



SOFTWARE ENGINEERING

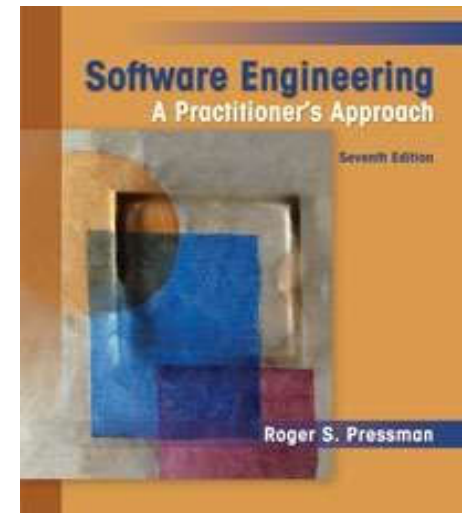
Department of Software Engineering

Outline

- Overview
- Goals
- Chapters
 - Chapter I – Software & Software Engineering
 - Chapter II – Requirement s Analysis & Software Specification
 - Chapter III – Software Design
 - Chapter IV – Guaranty, Testing & Maintaining
 - Chapter V – Programming effectively
- Review

Overview

- Software Engineering **vs.** Software Technology
- General processes (a series of predictable steps)
 - requirement analysing
 - modelling (specification)
 - designing
 - checking the quality (SQA)
 - implementing
 - testing
 - maintaining
- Not only building software systems but also building them ***effectively***
- Textbook: ***Software Engineering: A Practitioner's Approach*** by Roger Pressman, Mc Graw-Hill, 2009 (7/e, 1/e 1982)



Goals

- **Equipes IT students with**
 - basic knowledge of software developing methods
- **Helps IT students**
 - to developing software on principle through methods, procedures and tools
 - to build software effectively

CODER



SOFTWARE ENGINEER

Chapter I

Software & Software Engineering

- What is software? Software categories?
- Software developing patterns

Section I – Software

- Definition “*software*”
- Growing process
- Software characteristics
- Types of software
- Challenges

Definition “software”

- Software is a **collection of instructions** that can be ‘run’ on a computer. These instructions tell the computer what to do. ¹
- *“Computer software, or simply software, refers to the non-tangible components of computers, known as computer programs. The term is used to contrast with computer hardware, which denotes the physical tangible components of computers.” [Wikipedia]*

Use over time for: software



(1) <http://www.igcseict.info/theory/1/hwsw/>

Definition “*software*”

