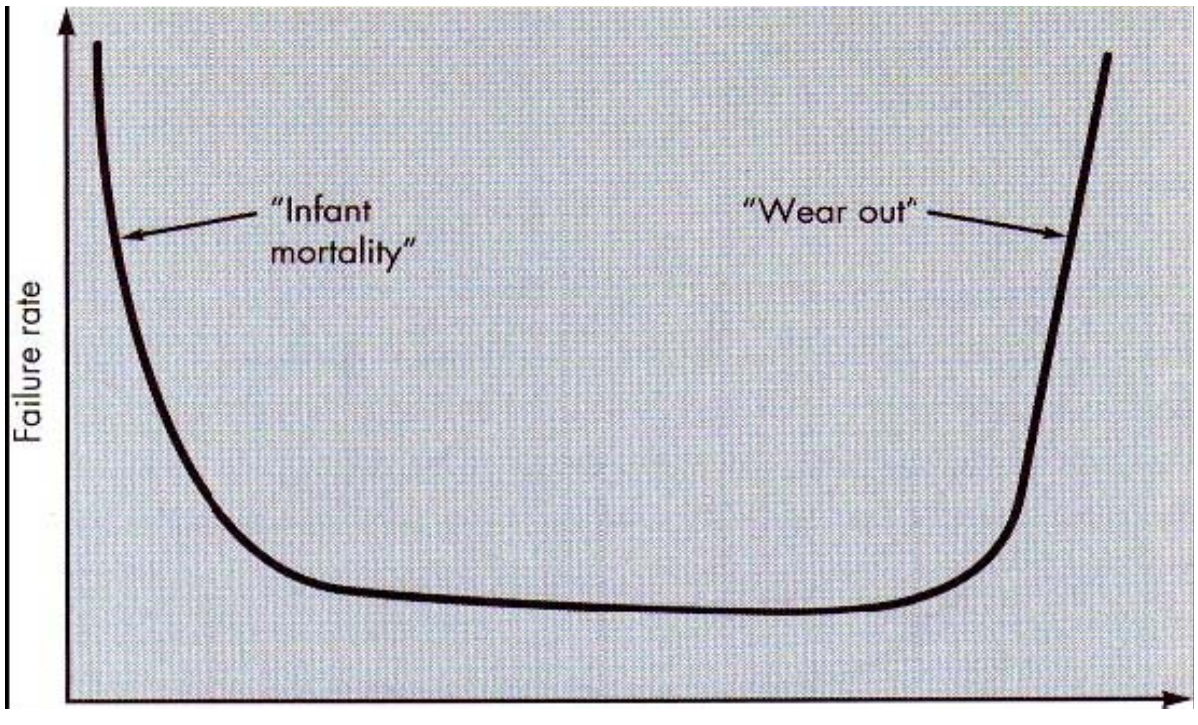


Figure (1.4): Failure curve for hardware

2. Software doesn't "wear out."

Figure (l .4) depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.5.

Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear-software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering the "actual curve" shown in Figure 1.5. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.5. Before the curve can return to the original steady state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise-the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

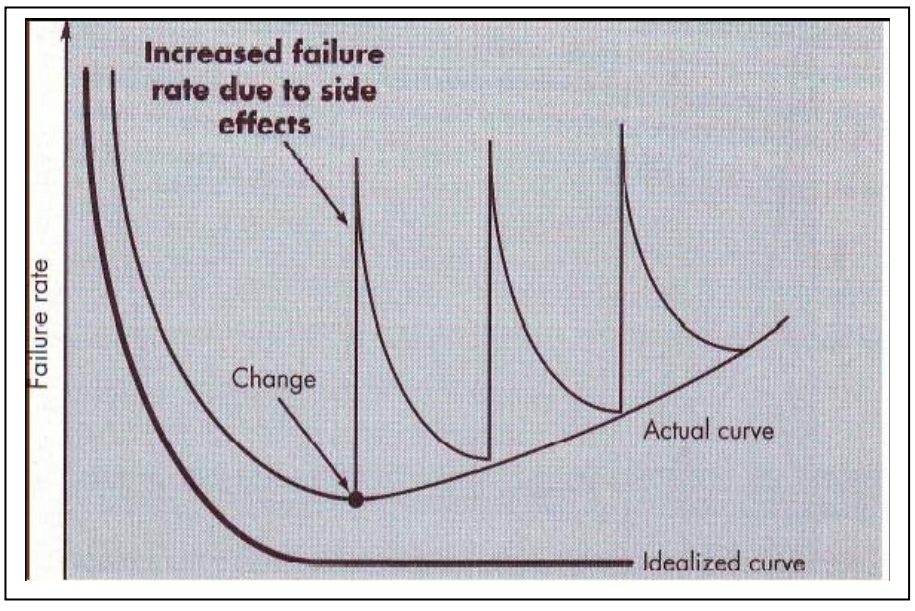


Figure (1.5): Idealized and actual failure curves for software

3. Although the industry is moving toward component-based assembly, most software continues to be custom built.

In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

1.6 Software Applications

The following software areas indicate the breadth of potential applications:

1. System software: It is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data.

2. Real-time software: Software that monitors/analyzes/controls real world events as they occur is called real time. Real-time differs from “interactive” or “time sharing“. A real-time system must respond within strict time constraints. The response time of an interactive (or time sharing) system can normally be exceeded without results.

3. Business software: Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory).

4. Engineering and scientific software: modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

5. Embedded software: Intelligent products have become commonplace in nearly every consumer and industrial market (e.g., keypad control for a microwave oven or digital functions in an automobile such as fuel control, and braking systems).

6. Personal computer software: Such as(Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management).

7. Web-based software: The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).

8. Artificial intelligence software: It makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

1.7 Software: A crisis on the horizon

Whether we call it a software crisis or affliction, the term alludes to a set of problems that are encountered in the development of computer software. The problems are not limited to software that “doesn’t function properly”. Rather, the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

1.8 The Attributes of Good Software

As well as the service which they provide software products have a number of other associated attributes which reflect the quality of that software.

These attributes are not directly concerned with what the software dose, rather they reflect its behavior which it is executing and the structure and organization of the source program and associated documentation. Examples of these attributes (some time called non-functional attributes) are the software's response time to use query and the understandability of the program code. The specific set of attributes which you might expect from a software system obviously depends on its application. Therefore a banking system must be secure, on interactive game must be responsive, a telephone switching system must be reliable, etc. these can be generated in the following attributes:

1- Maintainability: software should be written in such a way that it may evolve to meet the changing needs of customer. This is critical attribute because software change is an inevitable

2- Dependability: software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.

3- efficiency: software should not make wasteful use of system resources, such as memory and processor cycles. Therefore efficiency includes responsiveness, processing time, memory utilization etc…

4- Usability: software must be usable, without under effort by the type of user for whom it is designed. This means that it should have an appropriate userinterface and adequate documentation.

1.8 Software Lifecycle

Each software product proceeds to a number of distinct stages, these are:

• Requirements engineering

• Software design
• Software construction
• Validation and verification
• Software testing
• Software deployment
• Software maintenance

Depending the software process used for the development of the software product, these stages may occur in different orders, or frequency.

1.9.1 Requirements Engineering (requirement analysis and definition by using engineering approach)

Requirements engineering is the interface between customers and developers on a software project. Requirements should make explicit the ideas of the customer about the prospective system.

1.9.2 Software Design

The designers converts the logical software requirements from stage 1 into a technical software design by describe the software in such a way that programmers can write line of code that implement what the requirements specify.

1.9.3 Software Construction

Software construction is concerned with implementing the softwaredesign by means of programs in one or more programming languages and setting up a build management system for compiling and linking the programs.

This stage content several steps, these are :

a. Software reuse

1. Component based software engineering
2. Software product lines

b. Security and reliability

c. Software documentation

d. Coding standards

a. Software Reuse

The goal of software engineering is to achieve many features with little effort and few defects. Software reuse is believed to play an important role in achieving this goal by encapsulating effort in units of source code, which can be reused in other projects. However, the effort needed to make something reusable may not be worth it, if it is only reused few times, or needs extensive adaptation for each reuse.

a.1. Component Based Software Engineering

Building software systems from prefab software components is an old dream of software engineering.

a.2. Software Product Lines

Software systems are often part of a family of similar systems. The goal of a software product line is to maintain a set of reusable core artifacts that are common to all systems in the product line. Thus, code for a specific product can focus on the specifics of that product, reusing the common functionality.

b. Security And Reliability

Software must be dependable by making it reliable (software should work very well under any environments), secure and safety (by verifying from user authentication to using any system).

c. Software Documentation

- User documentation?
- Technical documentation?
- Documentation generation?

d. Coding Standards

Coding standards are important to ensure portability and make code maintainable by others than the original developer.

1.8.4 Validation And Verification

- Software inspection
- Software testing

1.9.4.a) Software Inspection

Software inspections are reviews of the code with the purpose of detecting defects. In an inspection someone other than the programmer reads a program unit of limited size to determine whether it satisfies the requirements and specification. A formal process and checklist are used to ensure that no aspects are forgotten.

1.9.4.b) Software Testing

Testing each unit founded in this software, follow by testing software integration.

1.9.5 Software Deployment

After development, software should be put to use. That is, it should be released and made available to users, who can then download, install, and activate it. These activities are captured under the common term *software deployment*. Richard S. Hall in the 'Software Deployment Information Clearinghouse' defines software deployment as follows: "The term software deployment refers to all of the activities that occur after a software system has been developed and made available for release. As such, software deployment includes activities such as packaging, releasing, installing, configuring, updating, and uninstalling a software system." and "Software deployment is the assembly and maintenance of the resources necessary to use a version of a system at a particular site".

The following deployment activities make up the software deployment process:

Release

Packaging

Transfer

Installation

Configuration

Activation

De-activation

Update

Adapt

De-installation

De-release

These activities are not necessarily performed sequentially. Many phases of the deployment process are often performed manually.

For example, downloading, building and installing a source distribution of a software package, requires a number of commands to be formulated and executed. Each such command requires knowledge of some sort about the activity.

Manual deployment does not scale when deploying:

many applications

applications composed from separately deployed components

on multiple machines

on different types of machines

1.9.6 Software Maintenance

As software evolves after its first release, software maintenance is needed to improve it, i.e., repair defects, and to extend it, i.e., add new functionality.

1.10 Software development

Three phases to develop the software

1- definition

2- design

3- maintenance

1- Definition

1- what information to be processed

2- What design constrains exist.

3- What function and performance desired.

4- What interfaces are desired.

5- What validation criteria are required?

6- What is modeling?

2- Design

1- How data structures to be designed.

2- How procedural details to be implemented.

3- How design to be translated into language.

4- How testing is performed.

3- Maintenance

1- error

2- Adaptation.

3- Modification

University of Technology
Computer Science Department
Software Engineering 3rd Class
Lecturer: Yossra Hussain

Chapter two

Topics

1- Software Engineering - A Layered Technology:
2- Software Process Models
   - 2.1 The Waterfall Model
   - 2.2 The Prototype Model
   - 2.3 Evolutionary Software Process Models
     - 2.2.3.a The Incremental Model
     - 2.2.3.b The Spiral Model
   - 2.4Component – Based Development

2.1 Software Engineering - A Layered Technology:

The IEEE (IEEE93) has developed a more comprehensive definition when it states:

Software Engineering: the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software engineering is a layered technology (figure 2.1). These layers are:

1- A quality focus: any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a focus on quality.

2- Process: the foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas (KPAs) that must be established for effective delivery of software engineering technology. The key process areas from the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

3- Methods: software engineering methods provide the technical "how to s" for building software methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing and maintenance. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

4- Tools: software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer - aided software engineering (CASE), is established. CASE combines software, hardware, and software engineering database (a repository containing important information about analysis, design program construction, and testing) to create a software engineering environment that is analogous to CAD/CAE (computer - aided design/engineering) for hardware.

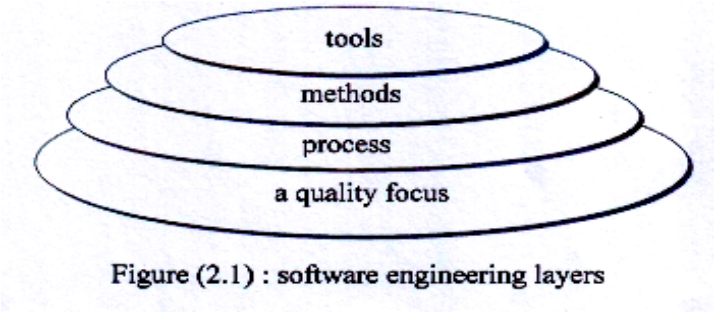


Figure (2.1) : software engineering layers

2.2 Software Process Models

There are various software development approaches defined and designed which are used/employed during development process of software, these approaches are also referred as "Software Development Process Models". Each process model follows a particular life cycle in order to ensure success in process of software development.

2.2.1 The Waterfall Model

One such approach/process used in Software Development is "The Waterfall Model". Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate process phases, these are:

1) Requirement Specifications (analysis and definition).
2) Software Design.
3) Implementation.
4) Testing.
5) Maintenance.

All these phases are cascaded to each other so that second phase is started as and when defined set of goals are achieved for first phase and it is signed off, so the name "Waterfall Model".

1) **Requirement Analysis & Definition:** All possible requirements of the system to be developed are captured in this phase. Requirements are set of functionalities and constraints that the end-user (who will be using the system) expects from the system. The requirements are gathered from the end-user by consultation, these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied. Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.

2) **System & Software Design:** Before a starting for actual coding, it is highly important to understand what we are going to create and what it should look like? The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.
3) **Implementation & Unit Testing:** On receiving system design documents, the work is divided in modules/units and actual coding is started. The system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality; this is referred to as Unit Testing. Unit testing mainly verifies if the modules/units meet their specifications.
4) **Integration & System Testing:** As specified above, the system is first divided in units which are developed and tested for their functionalities. These units are integrated into a complete system during Integration phase and tested to check if all modules/units coordinate between each other and the system as a whole behaves as per the specifications. After successfully testing the software, it is delivered to the customer.
5) **Maintenance:** Generally, problems with the system developed (which are not found during the development life cycle) come up after its practical use starts, so the issues related to the system are solved after deployment of the system. Not all the problems come in picture directly but they arise time to time and needs to be solved; hence this process is referred as Maintenance.

There are some disadvantages of the Waterfall Model, these are:

1. As it is very important to gather all possible requirements during the Requirement Gathering and Analysis phase in order to properly design the system, not all requirements are received at once, the requirements from customer goes on getting added to the list even after the end of "Requirement Gathering and Analysis" phase, this affects the system development process and its success in negative aspects.
2. The problems with one phase are never solved completely during that phase and in fact many problems regarding a particular phase arise after the phase is signed off, this results in badly structured system as not all the problems (related to a phase) are solved during the same phase.
3. The project is not partitioned in phases in flexible way.
4. As the requirements of the customer goes on getting added to the list, not all the requirements are fulfilled, this results in development of almost

unusable system. These requirements are then met in newer version of the system; this increases the cost of system development. Figure (2.2) shows the flow diagram of the waterfall model.