- **Software is**

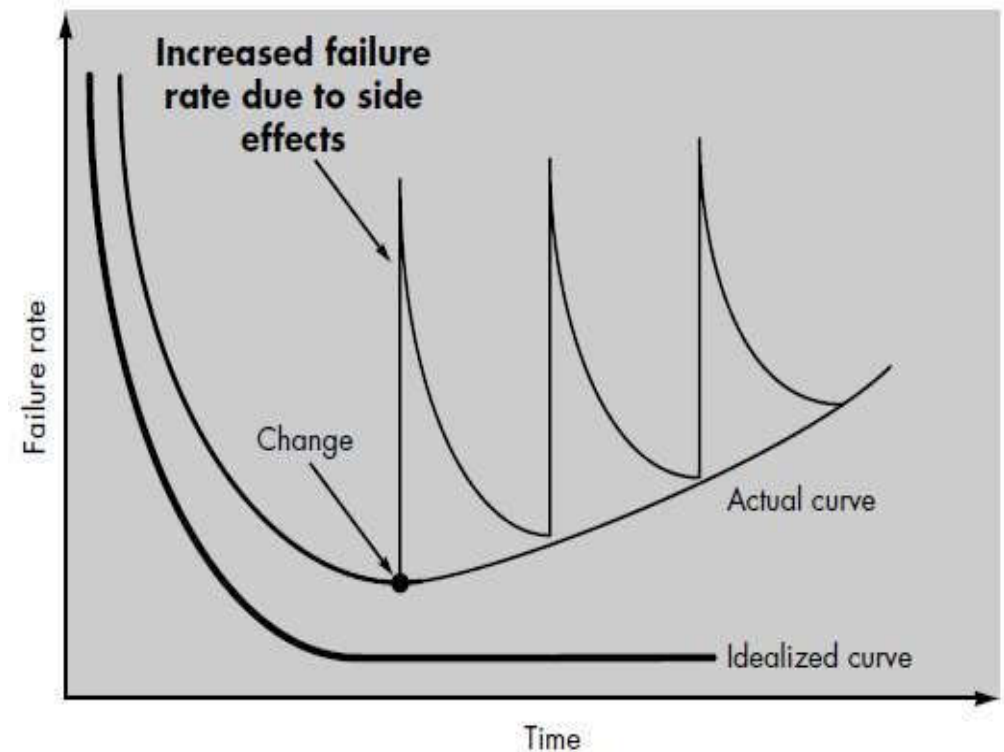
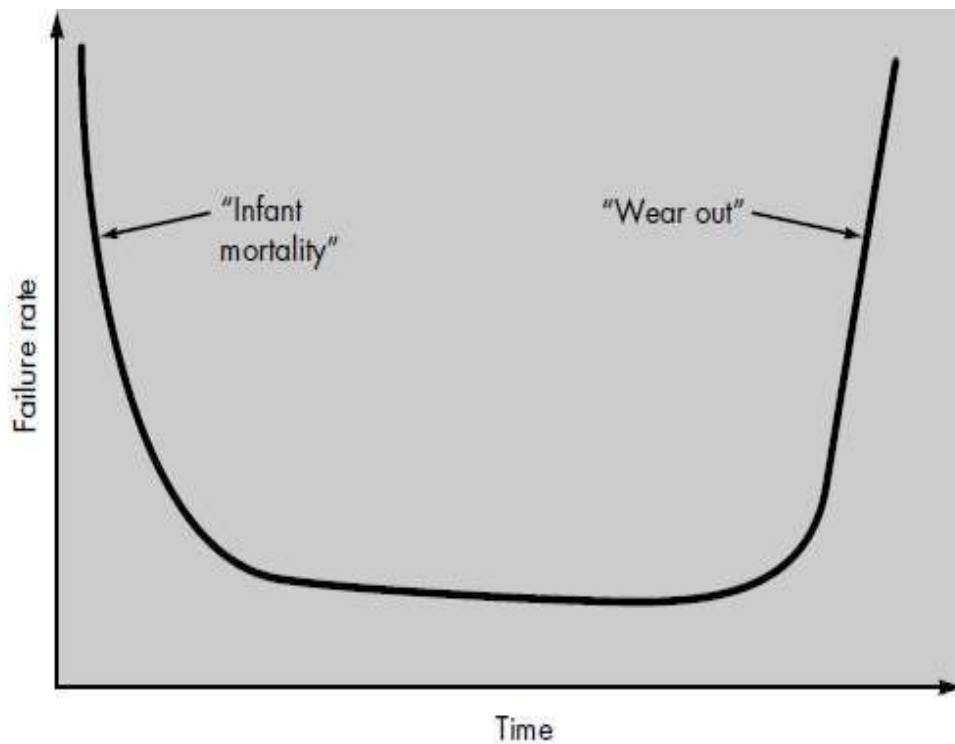
- (1) instructions (computer programs) that when executed provide desired function and performance,
- (2) data structures that enable programs to adequately manipulate information,
- And (3) documents that describe the operation and use of the programs.

Growing process

- **1950 – 1960:** early days of computing
- **1960 – middle 1970:** software crisis
- **middle 1970 – 1990:** network, cheap hardware
- **1990 – present:** O-O, expert systems, artificial intelligence (AI), the fourth generation technics, cell-phone...

Software Characteristics

- Software is developed or engineered, it is not manufactured in the classical sense
- Software doesn't "wear out"



Software Characteristics

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

Attributes of good software

● Maintainability

- Software must evolve to meet changing needs

● Dependability

- Software must be trustworthy

● Efficiency

- Software should not make wasteful use of system resources

● Usability

- Software must be usable by the users for which it was designed

Types of software

- **System Software.** A collection of programs written to service other programs at system level. For example, compiler, operating systems.
- **Business Software.** Programs that access, analyze and process business information.
- **Real-time Software.** Programs that monitor/analyze/control real world events as they occur.
- **Engineering and Scientific Software.** Software using “number crunching” algorithms for different science and applications. System simulation, computer-aided design.
- **Embedded Software.** Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. It has very limited and esoteric functions and control capability.
- **Artificial Intelligence (AI) Software.** Programs make use of AI techniques and methods to solve complex problems. Active areas are expert systems, pattern recognition, games

Types of software

- **Personal computer software.** Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.
- **Web-based software.** The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).
- **Software Tools and CASE environment.** Tools and programs that help the construction of application software and systems. For example, test tools, version control tools.

Challenges

- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

IBM survey of 24 companies developing distributed systems:

- 55% of the projects cost more than expected
- 68% overran their schedules
- 88% had to be substantially redesigned

Challenges

- **Software product size is increasing exponentially**
 - faster, smaller, cheaper hardware
- **Software is everywhere:** from TV sets to cell-phones
- **Software is in safety-critical systems**
 - cars, airplanes, nuclear-power plants
- **We are seeing more of**
 - distributed systems
 - embedded systems
 - real-time systems
 - These kinds of systems are harder to build
- **Software requirements change**
 - software evolves rather than being built

Section II – Software Engineering

- **Definition *Software Engineering* (SE)**
- **Why is SE?**
- **What is SE?**
- **Software Models**

Definition SE

- “A systematic approach to the analysis, design, implementation and maintenance of software.”

(The Free On-Line Dictionary of Computing)

- “The systematic application of tools and techniques in the development of computer-based applications.”

(Sue Conger in The New Software Engineering)

- “Software Engineering is about designing and developing high-quality software.”

(Shari Lawrence Pfleeger in Software Engineering - The Production of Quality Software)

Definition SE

- **Definition proposed by Fritz Bauer**

- “[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.”

- **Definition developed by IEEE**

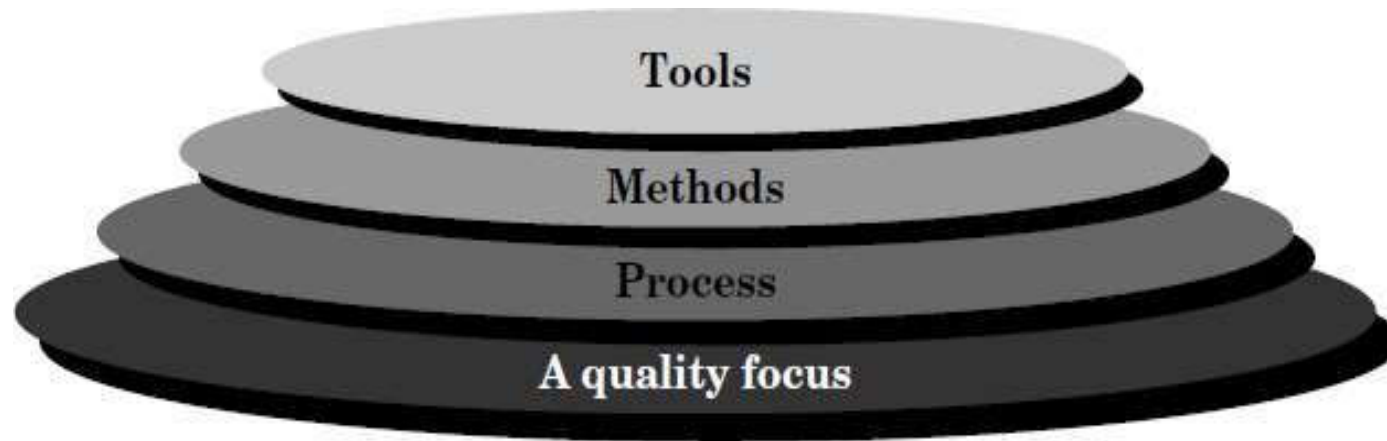
- “Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).”

Why SE?