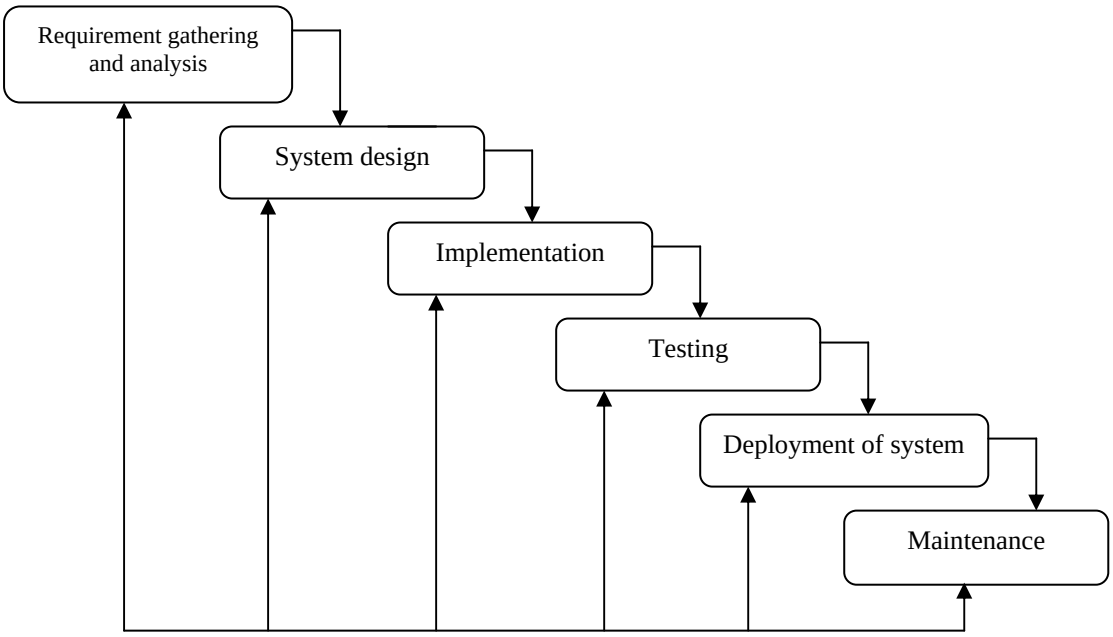


Figure (2.2): The flow diagram of the waterfall model

2.2.2 The Prototype Model

The prototype model is using for many reasons, such as:

1. A customer define a set of general objectives for software but does not identify detailed input, processing, or output requirements.
2. The developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The stages of the prototyping model:

1) Requirements gathering (listen to customer): developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats).
2) Construction of a prototype (build/revise mock-up): writing the software code depending on the quick design information.

3) Evaluation (customer test drives mock-up): the prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is turned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Yet, prototyping can also be problematic for the following reasons:

a. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire" unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

b. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability. Figure (2.3) shows the flow diagram of the prototype model.

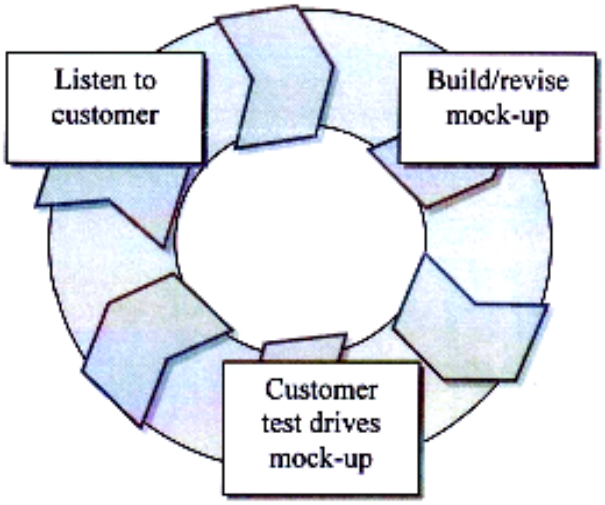


Figure (2.3): The prototyping paradigm

2.2.4 Evolutionary Software Process Models

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software. There are several types of that model, these are:

a. The Incremental Model
b. The Spiral Model
c. The WINWIN Spiral Model
d. The Concurrent Development Model

2.2.4.a The Incremental Model

The incremental model combines elements of the (linear sequential model with the iterative philosophy of prototyping). Referring to Figure 2.4, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software. For example, word-processing software developed using the incremental paradigm might:

1) Deliver basic file management, editing, and document production functions in the first increment.
2) More sophisticated editing and document production capabilities in the second increment.
3) Spelling and grammar checking in the third increment.
4) Advanced page layout capability in the fourth increment.

When an incremental model is used, the first increment is often *a core product*. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

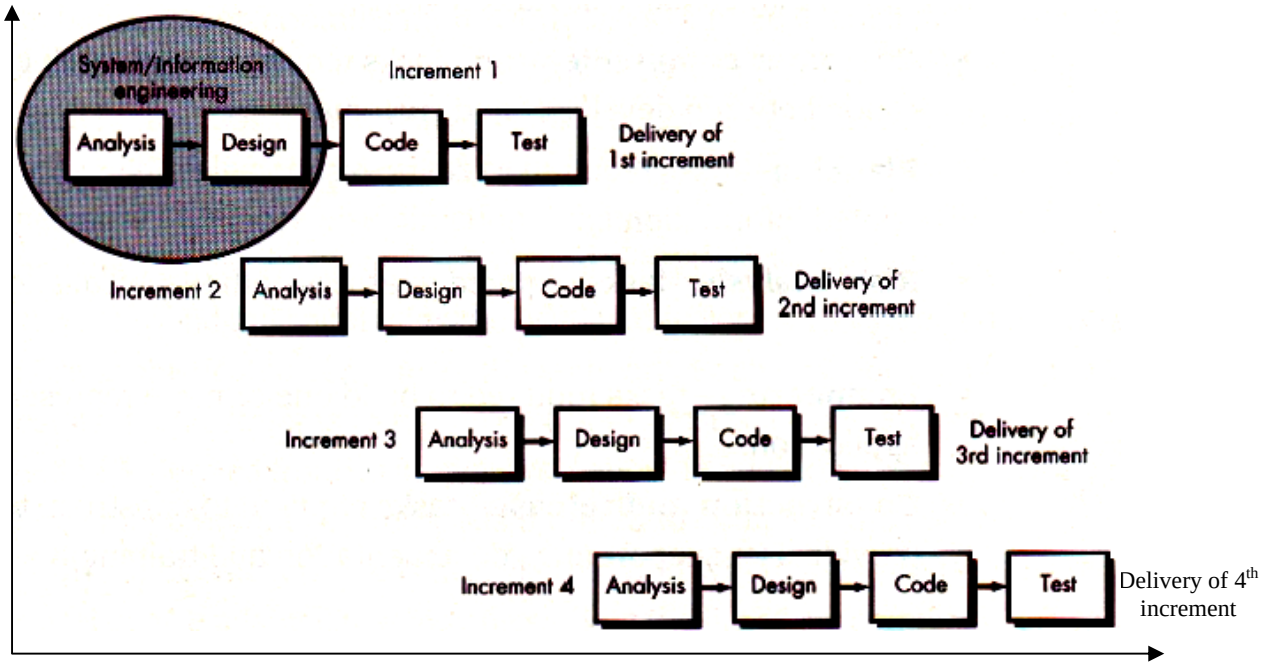


Figure (2.4): The incremental model

Incremental development is particularly useful when:

A- Staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

B- Increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

2.2.3.b The Spiral Model

Spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations,

the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

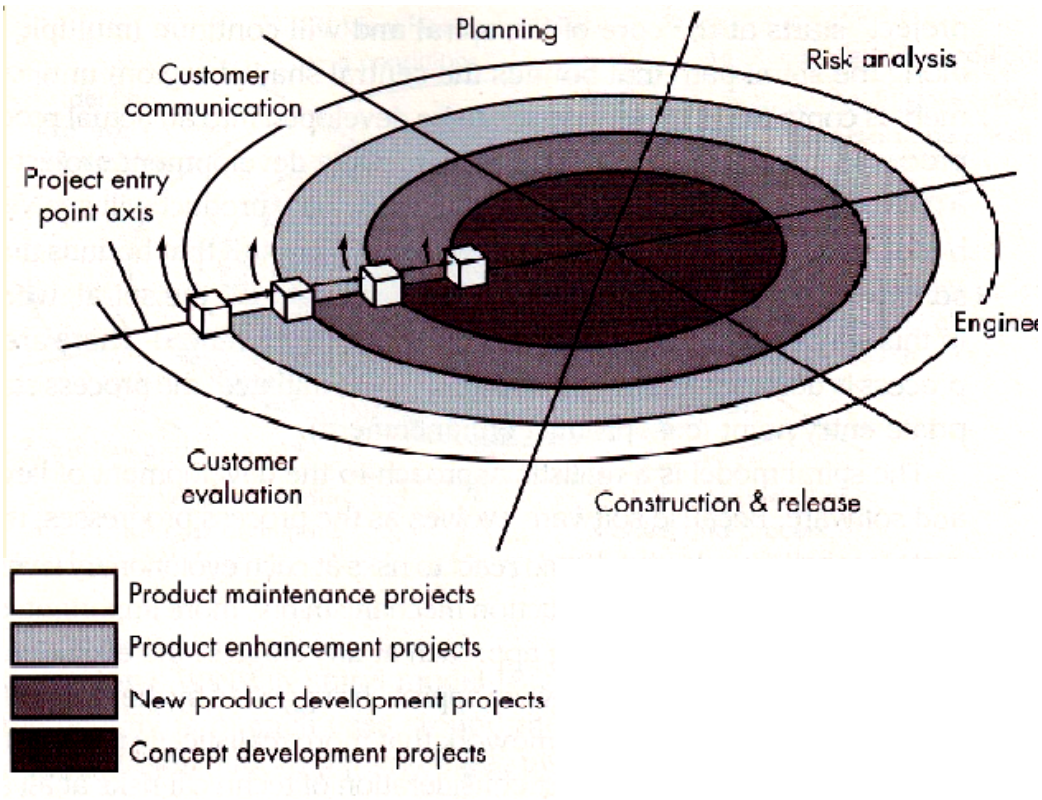


Figure (2.5): The Spiral model

A spiral model is divided into a number of framework activities, also called task regions. Figure 2.5 depicts a spiral model that contains six task regions:

1. **Customer communication-tasks** required to establish effective communication between developer and customer.
2. **Planning-tasks** required to define resources, timelines, and other project-related information.
3. **Risk analysis-tasks** required to assess both technical and management risks.
4. **Engineering-tasks** required to build one or more representations of the application.
5. **Construction and release-tasks** required to construct, test, install, and provide user support (e.g., documentation and training).
6. **Customer evaluation-tasks** required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike classical process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. An alternative view of the spiral model can be considered by examining the project entry point axis, also shown in Figure 2.8. Each cube placed along the axis can be used to represent the starting point for different types of projects.

But like other paradigms, the spiral model is not a panacea, for many reasons:

1. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
2. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.
3. The model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

2.2.4 Component – Based Development

Object-oriented technologies provide the technical framework for a component-based process model for software engineering. The object-oriented paradigm emphasizes the creation of classes that (encapsulate both data and the algorithms used to manipulate the data). If properly designed and implemented, object-oriented classes are reusable across different applications and computer-based system architectures.

The component-based development (CBD) model (Figure 2.6) incorporates many of the characteristics of the spiral model, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from

prepackaged software components (called classes).

Classes created in past software engineering projects are stored in a class library or repository. Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods. The first iteration of the application to be built is then composed, using classes extracted from the library and any new classes built to meet the unique needs of the application. Process flow then returns to the spiral and will ultimately re-enter the component assembly iteration during subsequent passes through the engineering activity.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability, QSM Associates, Inc., reports component assembly leads to a 70 percent reduction in development cycle time; an 84 percent reduction in project cost.

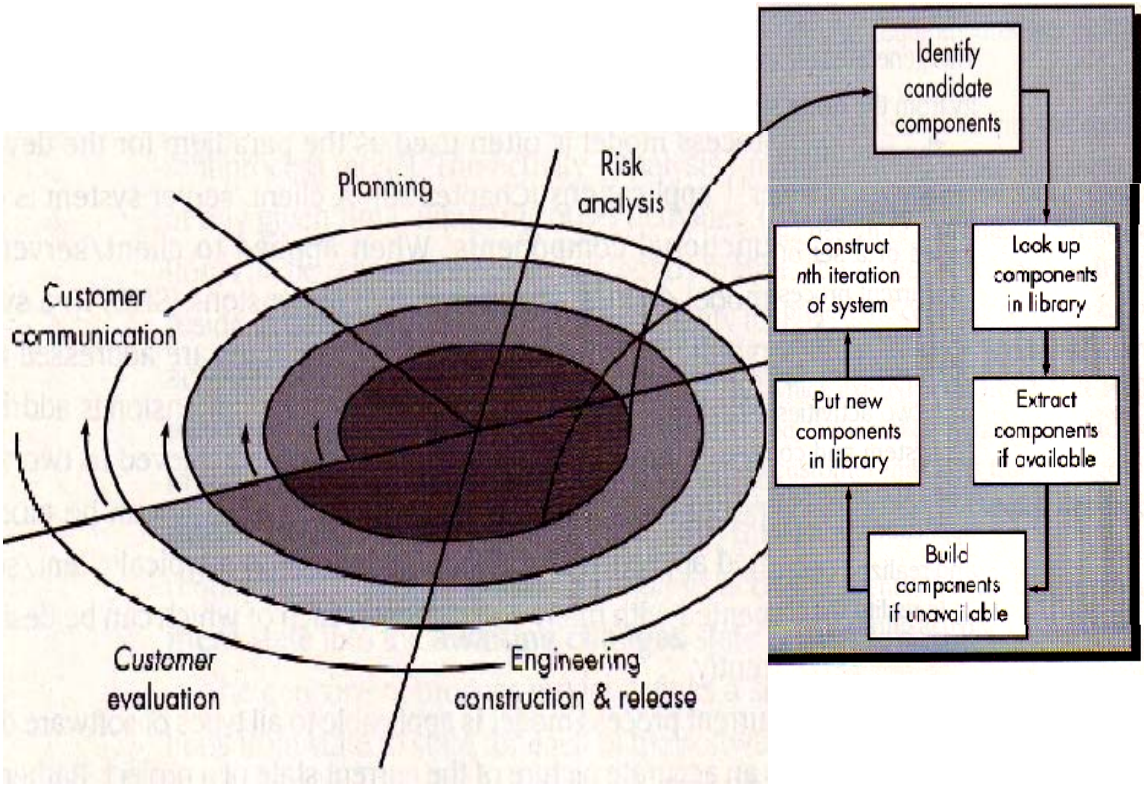


Figure (2.6): The Component – Based Development

University of Technology
Computer Science Department
Software Engineering 3rd Class
Lecturer Yossra Hussain

Chapter Three

Software Process And Project Metrics

Topics:

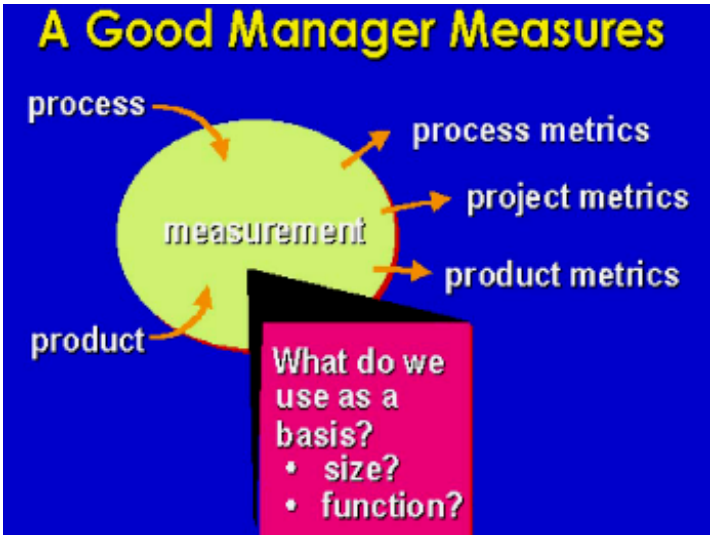
1. Introduction
2. Measures, Metrics, and Indicators
3. Process and Project Indicators
4. Process Metrics
5. Project Metrics
6. Software Measurement
   a. Size-Oriented Metrics
   b. Function-Oriented Metrics
7. Software Quality Metrics
8. Defect Removal Efficiency
9. Integrating Metrics with Software Process
10. Statistical Process Control
11. Metrics for Small Organizations
12. Establishing a Software Metrics Program

3.1 Introduction

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. The four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

3.2 Measures, Metrics, and Indicators

- Measure - provides a quantitative indication of the size of some product or process attribute
- Measurement - is the act of obtaining a measure
- Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute

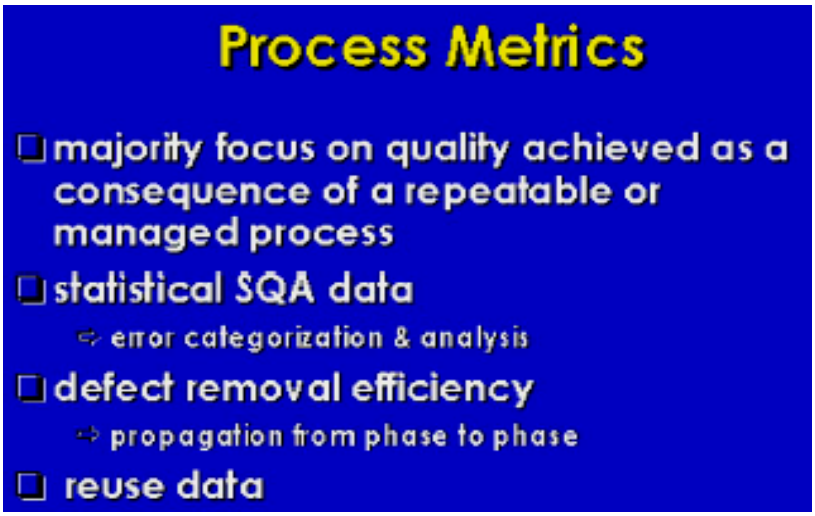


3.3 Process and Project Indicators

- Metrics should be collected so that process and product indicators can be ascertained
- Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality

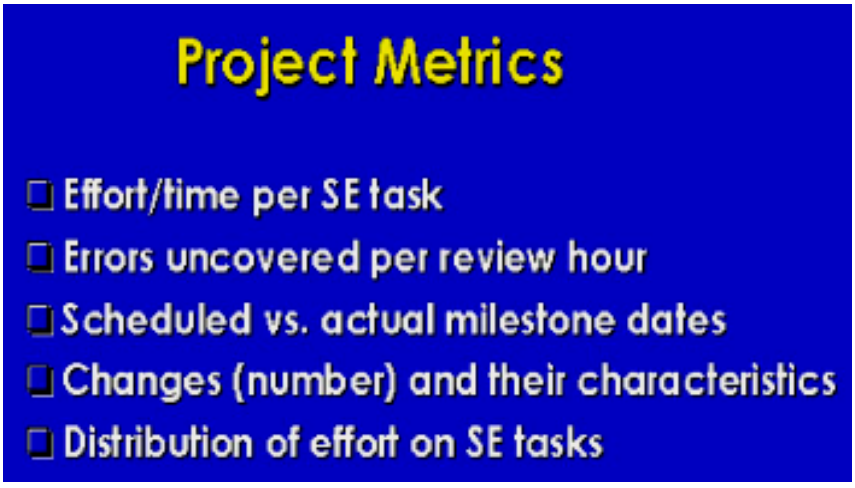
3.4 Process Metrics

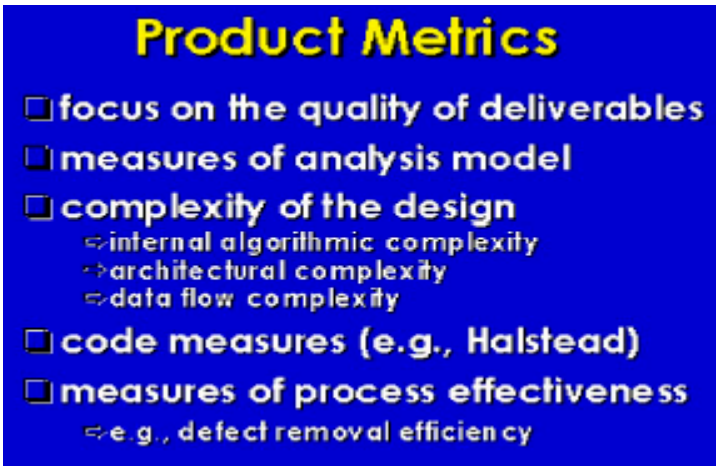
- Private process metrics (e.g. defect rates by individual or module) are known only to the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps an organization to discover where they are strong and where are weak.

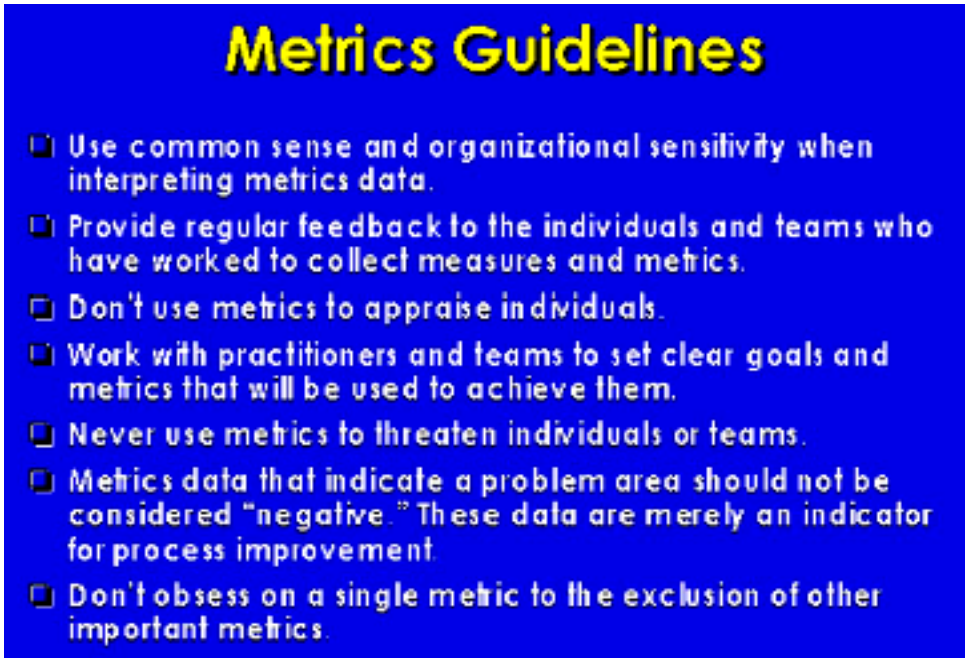


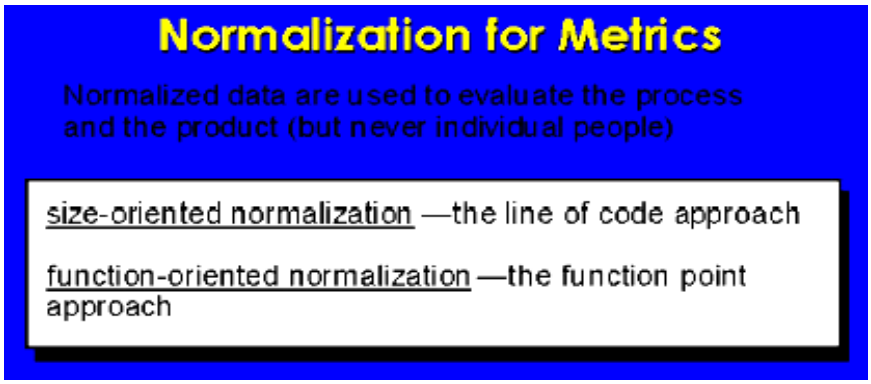
3.5 Project Metrics

- Software project metrics are used by the software team to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).







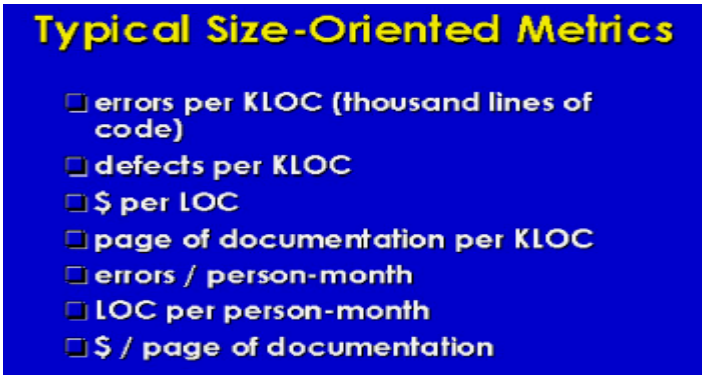


3.6 Software Measurement

- Direct measures of software engineering process include cost and effort.
- Direct measures of the product include lines of code (LOC), execution speed, memory size, defects per reporting time period.
- Indirect measures examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

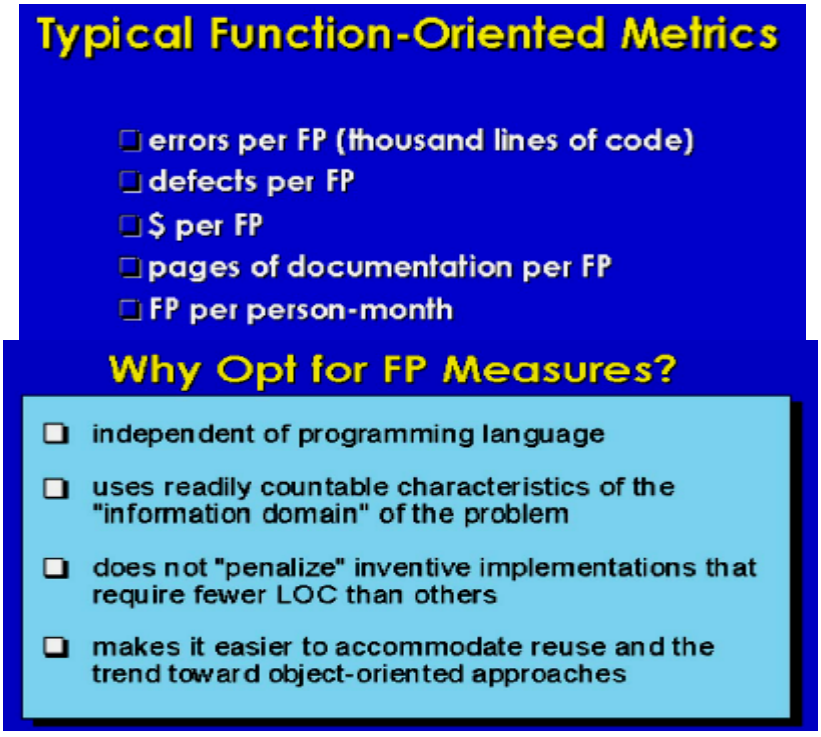
3.6.a Size-Oriented Metrics

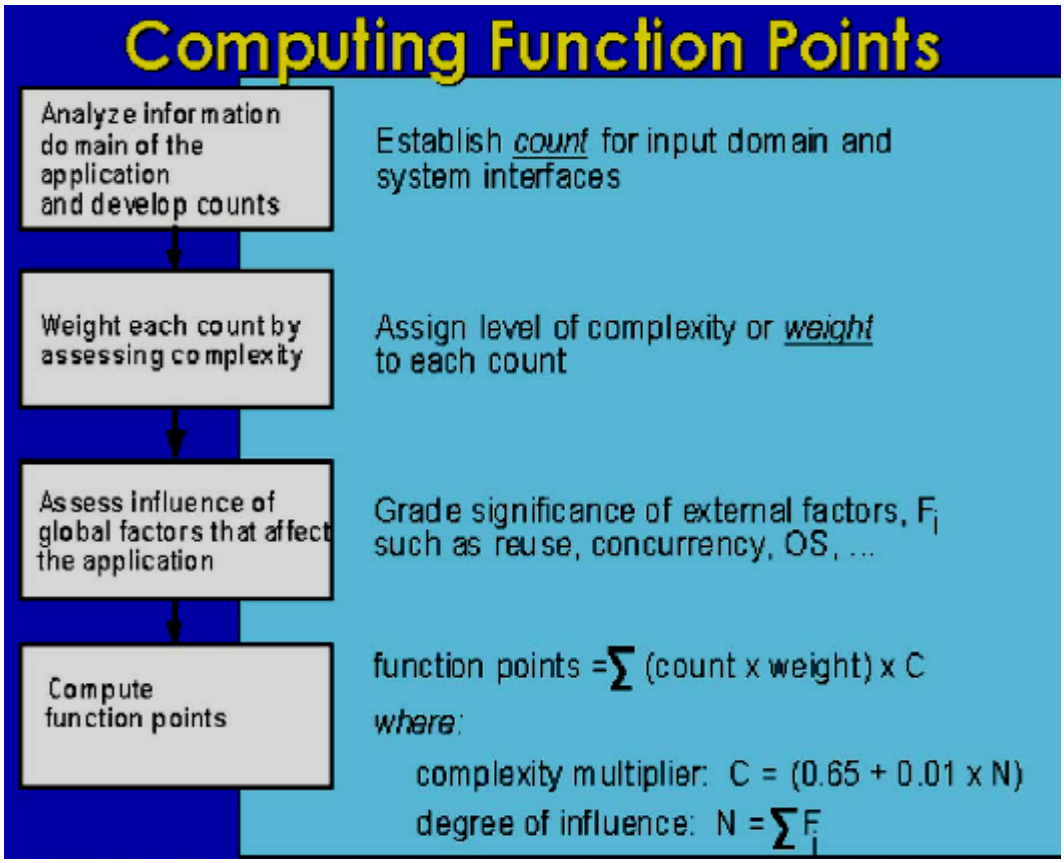
- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