● Major Goals

- To increase software productivity and quality
- To effectively control software schedule and planning
- To reduce the cost of software development
- To meet the customers' needs and requirements
- To enhance the conduction of software engineering process
- To improve the current software engineering practice
- To support the engineers' activities in a systematic and efficient manner

What is SE?



- **Software engineering is a layered technology**
 - **Process**
 - **Methods**
 - **Tools**

What is SE?

● Process

- Glue – holds the technology layers together
- Foundation for software engineering
- Enables timely development
- Forms the basis for management control of software projects
- Establishes the context in which technical methods are applied
- Work products are produced
- Milestones are established
- Quality is ensured
- Change is properly managed

What is SE?

● Methods

- Provide the technical “how to’s” for constructing software
- Tasks include communications, requirement analysis, design modeling, program construction, testing, and support.
- Rely on set of basic principles
 - To govern each area of the technology
 - Include modeling activities

What is SE?

- **Tools**

- Automated or semi-automated support for the process and methods

- **A quality focus**

- The bedrock
- Any engineering approach must rest on an organizational commitment to quality.
- Foster a continuous process improvement culture

SE phases

- SE work is categorized into three generic phases
 - Definition phase
 - Development phase
 - Support phase

SE phases

- **Definition phase: “WHAT”.** To identify
 - what information be processed,
 - what function and performance desired,
 - what system behavior expected,
 - what interfaces be established,
 - what design constraints exist,
 - and what validation criteria required.

>>The key requirements of the system and the software are identified.

SE phases

- **Development phase:** “HOW”. To define
 - how data are to be structured,
 - how function to be implemented within a software architecture,
 - how procedural details are to be implemented,
 - how interfaces are to be characterized,
 - how the design will be translated into a programming language (or nonprocedural language),
 - and how testing will be performed.

SE phases

- **Support phase:** “CHANGE” associated with
 - error correction,
 - adaptations required as the software's environment evolves,
 - and changes due to enhancements brought about by changing customer requirements.

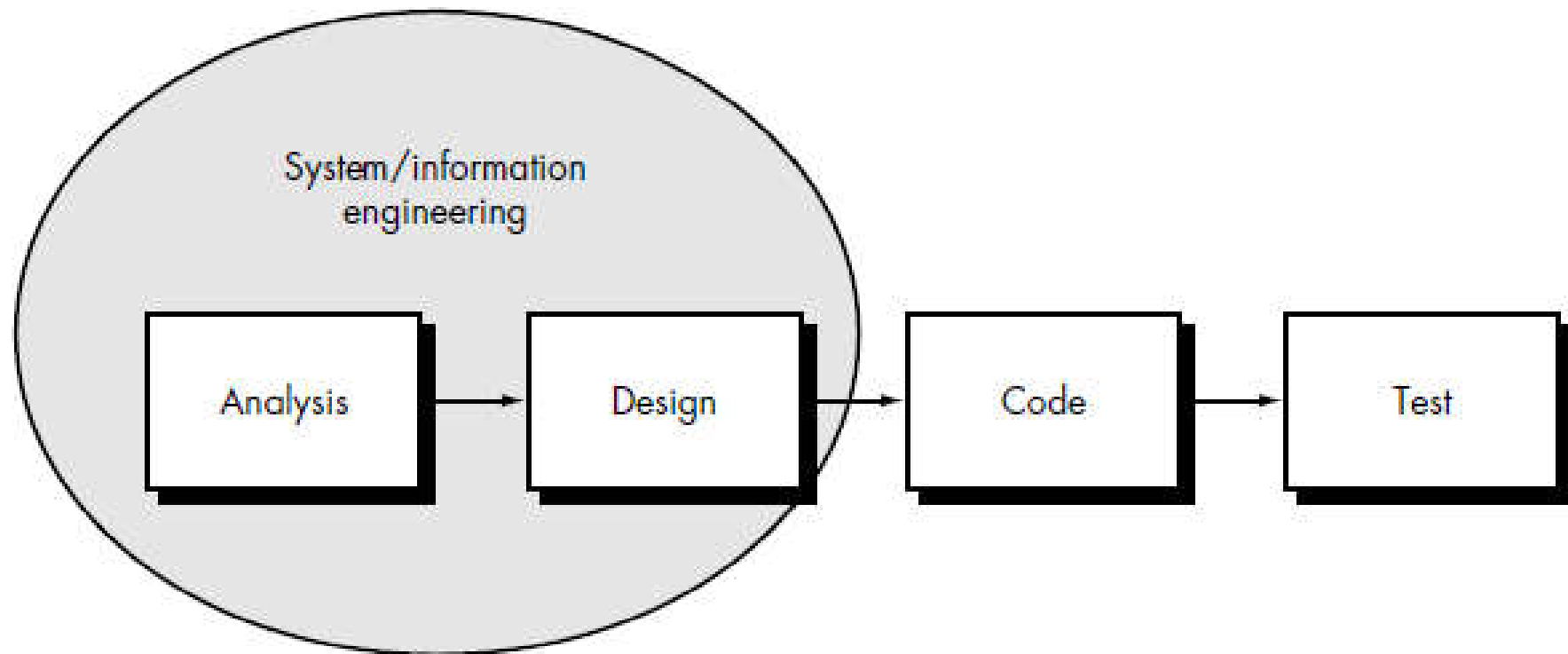
>>Four types of change are encountered during the support phase: **Correction, Adaptation, Enhancement, and Prevention.**

Model I

The linear sequential Model

- The Model
- Problems when applied

Linear sequential Model



Linear sequential Model

- *classic life cycle = the waterfall model = linear sequential model*
- suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support.

Linear sequential Model

- **System/information engineering and modeling**
- **Software requirements analysis**
- **Design**
- **Code generation**
- **Testing**
- **Support**

Linear sequential Model

- **System/information engineering and modeling**
 - software is always part of a larger system => establishing requirements for all system elements and then allocating some subset of these requirements to software
 - System engineering and analysis encompass requirements gathering at the system level with a small amount of top level design and analysis

Linear sequential Model

- **Software requirements analysis**

- The requirements gathering process is intensified and focused specifically on software
- the software engineer ("analyst") must understand the information domain for the software, required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customers

Linear sequential Model

● Design

- Software design is actually a multistep process that focuses on four distinct
 - attributes of a program: data structure, software architecture, interface representations,
 - and procedural (algorithmic) detail.
- The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

Linear sequential Model

● Code generation

- The design must be translated into a machine-readable form.
- The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Linear sequential Model

● Testing

- Once code has been generated, program testing begins.
- The testing process focuses on
 - the logical internals of the software, ensuring that all statements have been tested,
 - and the functional externals;
 - that is conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Linear sequential Model

● Support

- Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software)
- Change will occur because
 - errors have been encountered,
 - the software must be adapted to accommodate changes in its external environment (new operating system or peripheral device),
 - the customer requires functional or performance enhancements.
- Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one