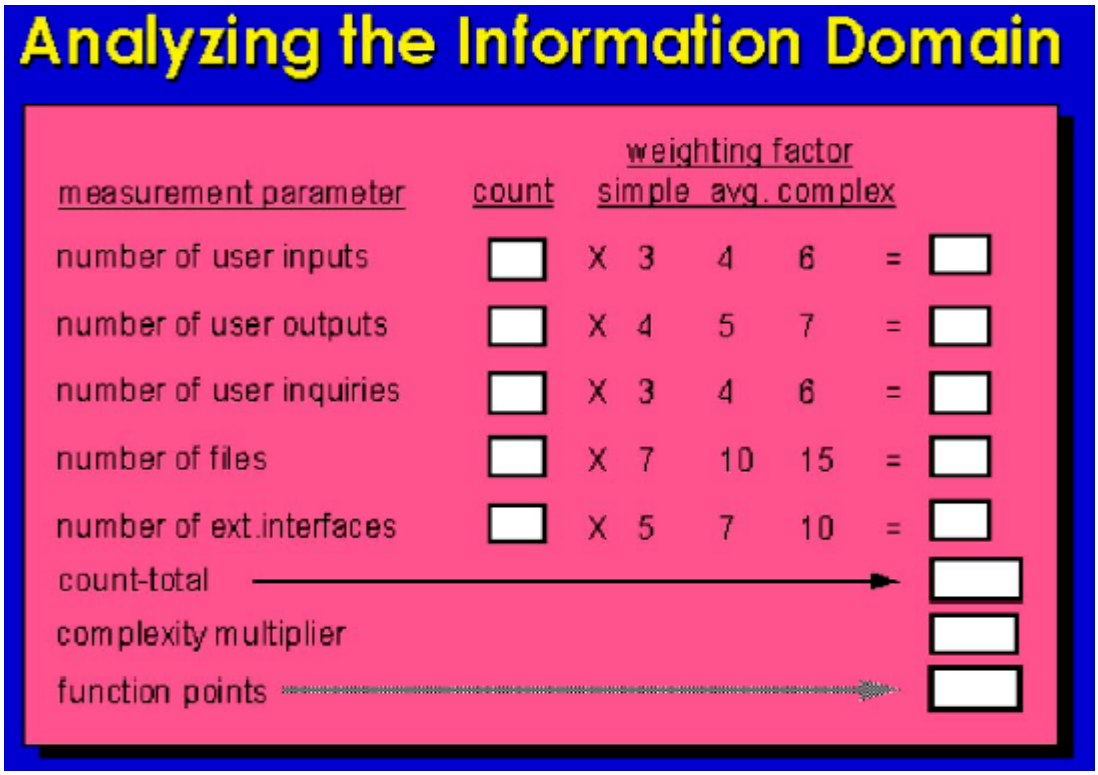


3.6.b Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
- Feature points and 3D function points provide a means of extending the function point concept to allow its use with real-time and other engineering applications.
- The relationship of LOC and function points depends on the language used to implement the software.

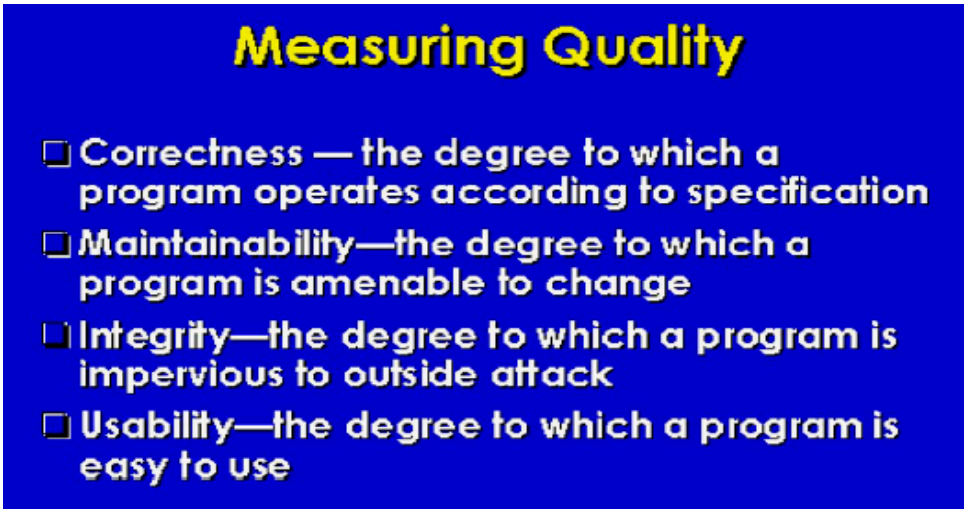



Computing Function Points
Analyze information domain of the application and develop counts
Establish count for input domain and system interfaces
Weight each count by assessing complexity
Assign level of complexity or weight to each count
Assess influence of global factors that affect the application
Grade significance of external factors, F_i such as reuse, concurrency, OS, ...
Compute function points
function points = $\sum$ (count x weight) x C
where:
complexity multiplier: C = (0.65 + 0.01 x N)
degree of influence: N = $\sum F_i$


Analyzing the Information Domain
weighting factor
measurement parameter
count
simple avg. complex
number of user inputs X 3 4 6 =
number of user outputs X 4 5 7 =
number of user inquiries X 3 4 6 =
number of files X 7 10 15 =
number of ext.interfaces X 5 7 10 =
count-total
complexity multiplier
function points

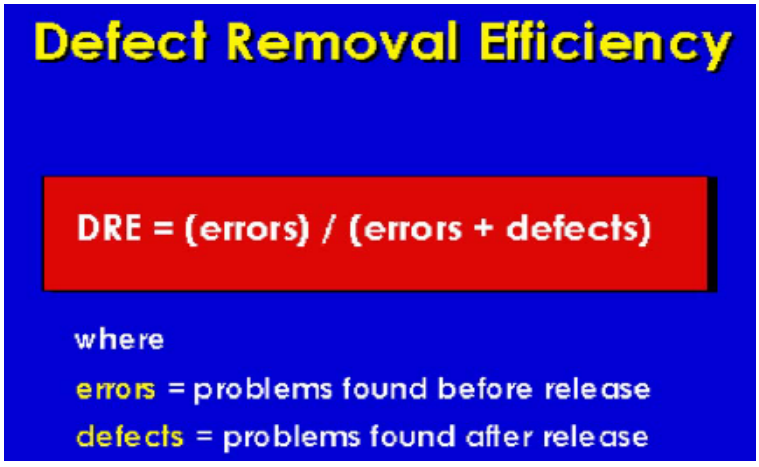
3.7 Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include correctness (defects per KLOC), maintainability (mean time to change), integrity (threat and security), and usability (easy to learn, easy to use, productivity increase, user attitude).
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied throughout the process framework.



3.8 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.



3.9 Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

3.10 Statistical Process Control

- It is important to determine whether the metrics collected are statistically valid and not the result of noise in the data.
- Control charts provide a means for determining whether changes in the metrics data are meaningful or not.
- Zone rules identify conditions that indicate out of control processes (expressed as distance from mean in standard deviation units).

3.11 Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

3.12 Establishing a Software Metrics Program

1. Identify business goal
2. Identify what you want to know
3. Identify subgoals
4. Identify subgoal entities and attributes
5. Formalize measurement goals
6. Identify quantifiable questions and indicators related to subgoals
7. Identify data elements needed to be collected to construct the indicators
8. Define measures to be used and create operational definitions for them
9. Identify actions needed to implement the measures
10. Prepare a plan to implement the measures

University of Technology
Computer Science Department
Software Engineering 3rd Class
Lecturer Yossra Hussain

Software Project Planning

Topics:

4.1 Introduction

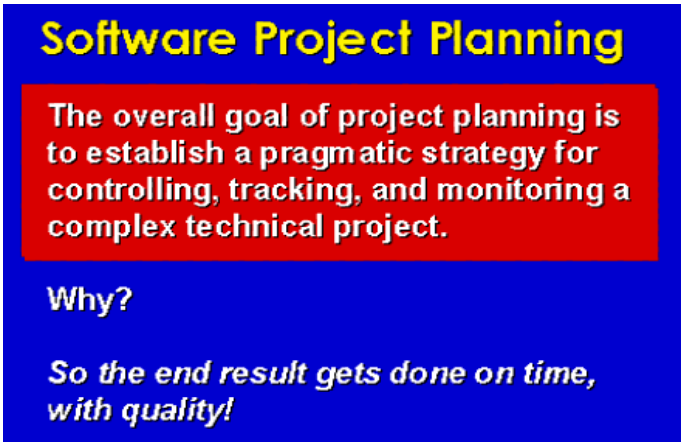
Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system. After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each. The final estimates are typically adjusted by taking project complexity and risk into account. The resulting work product is called a project management plan.

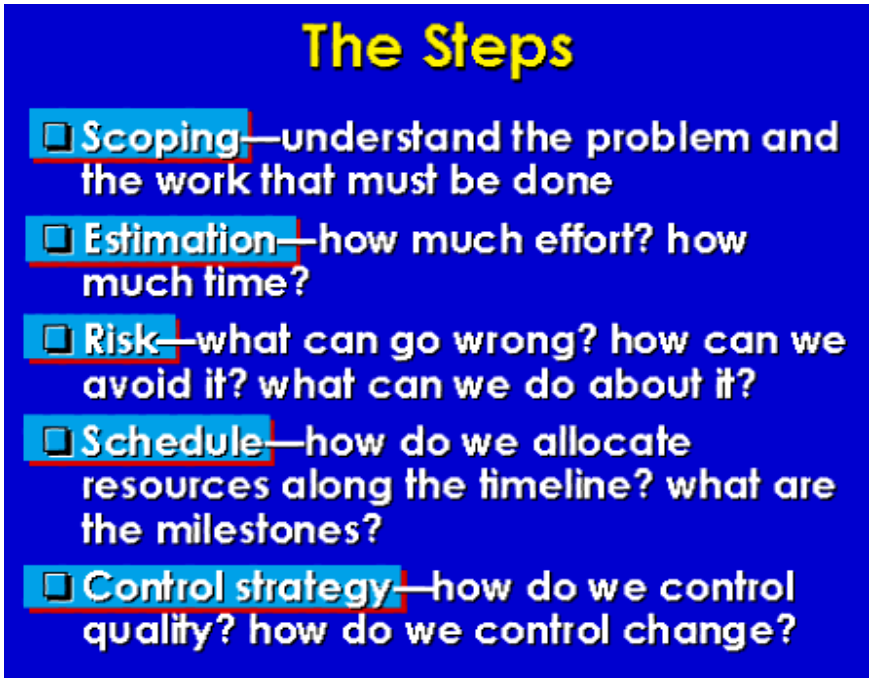
4.2 Estimation Reliability Factors

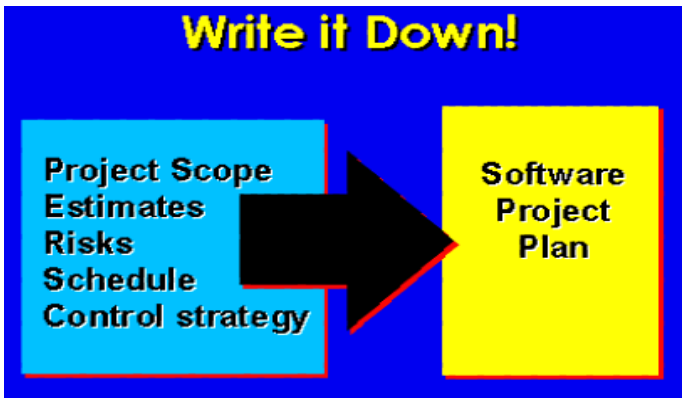
- Project complexity
- Project size
- Degree of structural uncertainty (degree to which requirements have solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information processed)
- Availability of historical information

4.3 Project Planning Objectives

- To provide a framework that enables software manager to make a reasonable estimate of resources, cost, and schedule.
- Project outcomes should be bounded by 'best case' and 'worst case' scenarios.
- Estimates should be updated as the project progresses.

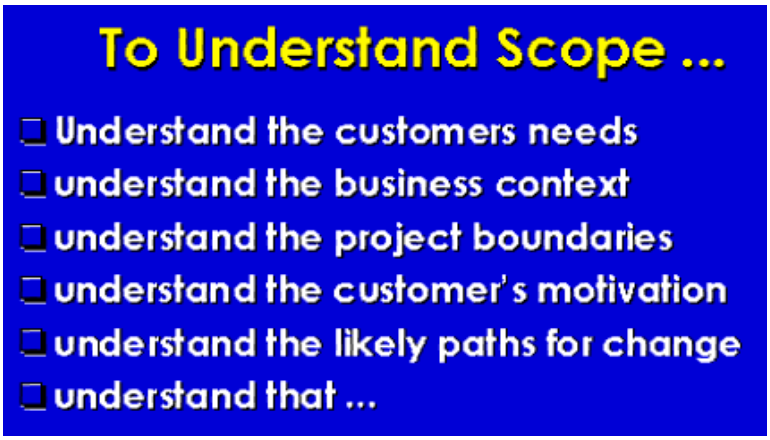






4.4 Software Scope

- Describes the data to be processed and produced, control parameters, function, performance, constraints, external interfaces, and reliability.
- Often functions described in the software scope statement are refined to allow for better estimates of cost and schedule.

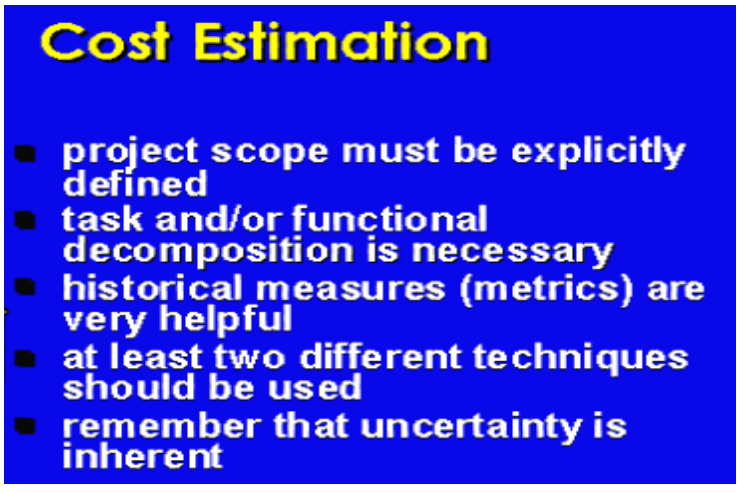


4.5 Estimation of Resources

1. Human Resources (number of people required and skills needed to complete the development project)
2. Reusable Software Resources (off-the-shelf components, full-experience components, partial-experience components, new components)
3. Development Environment (hardware and software required to be accessible by software team during the development process)

4.6 Software Project Estimation Options

1. Delay estimation until late in the project.
2. Base estimates on similar projects already completed.
3. Use simple decomposition techniques to estimate project cost and effort.
4. Use empirical models for software cost and effort estimation.
5. Automated tools may assist with project decomposition and estimation.



4.7 Decomposition Techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems. There are several types of decomposition techniques, these are:

1. Software sizing (fuzzy logic, function point, standard component, change)
2. Problem-based estimation (using LOC decomposition focuses on software functions, using FP decomposition focuses on information domain characteristics)
3. Process-based estimation (decomposition based on tasks required to complete the software process framework)

4.8 Estimation Models

1. The Structure of Estimation Models, typically derived from regression analysis of historical software project data with estimated person-months as the dependent variable and KLOC or FP as independent variables.
2. Constructive Cost Model (COCOMO) is an example of a static estimation model.
3. The Software Equation is an example of a dynamic estimation model.