Linear sequential Model problems

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Model II

The Prototyping Model

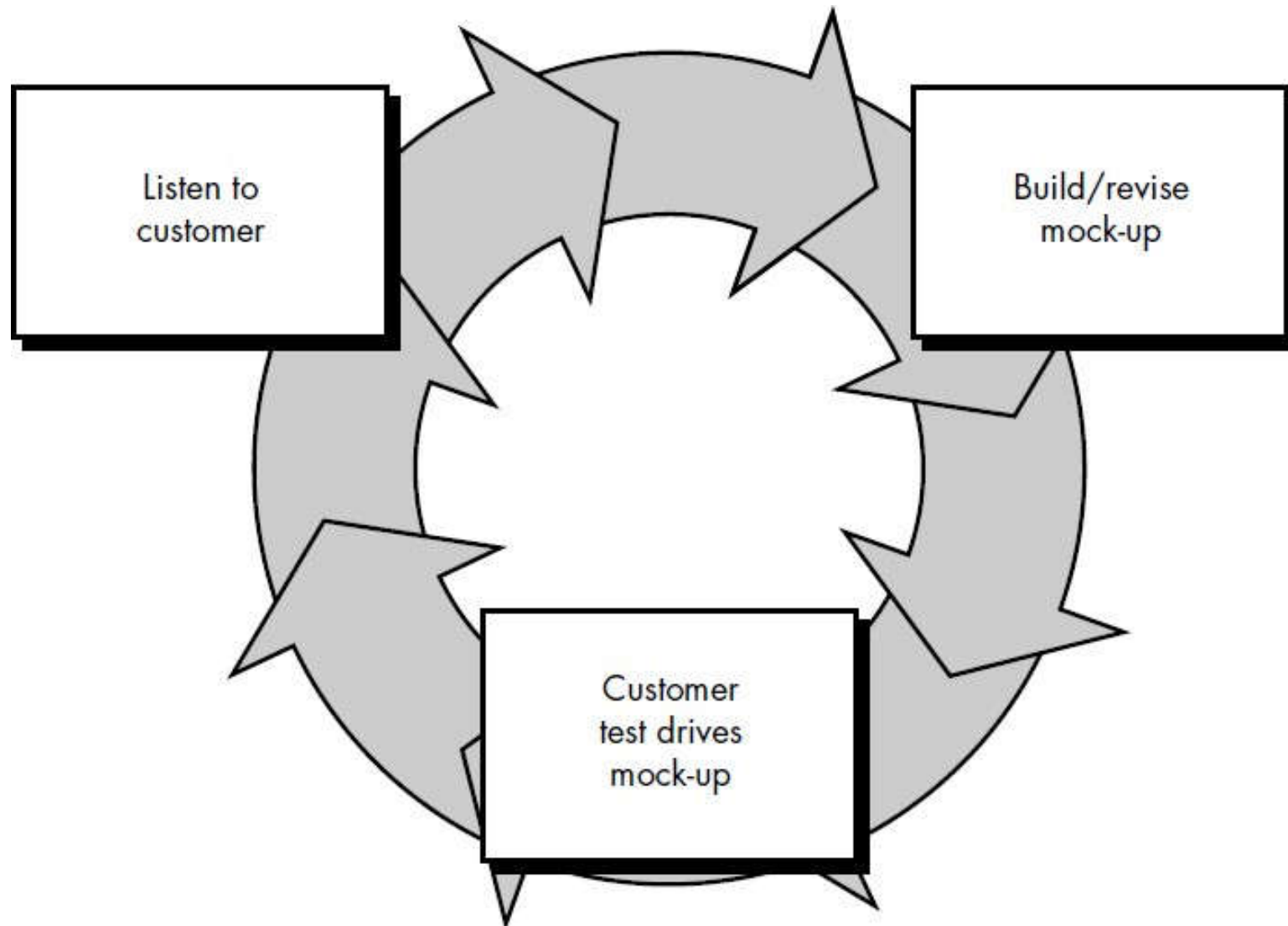
- Situations for applying
- The Model
- Problems when applied

Situations for applying

- a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements
- 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
- etc.,

>> *Prototyping paradigm* may offer the best approach

Prototyping Model



Prototyping Model

- **Requirements gathering:** Developer & customer define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- **A “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); leads to the construction of a prototype.
- 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 tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done

Prototyping Model Problems

1. No one has considered overall software quality or long-term maintainability.
2. 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.
3. The developer often makes implementation compromises in order to get a prototype working quickly. 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.

Prototyping Model Problems

- The customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements
- The actual software is engineered with an eye toward quality and maintainability