4.8.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [MAT94]

$$E = A + B \times (ev)^C \qquad (5\text{-}2)$$

where *A, B,* and *C* are empirically derived constants, *E* is effort in person-months, and *ev* is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (5-2), the majority of estimation models have some form of project adjustment component that enables *E* to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).

4.8.2 The COCOMO Model

In his classic book on "software engineering economics," Barry Boehm [BOE81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel.* The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved

into a more comprehensive estimation model, called *COCOMO II* [BOE96, BOE00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

1. **Application composition model.** Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
2. **Early design stage model.** Used once requirements have been stabilized and basic software architecture has been established.
3. **Post-architecture-stage model.** Used during the construction of the software.

Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points and is illustrated in the following paragraphs. It should be noted that other, more sophisticated estimation models (using FP and KLOC) are also available as part of COCOMO II.

Table (4.1)

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Like function points (Chapter 3), the *object point* is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

Once complexity is determined, the number of screens, reports, and components are weighted according to Table 4.1. The object point count is then determined by multiplying the original number of object instances by the weighting factor in Table 4.1 and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

NOP = (object points) x [(100 - %reuse)/100]

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Table 4.2 presents the productivity rate

PROD = NOP/person-month

Table (4.2)

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

for different levels of developer experience and development environment maturity. Once the productivity rate has been determined, an estimate of project effort can be derived as:

estimated effort = NOP/PROD

In more advanced COCOMO II models, a variety of scale factors, cost drivers, and adjustment procedures are required. A complete discussion of these is beyond the scope of this book. The interested reader should see [BOE00] or visit the COCOMO II Web site.

4.8.3 The Software Equation

The software equation [PUT92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, an estimation model of the form

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4) \qquad (5\text{-}3)$$

where E = effort in person-months or person-years

t = project duration in months or years

B = "special skills factor "

P = "productivity parameter" that reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skills and experience of the software team
- The complexity of the application

Typical values might be P = 2,000 for development of real-time embedded software; P = 10,000 for telecommunication and systems software; P = 28,000 for business systems applications. The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

It is important to note that the software equation has two independent parameters:

(1) an estimate of size (in LOC).

(2) an indication of project duration in calendar months or years.

4.9 Automated Estimation Tools

The decomposition techniques and empirical estimation models described in the preceding sections are available as part of a wide variety of software tools. These automated estimation tools allow the planner to estimate cost and effort and to perform "what-if" analyses for important project variables such as delivery date or staffing. Although many automated estimation tools exist, all exhibit the same general characteristics and all perform the following six generic functions [JON96]:

1. **Sizing of project deliverables.** The "size" of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), descriptive information (e.g. documents).
2. **Selecting project activities.** The appropriate process framework (Chapter 2) is selected and the software engineering task set is specified.
3. **Predicting staffing levels.** The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.
4. **Predicting software effort.** Estimation tools use one or more models (e.g., Section 5.7) that relate the size of the project deliverables to the effort required to produce them.
5. **Predicting software cost.** Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.
6. **Predicting software schedules.** When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities.

When different estimation tools are applied to the same project data, a relatively large variation in estimated results is encountered. More important, predicted values sometimes are significantly different than actual values.

University of Technology
Computer Science Department
Software Engineering 3rd Class
Lecturer Yossra Hussain

Analysis Concepts and Principles

Topics:

5.1 Introduction

After system engineering is completed, software engineers need to look at the role of software in the proposed system. Software requirements analysis is necessary to avoid creating software product that fails to meet the customer's needs. Data, functional, and behavioral requirements are elicited from the customer and refined to create specification that can be used to design the system. Software requirements work products must be reviewed for clarity, completeness, and consistency.

5.2 Requirements Analysis

- Software engineering task that bridges the gap between system level requirements engineering and software design.
- Provides software designer with a representation of system information, function, and behavior that can be translated to data, architectural, comonents-level design.
- Expect to do a little bit of design during analysis and a little bit of analysis during design.

Software Requirements Analysis

- Identify the "customer" and work Together to negotiate "product –level"
- Build an analysis model

  -Focus on data
  - define function
  - represent behavior

- Prototype areas of uncertainty
- Develop a specification that will guide design
- Conduct formal technical reviews.

5.3 software Requirements Analysis Phases

- Problem recognition
- Evaluation and synthesis (focus is on what not how)
- Modeling
- Specification
- Review

5.4 software requirements elicitation

- Customer meetings are the most commonly used technique.
- Use context free question to find out customers goal and benefits, identify stakeholders, gain understanding of problem, determine, customer reactions to proposed solution, and assess meeting effectiveness.
- If many users are involved, be certain that a representative cross section of users is interviewed.

5.4.1 Facilitated action Specification Techniques (FAST)

- Meeting held at neutral site, attended by both software engineering and customers.
- Rules established for preparation and participation.
- Agenda suggested to cover important points and to allow for brainstorming.
- Meeting controlled by facilitator (customer, developer, or outsider).
- Definition mechanism (flip charts, stickers, electronic device, etc.) is used.
- Goal is to identify problem, propose elements of solution, negotiate different approaches, and specify a preliminary set of solution requirements.

Fast Guidelines

- Participants must attend entire meeting
- All participants are equal
- Preparation is as important as meeting
- All pre-meeting documents are to be viewed as " proposed"
- Off-sit meeting location is preferred
- Set an agenda and maintain it
- Don’t get mired in technical

5.4.2 Quality Function Deployment (QFD)

- Translates customer needs into technical software requirements.
- Uses customer interviews, observation, surveys, and historical data for requirements gathering.
- Customer voice table(contains summary of requirements)
- Normal requirements (must be present in product for customer to be satisfied)
- Expected requirement (things like ease of use or reliability of operation, that customer assumes will be present in a professionally developed product without having to request them explicitly)
- Exciting requirements (features that go beyond the customers expectations and prove to very satisfying when they are present)
- Function deployment (used during customer meeting to determine the value of each function required for system)
- Information deployment (identifies data objects and events produced and consumed by the system)
- Task deployment(examines the behavior of product within in environment)
- Value analysis (used to determine the relative priority of requirements during function , information, and task deployment)

Quality Function Deployment

- Function deployment determines the value (as perceived by the customer)of each function required of the system
- Information deployment identifies data objects and events
- Task deployment examines the behavior of the system
- Value analysis determines the relative priority of requirements

5.4.3 Use - case

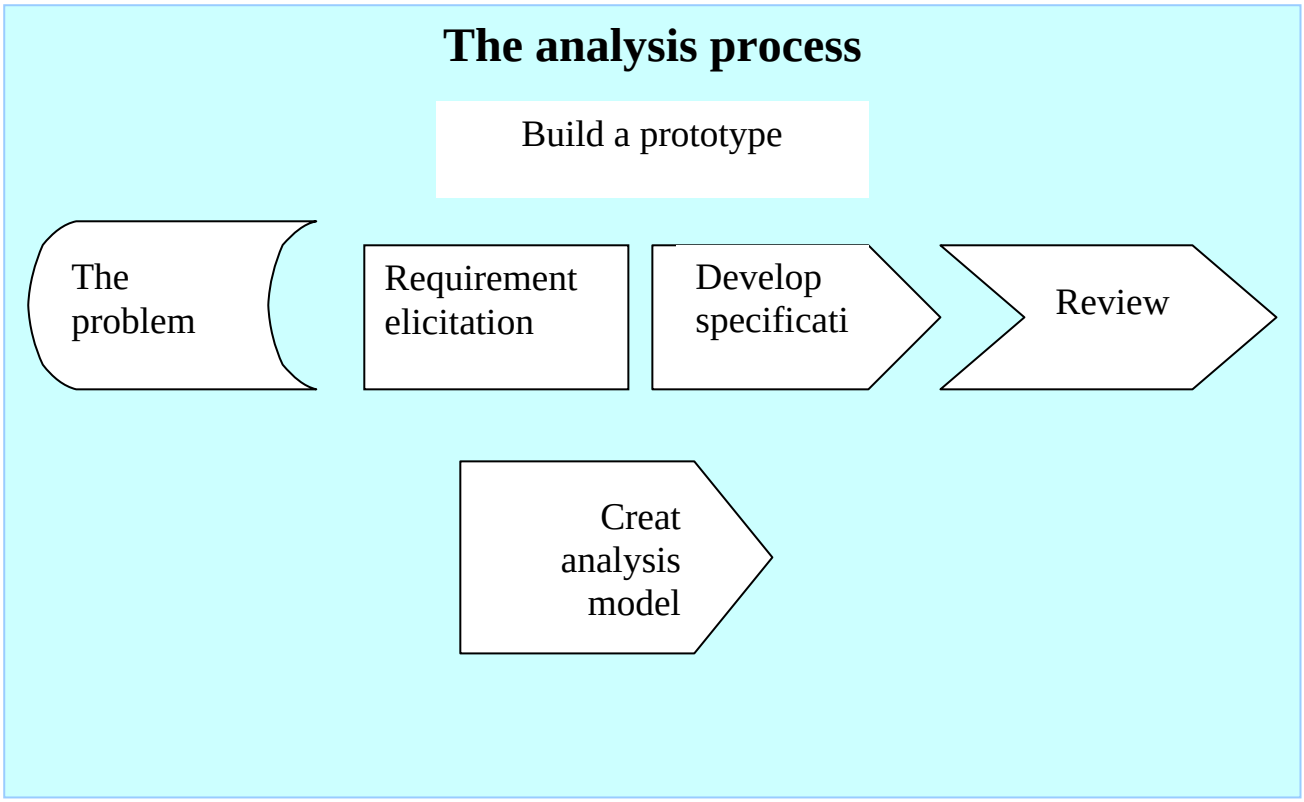
- Scenarios that describe how the product will be used in specific situations.
- Written narratives that describe the role of an actor (user of device) as interaction with the system occurs.
- Use-cases are designed from the actor's point of view.
- Not all actors can be identify the primary actors before developing the use-cases

Use-cases

- A collection of scenarios that describe the thread of usage of a system
- Each of an "actor" – a person or device that interacts with the software in some way
- Each scenario answers the following questions :
  - What are the main tasks of functions performed by the actor?
  - What system information will the actor acquire, produce or change?
  - Will the actor inform the system about environmental changes?
  - What information does the actor require of the system?
  - Does the actor wish to be informed about unexpected changes?

5.5 Analysis Principles

- The information domain of the problem must be represented and understood.
- The functions that the software is to perform must be defined.
- Software behavior must be represented.
- Models depicting information, function, and behavior most be partitioned in a hierarchal manner detail.
- The analysis process should move from the essential information toward implementation details.

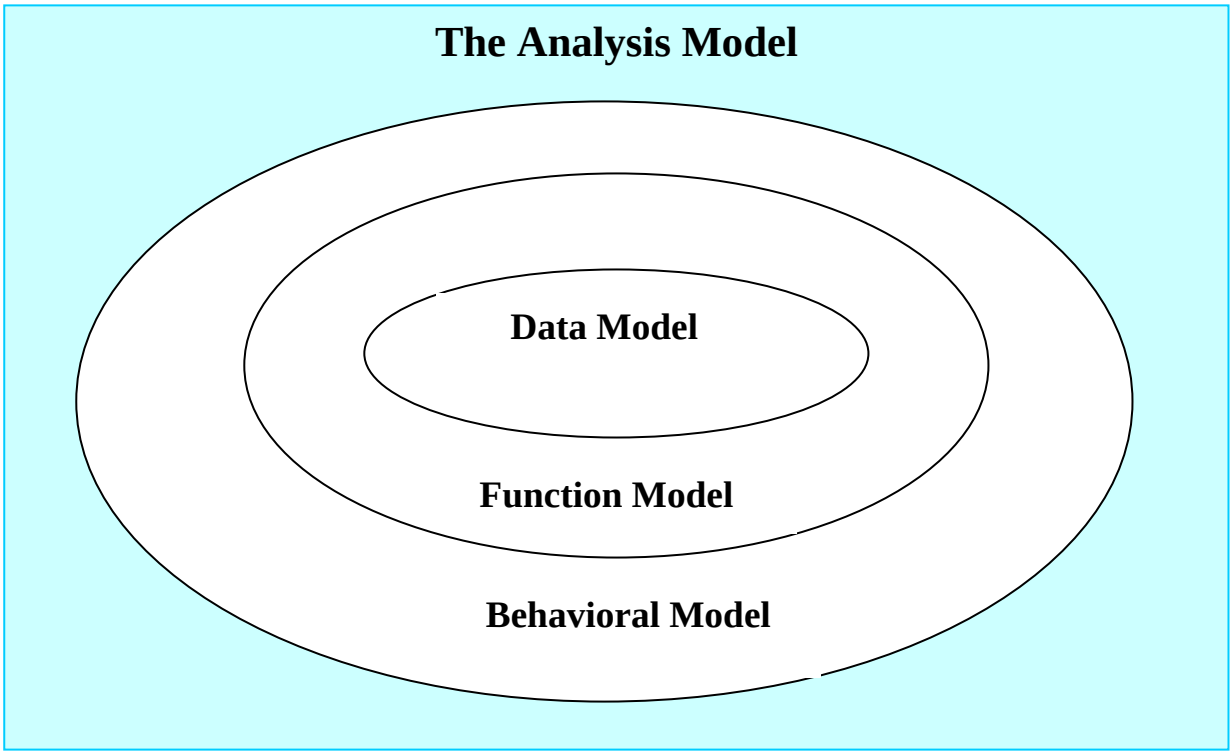


5.5.1 Information Domain

- Encompasses all data objects that contain numbers, text, images, audio, or video
- Information content or data model (shows the relationships among the data and control objects that make up the system)
- Information flow(represents the manner in which data control objects change as each moves through the system)
- Information structure (representations of the internal organizations of various data and control items)

5.5.2 Modeling

- data model (shows relationships among system objects)
- function model (description of the functions that enable the transformation of system objects)
- behavioral model (manner in which software responds to events from the outside world)



Analysis Principle I Model the Data Domain

- Define data object
- Describe data attributes
- Establish data relationships

Analysis Principle II Model Function

- Identify functions that transform data objects
- Indicate how data flow through the system
- Represent producers and consumers of data

Analysis Principle III Model Behavior

- Identify deferent states of the system
- Specify events that cause the system to change state

5.5.3 Partitioning

- Process that result in the elaboration of data, function, or behavior.
- Horizontal partitioning is a breadth –first decomposition of the system function, behavior, or information, one level at a time.
- Vertical portioning is a depth – first elaboration of the system function, behavior, or information, one subsystem at a time.

Analysis principles | | | | Partition the Models

- Refine each model to represent lower level of abstraction
  - refine data objects
  - create a functional hierarchy
  - represent behavior of different levels of detail

5.5.4 Software Requirements Views

- Essential view- presents the functions to be accomplished and the information to be processed without regard to implementation.
- Implementation view- presents the real world manifestation of processing functions and information structures.
- Avoid the temptation to move directly to the implementation view, assuming that the essence of the problem is obvious.

5.6 Software Prototyping

- Throwaway prototyping (prototype only used as a demonstration of product requirements, finished software is engineered using another paradigm)
- Evolutionary prototyping (prototype is refined to build the finished system)
- Customer resources must be committed to evaluation and refinement of the prototype.
- Customer must be capable of making requirements decisions in a timely manner.

5.6.1 Prototyping Methods and Tools

- Fourth generation techniques (4 GT tools allow software engineer to generate executable code quickly)
- Reusable software components (assembling prototype from a set of existing software components)
- Formal specification and prototyping environments (can interactively create executable programs from software specification models)
-

5.7 Specification Principles

- Separate functionality from implementation.
- Develop a behavioral model that describes functional responses to all system stimuli.
- Define the environment in which the system operates and indicate how the collection of agents will interact with it.
- Create a cognitive model rather than an implementation model.
- Recognize that the specification must be extensible and tolerant of incompleteness.
- Establish the content and structure of a specification so that it can be changed easily.

University of Technology

Computer Science Department

Software Engineering 3rd Class

Lecturer Yossra Hussain

Software Testing

Topics:

6.1 Introduction

6.2 Why Testing

6.3 What is Testing

6.4 Work Products of Testing stage:

6.5 Testing is Important

6.6 Software Testing Technique

6.7 What Testing Shows

6.8 Testing Stage of the Software Process

6.9 Testing Principles

6.10 Who tests the system6.11 Attributes of a Good Test Case

6.12 Engineering Methods on Product Testing

6.13 White–Box Testing

6.14 Black–box Testing

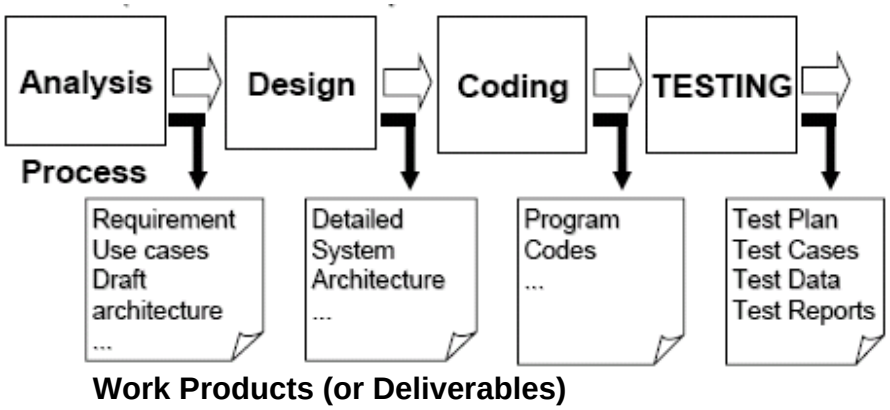
Software Engineering

- Software Engineering is about applying "sound engineering principles" in developing software.
- Science vs. Art: Science (and engineering) are based on theories, principles, practices, processes, and is precise. Pure art likes painting and music is relied on individual's artistic talent and is not precise.
- Is writing software a science (which is precise, predictable and Newton's laws of motion applied) or an art (like composing music, you need a Beethoven to compose a great piece of software)?
- The days of solo programmer (or a few programmers) are gone. Software today is created by a big team of people.
- Can we apply engineering principles, methods, techniques, processes, to developing software?

6.1 Introduction

Revisit the Software Life Cycle

⬟ Classical Life Cycle (or Linear Sequential Software Process Model, or Waterfall model)



⬟ Analogy: Construct a building, chemical & manufacturing process,etc.

6.2 Why Testing?

Two objectives – Verification and Validation (V&V):

⬟To uncover errors (or bugs) in the software before delivery to the client. This is called **Verification** -Verify that the program is working. "Are you building the product right? Right?"

⬟ To ascertain that the software meet its requirement specification. This is called **Validation –** Validate that the software meets its requirements. "Are you building the right product? "

6.3 What is Testing?

- At a lower level, testing involves designing a series of "TEST CASES" (or "TEST SUITE") to uncover errors and validate conformance to requirements.
- At a higher level, testing involves formulating a test plan and test strategy for the execution of the testing process.

6.4 Work Products (or Deliverables) of Testing stage:

- Test Plan (Test Strategies).
- Test Report (includes Test Cases, Test Data, etc.).

6.5 Testing is Important

"Testing is as important, if not more important than coding. "
Your clients will not accept programs that are full of bugs, or worse still, don t meet the requirements!.

"The older I get, the more experience on creating software, the more aggressive I get about testing."

- Although testing is one of the steps in the software process, testing would be the most time consuming and costly step among all.
- Not unusual to cost 30 40% of the total project cost. In the extreme (e.g., flight control system, nuclear reactor), testing can cost 3 5 times as much as all the other steps combined. E.g., NEL was delay for many months to carry out "System Test".

A Trivial Example on Testing

Requirement:

Write a program to assign an alphabetic grade to raw marks as follows:

Marks	Grade
0 - 49	'F'
50 - 59	'E'
60 - 69	'D'
70 - 79	'C'
80 - 89	'B'
90 - 100	'A'

Program without Testing

```
if (marks >= 90 && marks < 100) return 'A';
else if (marks >= 80 && marks < 90) return 'B';
else if (marks >= 70 && marks < 80) return 'C';
else if (marks >= 60 && marks < 70) return 'D';
else if (marks >= 50 && marks < 60) return 'E';
else return 'F';
```

- This code can compile and run! (Compiler only catches syntax errors, NOT semantics errors or logical errors.)
- Is the program working? (May be!) (Verification)
- Is the program correct? Does the program meet its specification? (NO!) (Validation)
- Is the program efficient? (This is an issue on Software Quality Assurance (SQA), not testing–There are 10 comparisons in the code, many of them are redundant!)

Test Cases

- To verify and validate the program, we design "a series of test cases". Each test case contains a specific input and the expected output. For examples,

Test Case No	Input	Expected Output	Purpose of Test
1	100	'A'	Grade 'A' upper limit
2	49	'F'	Grade 'F' upper limit
3	1	'F'	Grade 'F'
4	2	'F'	Grade 'F'
5	...	...	...

- How many test cases is "necessary and sufficient"? (101? How about numbers like 200, or–155? Countable Infinity!)
- This series of lectures on "testing techniques" teaches you how to design good test cases "applying sound engineering principles".

Integrate Testing into the "Build" Process

- Testing can be integrated into the build (compile) process.
  - The compiler checks for syntax errors in the program code.
  - The tests pick up semantic or logical errors.

- **Demonstration** : based on our trivial example,

Step 1 : Design Test Cases

-Typical values: 95(A), 84(B), 76(C), 67(D), 53(E), 30(F).

-Boundary values: 100(A), 90(A), 89(B), 80(B), 79(C), 70(C),69(D), 60(D), 59(E), 50(E), 49(F), 0(F).

-Out of range: 1(Error), 101(Error)

Step 2 : Write the program code.
-compile the program to ensure it is syntactically correct.
-run the test cases to ensure that it is semantically correct.
Step 3 : Integrate tests into build process, generate "daily build".
-Use an appropriate CASE tool (e.g., make, ant).
-Run the integrated build process whenever you modify the code.

Self Testing Software

"The older I get, the more aggressive I get about testing."
"Testing should be a continuous process. No code should be written until you know how to test it. Once you have written it , write the tests for it. Until the tests work, you cannot claim to have finish writing the code."
"Test code, once written, should be kept forever. Set up your tests so that you can run every test with a simple command. The tests should return with either "OK" or a list of failures"
"I find that writing unit tests actually increases my □rogramming g speed."

Best Practices on Testing

Q: When should test cases be written?
A: "Extreme Programming group" advocates writing the test cases first, before writing the code.
- Good tests tell you how to best design the system for its intended use.
- Good tests prevent you from over build the system. When all the tests pass, you know you've done.

Q: Do I have to write a test for everything?
A: No, just test those things that could reasonably break . Do not write tests, that ultimately turns out to be testing the compiler or the operating system, and not your own program.

Best Practices on Testing (cont.)

Q: How often should I run my tests?

A: As often as possible, ideally whenever the code is changed. Frequent testing gives you the confidence that your changes didn't break anything and generally lowers the stress of programming.

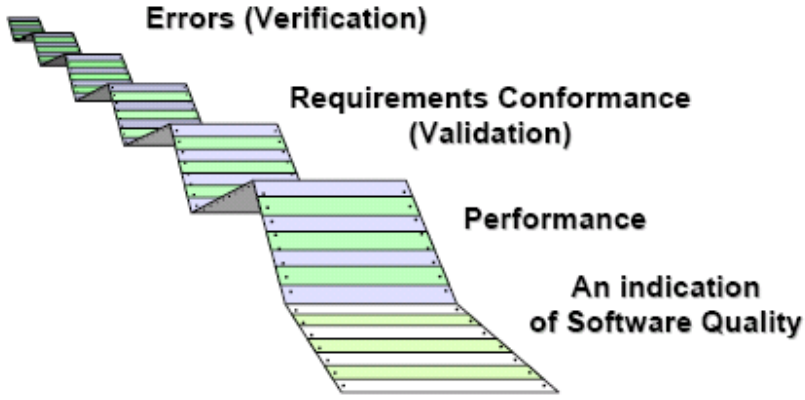
Run all you tests at least once a day. (You backup at least once a day, Don't you!)

6.6 Software Testing Technique

Software Testing

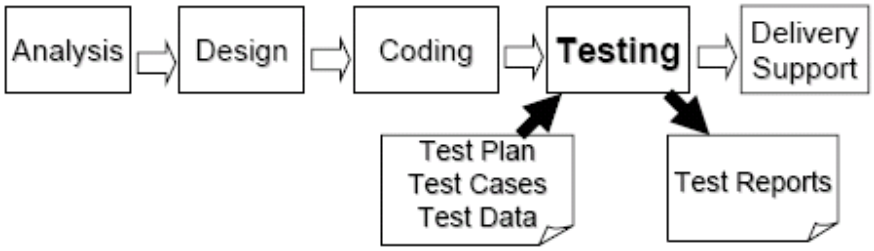
- Once source code has been written, software must be tested (and corrected) to uncover as many errors as possible before delivery to your customer.
- "Software Testing" is the process of exercising a program with the specific intent of finding errors prior to delivery to the e nd user (Verification).
- Testing also checks if the software meets its intended usage. Testing represents the ultimate review of the specification, analysis, design, and coding (Validation).
- Testing is a essential element of Software Quality Assurance (SQA), to achieve high quality software.
- **Testing is important** : Not unusual to cost 30- 40% of the total project cost. In the extreme (e.g., flight control system, nuclear reactor), testing can cost 3- 5 times as much as all the other steps combined.

6.7 What Testing Shows?

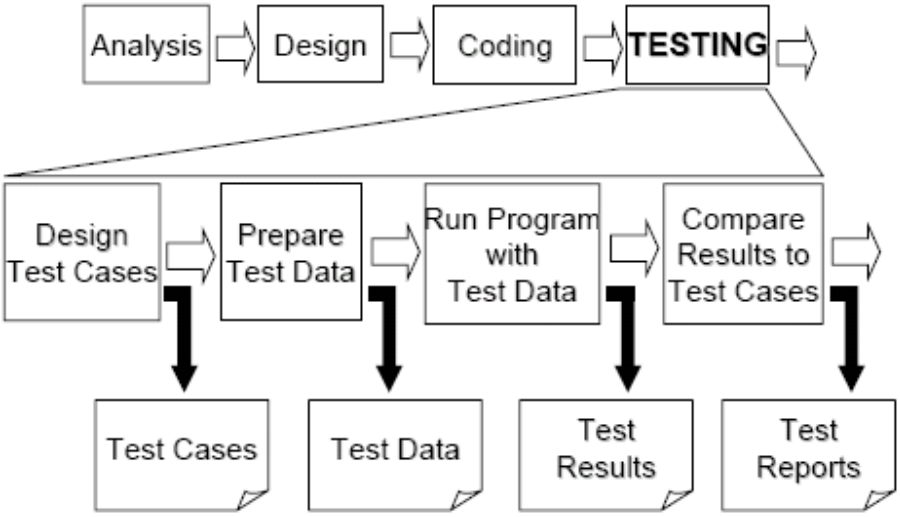


6.8 Testing Stage of the Software Process

- The goal of testing is to "design a series of test cases" that has "a high likelihood of finding errors".
- How? Design test cases systematically by "applying sound engineering principles and methods".
- The work product of the testing stage is a "test report" that documents all the test cases run, i.e., the test input, the expected output, the actual output, the purpose of the test and etc.



Testing Stage Details



6.9 Testing Principles

1. All tests should be traceable to customer requirements – to ensure that the software meets its intended use.
2. Tests should be planned long before testing begins – write tests first, before the coding.
3. Testing should begin "in the small" and progress toward testing "in the large" – perform unit tests, then integration tests, then validation test, and then system tests.
4. The "80–20 rule" applied – 80% of the errors are located in 20% of the software modules, isolate them and test them thoroughly.
5. To be more effective, testing should be conducted by an independent third third–party testing specialist (ITG or Independent Testing Group).
6. Exhaustive test is not possible.

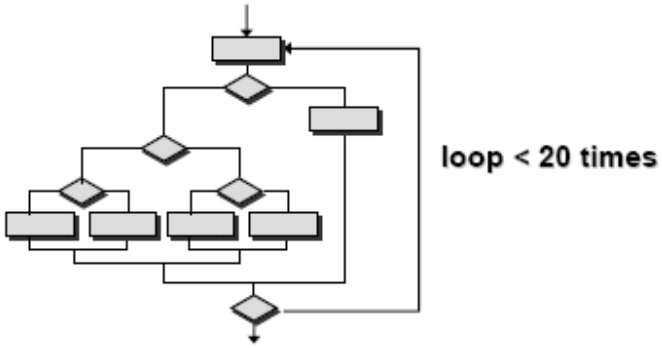
6.10 Who tests the system?

Developer
Understands the system,
but will test "gently",
and is driven by "delivery"

Independent Teste
Must learn about the system,
but will attempt to "break" it,
and is driven by "quality".

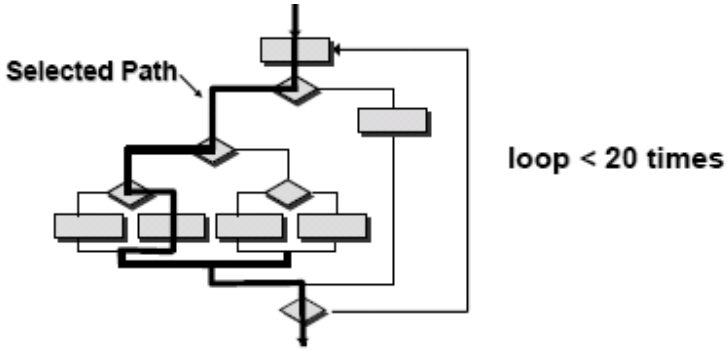
- During the early stages of testing, the developer performs the tests. As the testing progresses, independent test specialist may involved.
- "Open–source" software like Linux, Java, Apache are known to be more secure and less buggy because many independent parties have "tested" the source code.

Exhaustive Test is NOT Possible



- 5 forward paths, 1 to 20 passes, total number of possible paths is (5+52+…+520) about 1014. If we execute one test per millisecond, it would take 3171 years to exhaustively test all the paths.
- Observation: a decision block adds one extra forward path, aloop multiplies the number of forward paths.

Selective Testing is Desired



- Which path to select? How many testing paths is deem "necessary and ufficient"?
- Test case design techniques: Apply engineering principles and methods to design the "necessary and sufficient" test cases, instead of testing for everything.

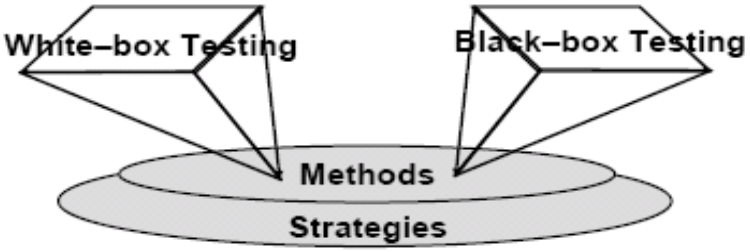
6.11 Attributes of a Good Test Case

- A good test has a high probability of finding an error. To design good tests, you have to understand how the program might fail. (E.g., in the trivial example, it probably fails at the boundary).
- A good test is not redundant. (E.g., If you have a test case of 96 marks, a test case of 97 marks is considered redundant, because they are trying to uncover the same class of error.)
- A good test should be neither too simple nor too complex. Each test case preferably has one and only one purpose.
- Don't
  - treat testing as an afterthought.
  - develop test cases that may "feel right" but have little assurance of being complete nor effective.

6.12 Engineering Methods on Product Testing

An "engineered product" can be tested in one of two ways:

1. knowing the specified functions that the product is designed to perform, conduct tests to validate each of these functions – this is called black– box testing.
2. Knowing the internal workings of the product, conduct tests to ensure that the internal operation are correct and internal components are properly exercised – this is called white–box testing (or glass–box testing).



Software Testing Methods

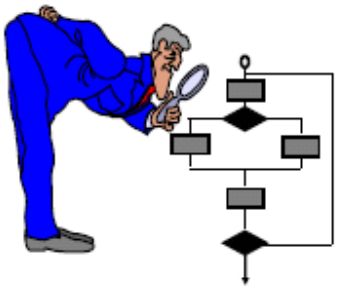
- **Black–Box Testing:**

- Tests conducted at the system interface, input/output domain.
- Tests are used to demonstrate that the software are operational, that input is properly accepted and output is correctly produced.
- Black–box tests examine the functional aspect of the system with little regard for the internal logical structure of the software.
- Only executable code needed for black–box testing. No source code needed.

- **White–box Testing:**
  - Examine the procedural details and internal structure.
  - All paths through the software are exercised at least once. The status of the program may be examined at various points to determine if the expected status (or asserted status) corresponds to the actual status.
  - Source code needed for white–box testing.

6.13 White–Box Testing

- **Goal**: to ensure that all statements and conditions have been executed at least once.



- **Principles:**
  - Exercise all independent paths at least once.
  - Exercise all the logical decisions (if–then–else) on their true and false sides.
  - Exercise all the loops (for, while–do loops) at their boundaries and within their bounds.
  - Exercise all the internal data structure to ensure their validity.

Why White box Testing?

- Black box testing on the functionality is not sufficient to deliver a robust software, examine the source code and internal thru white box testing is important because:
  - Errors are inversely proportional to a path's execution frequenc y. Everyday processing tends to be well understand and well scrutinized, while "special case" (such as error handling code) tends to fall into the cracks. Hence, there is a need to exerci se every paths, not just the frequently used ones.
  - We often believe that a path is not likely to be executed; in fa ct, it may get executed in a regular basis.If you made a wrong assumption and did not cater a path,the test would unveil it.
  - Typographically errors are random, and exist on mainstream path as well as obscure path. Hence, there is a need to oolean e all the paths.

White- box Methods

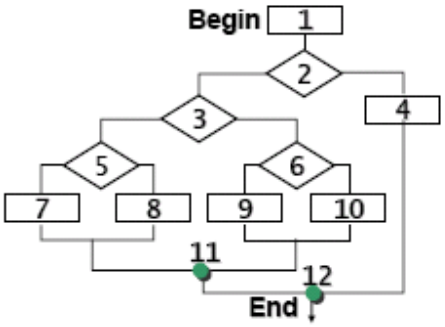
- White–box methods focuses on testing the control structure (e.g., if–then–else decision, loop) of the program.

- The commonly–used white–box methods are:
  1. Basis–Path Testing.
  2. Condition Testing:
     (a) Branch Testing
     (b) Domain Testing
     (c) Branch and Relational Operator (BRO) Testing
  3. Loop Testing
  4. Others

White–box Method 1– Basis Path Method

Objective: ” Find a basis set of independent execution paths “ that covers all the statements in the program “at least once “.

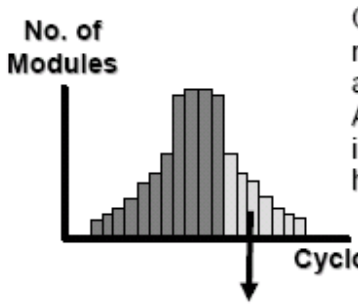
Step 1: Compute the cyclomatic complexity (C) of the program, which indicates the number of independent execution paths.
Step 2: ind a basis set of “C” independent execution paths.
Step 3: Design test cases for the basis set.



Example:
(1) Cyclomatic Complexity = 5
(2) The 5 independent paths are:
   a. 1⇒2⇒3⇒5⇒7⇒11⇒12
   b. 1→2→3→5⇒8⇒11→12
   c. 1→2→3⇒6⇒9⇒11→12
   d. 1→2→3→6⇒10⇒11→12
   e. 1→2⇒4⇒12
(3) Design 5 test cases accordingly.

Cyclomatic Complexity

- Cyclomatic complexity indicates the number of independent paths of the program.
- It gives us an upper bound for the number of tests that must be conducted to ensure that all the paths have been executed at least once.



Cyclomatic complexity (C) is a measure of the logical complexity of a program.
A number of industry studies have indicated that the higher the C, the higher the probability or errors.

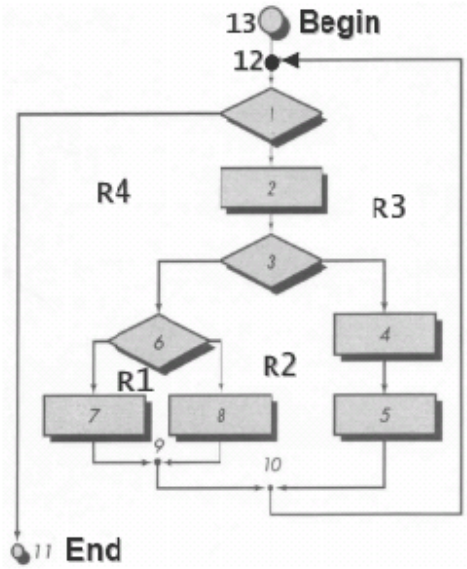
Modules in this range are more error prone.

Calculating Cyclomatic Complexity

Cyclomatic complexity (C) can be calculated using any one of the following three formula (which are based on the Graph Theory):

(1) **C = 1 + No. of Simple Decisions (P)**
(There is only 1 path without any decision. Each decision introduces an additional path.)
(2) **C = No. of Regions (R)**
(3) **C = No. of Edges (E) – No. of Nodes (N) + 2**
(You must draw a proper graph to use this formula. An edge starts from a node and ends at another node. All nodes must be connected.)

Cyclomatic Complexity –Example



Formula 1:
No of Decisions P = 3
$C_1 = 1 + P = 4$

Formula 2:
No of Regions R = 4
$C_2 = R = 4$

Formula 3:
No of Node N = 13
No of Edges E = 15
$C_3 = E - N + 2 = 4$

Cyclomatic complexity
$C = C_1 = C_2 = C_3 = 4$

Independent Paths

- An independent path is a path that introduces at least one new set of processing statement or a new condition. In the flow chart, an independent path must move along at least one edge that has not been traversed. For example:
  - Path 1: 13⇒12⇒1⇒11.

- Path 2: 13→12→1⇒2⇒3⇒6⇒7⇒9⇒10⇒12→1→11.
- Path 3: 13→12→1→2→3→6⇒8⇒9→10→12→1→11.
- Path 4: 13→12→1→2→3⇒4⇒5⇒10→12→1→11.

- These 4 independent paths form a basis set.
- The basis set is not unique.
- c.f. degrees of freedom, rank of matrix, independent vectors, coordinates system, and etc.

Independent Paths (cont.)

- Path 5 : 13→12→1→2→3→6→7→9→10→12→1→2→3→4→5→10→ 12→1→11 is not independent as it has no new edges.

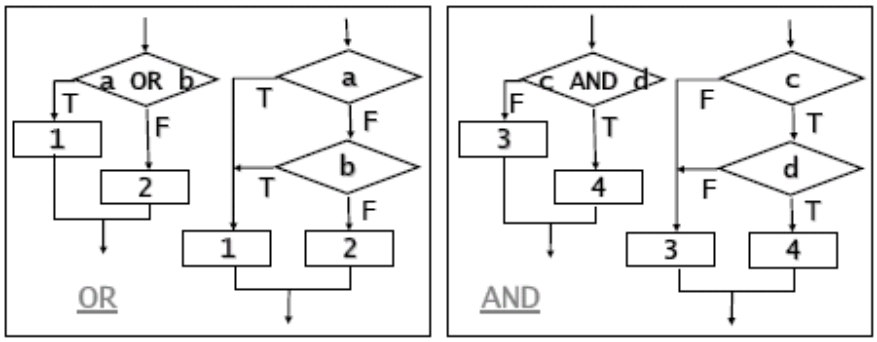
- **NEXT** : Design 4 test cases corresponding to the 4 independent paths of the basis set chosen.

Test Case No.	Path To Test	Input	Expected Output	Remarks
1	Path 1	...	...	...
2	Path 2	...	...	...
3	Path 3	...	...	...
4	Path 4	...	...	...

Compound Decision

- Compound decision made up of " AND " and " OR " operator must be broken up into simple decisions in performing basis path testing. For example:

```
if (a OR b) { block1 } else { block2 }
if (c AND d) { block4 } else { block3 }
```



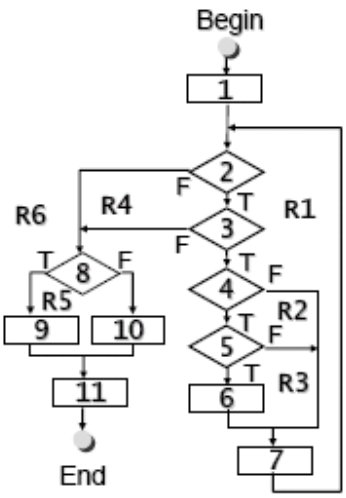
(Assume short–circuit evaluation for compound condition.)

Basis Path Method– A Complete Example

Requirement: The program computes the average of 100 or fewer numbers of an array that lie between an upper and lower bounds. The input is terminated by–999.

```
double average (int[] values, int min, int max) {
1  int sum=0, count=0, item=0;
   double average = 0.0;
     while (values[item] != -999 && item < 100 ) {
       if (values[item] >= min && values[item] <= max) {
6        sum += values[item];
         count++;
       }
7      item++;
     }
10   if (count > 0) average = (double)sum/count;
     else average = -999;
11   return average;
   }
```

Basis–Path Example (Cont.)



- **Step 1:** Draw the flow graph.
- **Step 2:** Find Cyclomatic complexity.
  C_1 = R (Region)= 6
  C_2 = P (Decision) + 1 = 5 + 1 = 6
  C_3 = E – N + 2 = 17 – 13 + 2 = 6.
  $C = C_1 = C_2 = C_3 = 6$
- **Step 3:** Find a basis set of independent paths.
  Path 1: 1⇒2⇒8⇒9⇒11
  Path 2: 1→2→8⇒10⇒11
  Path 3: 1→2⇒3⇒ 8→9→11
  Path 4: 1→2→3⇒4⇒7⇒2→...
  Path 5: 1→2→3→4⇒5⇒7→2→...
  Path 6: 1→2→3→4→5⇒6⇒7→2→...
  (Note: The basis set is not unique.)

- **Step 4:** Design test cases for each of the independent paths in the basis set chosen.

Path 1 Test Case:

Input: values={3, 5, 9, –999}, min=0, max=100
Expected output: (3+5+9)/3.
Purpose: to test correct Averaging.
Remark: Path 1 cannot be tested alone.

Path 2 Test Case:
Input: values={– 999}, min=0, max=100
Expected output: –999.
Purpose: to produce average= –999.
Path 3 Test Case:
Input: values={3, 33, …, 76} (101 numbers), min=0, max=100
Expected output: average of first 100 numbers.
Purpose: average only the first 100 valid numbers.
Path 4 Test Case :
Input: values={67, — 2, 23, —999}, min=0, max=100
Expected output: (67+23)/2.
Purpose: testing lower bound.
Path 5 Test Case :
Input: values={7, 32, 105, 86, 11, 2, –999}, min=0, max=100
Expected output: (7+32+86+11+2)/5.
Purpose: testing upper bound.
Path 6 Test Case :
Input: values={17, 34, 83, 39, – 999}, min=0, max=100
Expected output: (17+34+83+39)/4.
Purpose: proper averaging.

"Software engineers considerably underestimate the number of tests required to verify a straightforward program."

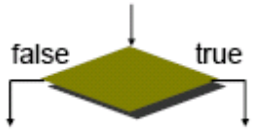
Basis Path Method

- Although basis path methods is simple and highly effective to guarantee that all statements and decisions in the program are executed at least once, it is not "sufficient" to detect many classes of semantic errors. For example, suppose `(values[i] > min)` is incorrectly used instead of `(values[i] >= min)` in the previous example and no test case having a `value[i]` of `min`.

- Basis– path method focuses on the "typical values" of each of the independent paths. Additional test cases (on top of basis– path test cases) needed, especially to exercise the boundary conditions.

White– box Method 2– Condition Testing

- Condition Testing is a test case design method that exercises the logical conditions (or oolean conditions).



- A simple condition is either:
  - A oolean variable which has value of either true or false (e.g., in C/C++, zero or–zero, in Java

oolean type).
- A relationship expression of the form `E1 <relational-operator> E2`where E1 and E2 are some arithmetic expressions and the relational–operator can be either <, <=, >, >=, ==,!=.
- Possibly preceded with a `NOT (!)`operator.

- A compound condition is composed of two or more simple conditions joined by oolean operators of `AND (&&)` or `OR (||)`.

Condition Testing

- A condition may include oolean operator (`AND` , `OR` , `NOT`), oolean variable, parentheses, relationship operator (, >= , , <= , == , !=), and arithmetic expression. E.g. `if (!done && ((i < 100) || ((j+k) > 10))) {…}`
- Hence, errors in a condition may cause by:
  oolean operator error (e.g., `AND` instead of `OR`).
  Relational operator error (e.g., >= instead of >).
  oolean variable error (e.g., wrong variable used).
  Parenthesis error (e.g., missing or wrong parenthesis).
  Arithmetic expression error (e.g., `j+k` instead of `j-K`).
- Condition Testing focus on testing the condition, the methods available are:
  1. Branch Testing
  2. Domain Testing
  3. Branch and Relational Operator (BRO) Testing

Branch / Domain Testing

- **Branch Testing** : For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once. (These tests are carried out in the Basis–Path method).
- **Domain Testing**:
  - For a relational expression `E1 <relational operator> E2`, three tests are designed for `E1==E2`, `E1>E2` , `E1<E2` . For example, `Condition: (X > 5)` 3 test cases: `X=5` , `X=3<5` , and `X=8>5` needed.
  - For a oolean expression with `n` oolean variables, all `2n` possible tests are required. (Only possible if n is small!). For example:
    `C = (B1 && B2) || (B3 && B4)`
    16 test cases needed for all combinations of `B1` , `B2` , `B3` , `B4`.

Branch and Relational Operator (BR0)

- Domain testing with `2n` tests is not feasible if the number of oolean variables `n` is large. BRO technique reduces the number of test cases.
- **Example 1**:

Condition `C: B1 AND B2`, where `B1` and `B2` are two oolean variables. The constraint set is `{(t,t),(t,f),(f,t)}(f,f)` is redundant, because if `B1`, `B2`, and/or operator `AND` is incorrect,

one of the 3 tests in the above constraint set will fail. (Prove it!) Design 3 test cases corresponding to the 3 members in the constraint set. (Recall that the Cyclomatic Complexity is 3.)

- **Example 2**:

Condition C: B1 AND (E1==E2), where B1 is a oolean variable, and E1 and E2 are arithmetic expressions.
Let B2 be (E1==E2), from Example 1, the constrain set for B1 AND B2 is {(t,t), (t,f), (f,t)}.
"t" for (E1==E2) is "=".
"f" for (E1==E2) is either ">" or "<".
Hence, the constrain set for C is

{(t,=),(t,>),(t,<),(f,=)}. Four test cases needed.
(t,t) (t,f) (f,t)

Note: Complete domain testing requires 2x3 = 6 test cases.

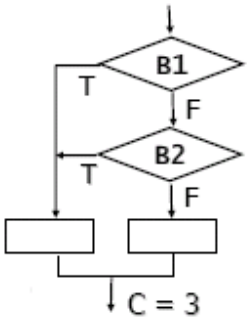
- **Example 3**:

Condition C: (E1>E2) AND (E3==E4), where E1, E2, E3 and E4 are arithmetic expressions.
Let B1 be (E1>E2) and B2 be (E3==E4). From Example 1, the constrain set for B1 AND B2 is {(t,t),(t,f),(f,t)}.
"t" for (E1>E2) is ">", "f" is either "<" or "=".
"t" for (E3==E4) is "=", "f" is either ">" or "<".
Hence, the constrain set for C is

{(>,=),(>,>),(>,<),(<,=),(=,=)}. Five test cases.
(t,t) (t,f) (f,t)

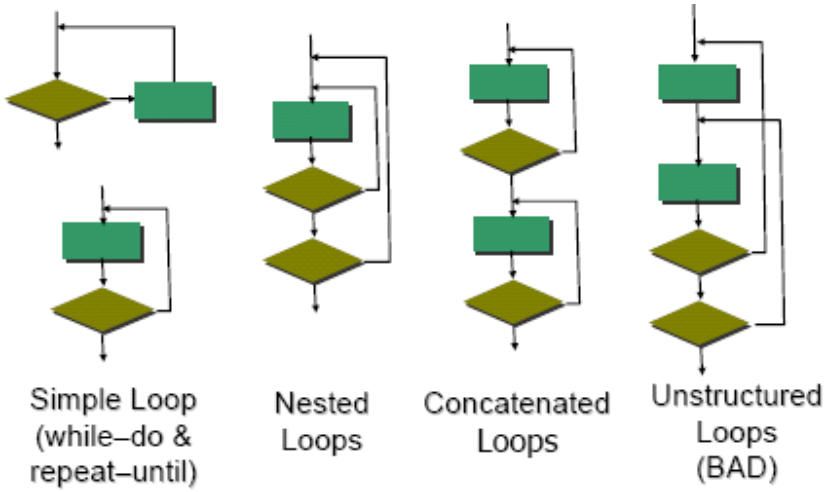
Note: Complete domain Testing requires 3x3 = 9 test cases.

- **Example 4**: Condition C: B1 OR B2 where B1 and B2 are oolean variables.
  The constrain set is {(f,f),(f,t),(t,f)}.
  (t,t) is redundant, because if B1, B2, and/or operator OR is incorrect, one the 3 tests in the above constraint set will fail. Note: Complete domain testing requires 4 tests cases.



"Even if your decide against condition testing, you should spend time evaluating each condition in an effort to uncover errors. This is a primary hiding place for bugs!"

White–box Method 3– Loop Testing



Simple Loop Testing

- Suppose n is the maximum number of allowable passes thru the simple loop, the test case are:
  - Skip the loop entirely.
  - Only one pass through the loop.
  - Two pass through the loop.
  - m pass through the loop, where m < n represents a "typical" value.
  - n–1, n, n+1(impossible) passes through the loop.

- That is, test for 0, 1, 2, a typical value m, n–1, n, n+1.

Nested Loop Testing

- Extending simple loop testing approach to nested loop may not be practical.
- To reduce the number of tests:
  - Start at the innermost loop. Set all other loops to minimum values.
  - Test the min+1, typical, max–1, and max for the innermost loop, while holding the outer loop at their minimum values.
  - Move outward, conducting test for the next loop, but keeping all the outer loops at minimum values and the inner loops at "typical" values.
  - Continue until all loops have been tested.

Other Loops

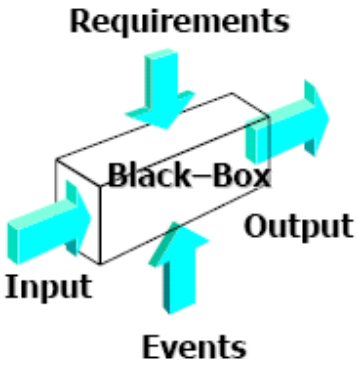
- Concatenated Loops: If the loops are independent, follow simple loop testing approach. If the second loop depends on the first loop (e.g., the initial value of the second loop), use the nested loop testing approach.

- Unstructured Loops: Bad! Rewrite!

"Complex loop structure are another hiding place for bugs. It is worth spending time designing tests that fully exercise loop structure."

6.14 Black–box Testing

- Focus on functional requirements.
- Not an alternative to white–box testing, but a complementary to uncover a different class of errors than white–box methods.
- Black–box testing attempts to find the following type of errors:
  - incorrect or missing functions.
  - interface errors.
  - errors in data structure or external databases.
  - behavior or performance errors.
  - initialization and termination errors.
  - others.

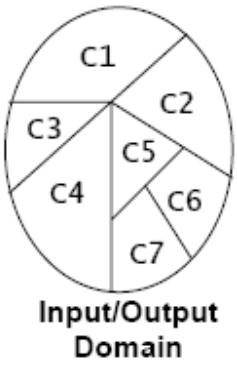


- The following black–box methods will be discussed:
  1. Equivalent Partitioning.
  2. Boundary Value Analysis (BVA).
  3. Comparison Testing.

Black–box Method 1: Equivalence Partitioning

- A black–box testing method that divide the input domain into equivalent classes . A representative test case is designed to test the entire class.

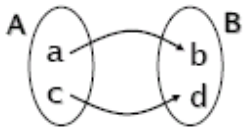
- **Rationale** : An ideal test case uncovers a class of errors that might otherwise require many arbitrary test tests.



1. Partitioning the input/output domain into equivalence classes.
2. Design one representative test case for each class.

Equivalence Partition

- "Equivalence Partition" is defined in SET theory.
  - A relation ρ on two sets Aand B is a subset of the Cartesian product AxB. E.g., aρb, cρd where a,c ∈Aand b,d∈B



  - Relation can be defined on the same set A.
  - A relation ρ on a set A is reflective if aρa∀a∈A.
  - A relation ρ on a set A is symmetric if aρb⇒bρa ∀a,b ∈A.
  - A relation ρ on a set A is transitive if aρb and bρc⇒aρc∀a,b,c∈A.
  - A relation that is reflective, symmetric and transitive is an equivalence relation. E.g., "=" on N (Integers) is an equivalent relation but ">" on N is not.
  - An equivalence relation partitions the set into disjoint equivalence classes. E.g., the relation ρ={(a,b): a,b ∈N, a+b is an even number}has two partitions {0,2,4,..} and {1,3,5,...}.

"Guidelines" on Partitioning

The textbook suggests 4 guidelines in partitioning:

1. If input specifies a range , partition into 1 valid and 2 invalid classes. Eg. If input is in the range of [0,100] , the valid class is 0<=x<=100 , the invalid classes are x<0 and x>100.
2. If input specifies a specific value , partition into 1 valid and 2 invalid classes. Eg. If input x=5 , the valid class is x=5 , the invalid classes are x<5 and x>5.
3. If input specifies a member of a set , partition into 1 valid and 1 invalid class. Eg. If input x belongs to the set Months={"Jan",..,"Dec"} , the valid class is x∈ Months , the invalid class is x∉ Months.
4. If input specifies a Boolean , partition into 1 valid and 1 invalid classes, corresponds to the two states.

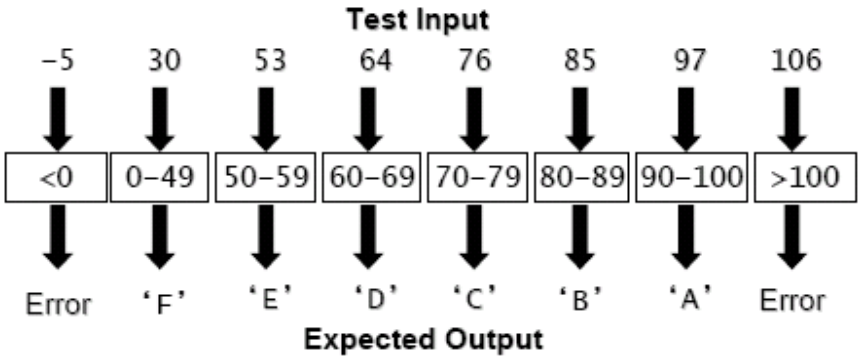
I add the fifth:

5. Use your engineering common sense, knowledge & intuition!

Partitioning the Trivial Example

- The input domain can be partitioned into 8 equivalence classes. 8 test cases are designed for the 8 classes.

- Note that the "Guidelines" suggested in the textbook are merely "guidelines". Use intuition to do the partitioning.



Equivalence Partitioning– Example 1

- An automated banking system requires the following inputs:
  - Userid – 6 or more digit alphanumeric string.
  - Password – 6–digit number, not beginning with 0.
  - Command – {check, deposit, pay bill, ...}
- The input domain can be partitioned as follows:

- Userid :
  Present (Boolean): (1) valid: present, (2) invalid: not present If present, 6 or more digit (Boolean): (1a) valid: >=6, (1b) invalid: <6. (Class 1 is further partitioned into 1a and 1b.)

- Password :
  Present (Boolean): (3) valid: present, (4) invalid: not present If present, 6–digit not beginning with 0 (Range): (3a) valid: 100000–999999, (3b) invalid: <100000, (3c) invalid: >999999

- Command (Member of a set): (5) valid: member, (6) invalid: not member.

Example 1 (cont.)

Input			Class	Remarks
Userid	Present	>= 6 digits	1a	Valid
		< 6 digits	1b	Invalid
	Not Present		2	Invalid
Password	Present	100000–999999	3a	Valid
		< 100000	3b	Invalid
		> 999999	3c	Invalid
	Not Present		4	Invalid
Command	Valid command		5	Valid
	Invalid Command		6	Invalid

Disjoint Partitions of Input Domain

Equivalence Partitioning– Example 2

- Design test cases for a search routine, which searches a integer array using a search key.

- Inputs:
  `int[] array` : the integer array to be searched.
  `int key` : search for this key (or element).

- Outputs:
  `boolean found` : `true` if `key` found in `array` , `false` otherwise.
  `int index` : If found, the first index of the `key` in the `array`
- Pre–condition:
  - The `array` is not empty, contains at least one element.
  - The `array` is sorted in ascending order.
- Post–condition:
  - If `key` is found, `found=true`, `array[index]=key`.
  - If `key` is not found, `found=false`, `index` is undefined.

Example 2 (Cont.)
The input domain can be partitioned as follows:

Input		Class No
Array	**Key**	
Single–element	In array	1
	Not in array	2
More than 1 element	First element in array	3
	Last element in array	4
	In array, not the first nor last.	5
	Not in array	6

Note that the pre–condition excludes empty array.

- The black box test cases with the test data and expected result are as shown:

Input		Expected Result	Classes Covered
Array	**Key**		
{3}	3	Found=true, Index=0	1
{3}	5	Found=false, Index undefined	2
{3, 5, 8, 9}	3	Found=true, Index=0	3
{3, 5, 8, 9}	9	Found=true, Index=3	4
{3, 5, 8, 9}	5	Found=true, Index=1	5
{3, 5, 8, 9}	6	Found=false, Index undefined	6

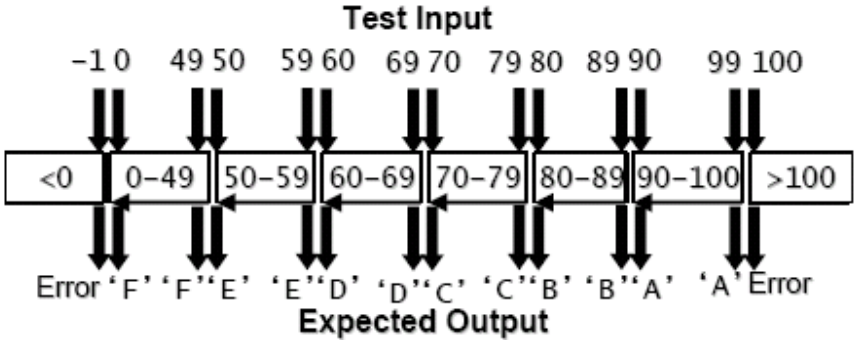
Black–box Method 2: Boundary Value Analysis

- Boundary Value Analysis (BVA) extends the equivalence partitioning by focusing on the boundaries of the input domain rather than its typical values.
- Guidelines for BVA:
  1. If input condition specifies a range between a and b, test cases should include a, b, and values just above and below a and b.
  2. If input condition specifies a number of values, test cases can exercise the minimum and maximum numbers, and values just above the maximum and just below the minimum.
  3. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output.
  4. If internal program data structures have boundaries (e.g., array, buffer), test cases should be designed to test these boundaries.

BVA for our Trivial Example

- The test cases for boundary value analysis for the trivial example is shown.

- We usually perform the equivalent partitioning first, then design test cases around the boundaries of the partitions instead of using "typical value" in equivalent–partition testing.



Black–box Method 3: Comparison Testing

- Comparison testing is a black box testing method for safety critical systems (eg. Aircraft avionics, automobile braking systems), where reliability is absolutely critical.
- In such applications, redundant (i.e., duplicate) hardware and software are often used to minimize the possibility of error and to achieve fault– tolerant.
- Separate software teams could independently implement the redundant systems.
- In comparison testing, test cases designed using black–box techniques (such as equivalent class) are applied to both versions of software that are independently developed, the conformance to the specifications are then compared.