Model III

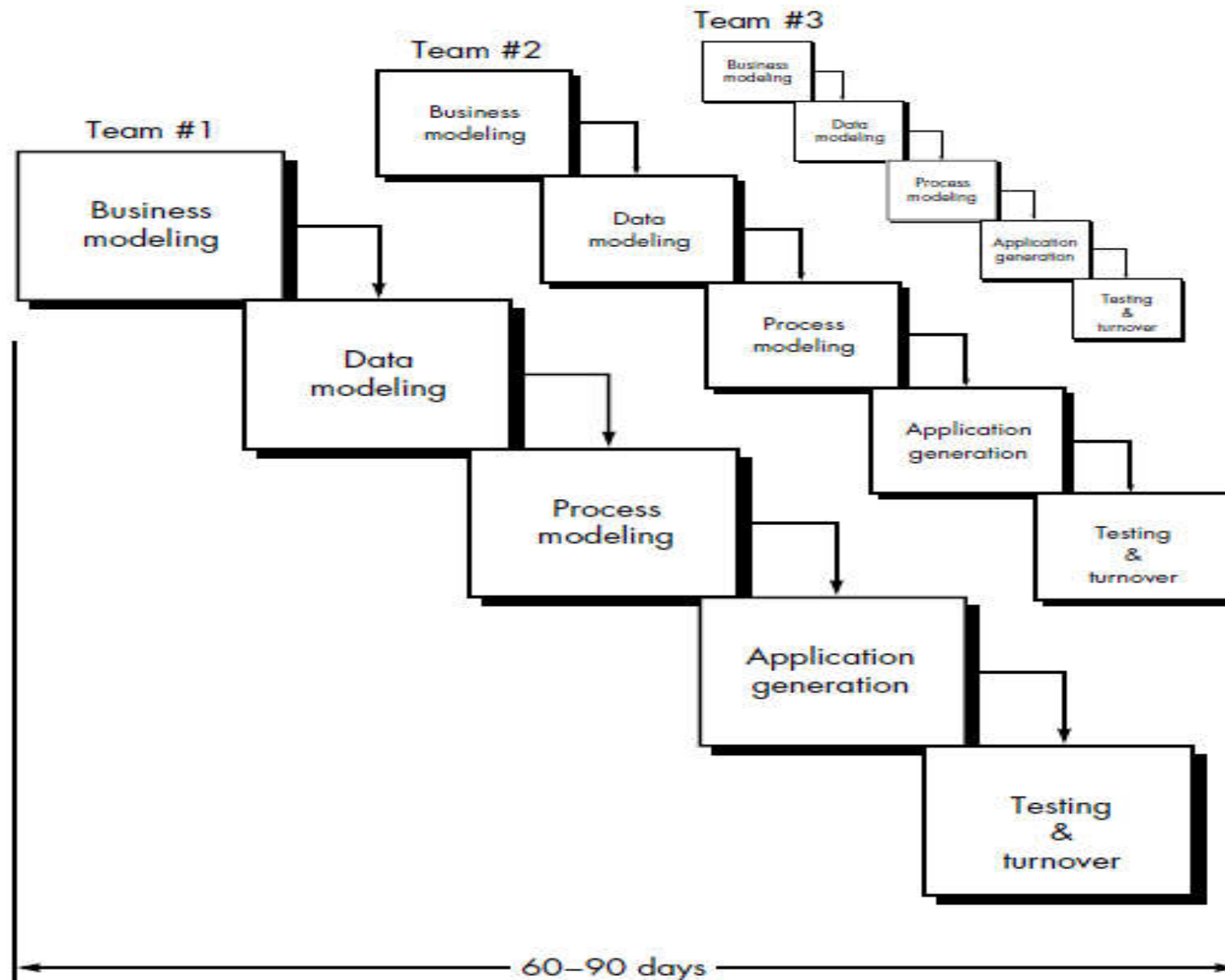
The RAD Model

- The Model
- Problems when applied

RAD Model

- *Rapid application development* (RAD) is an incremental software development process model that emphasizes an extremely short development cycle
- RAD model is a “high-speed” adaptation of the linear sequential model in which rapid development is achieved by using *component-based construction*
- If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a “fully functional system” within very short time periods (e.g., 60 to 90 days)

RAD Model



RAD Model

- **Business modeling.** The information flow among business functions is modeled in a way that answers the following questions:
 - What information drives the business process?
 - What information is generated?
 - Who generates it?
 - Where does the information go?
 - Who processes it?

RAD Model

- **Data modeling.** The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics (called *attributes*) of each object are identified and the relationships between these objects defined.

RAD Model

- **Process modeling.** The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

RAD Model

- **Application generation.** RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

RAD Model

- **Testing and turnover.** Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

RAD Model Candidate

- Candidate for RAD

- If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously). Each major function can be addressed by a separate RAD team and then integrated to form a whole.

RAD Model Problems

- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.

RAD Model Problems

- Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic.
- RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

Model IV

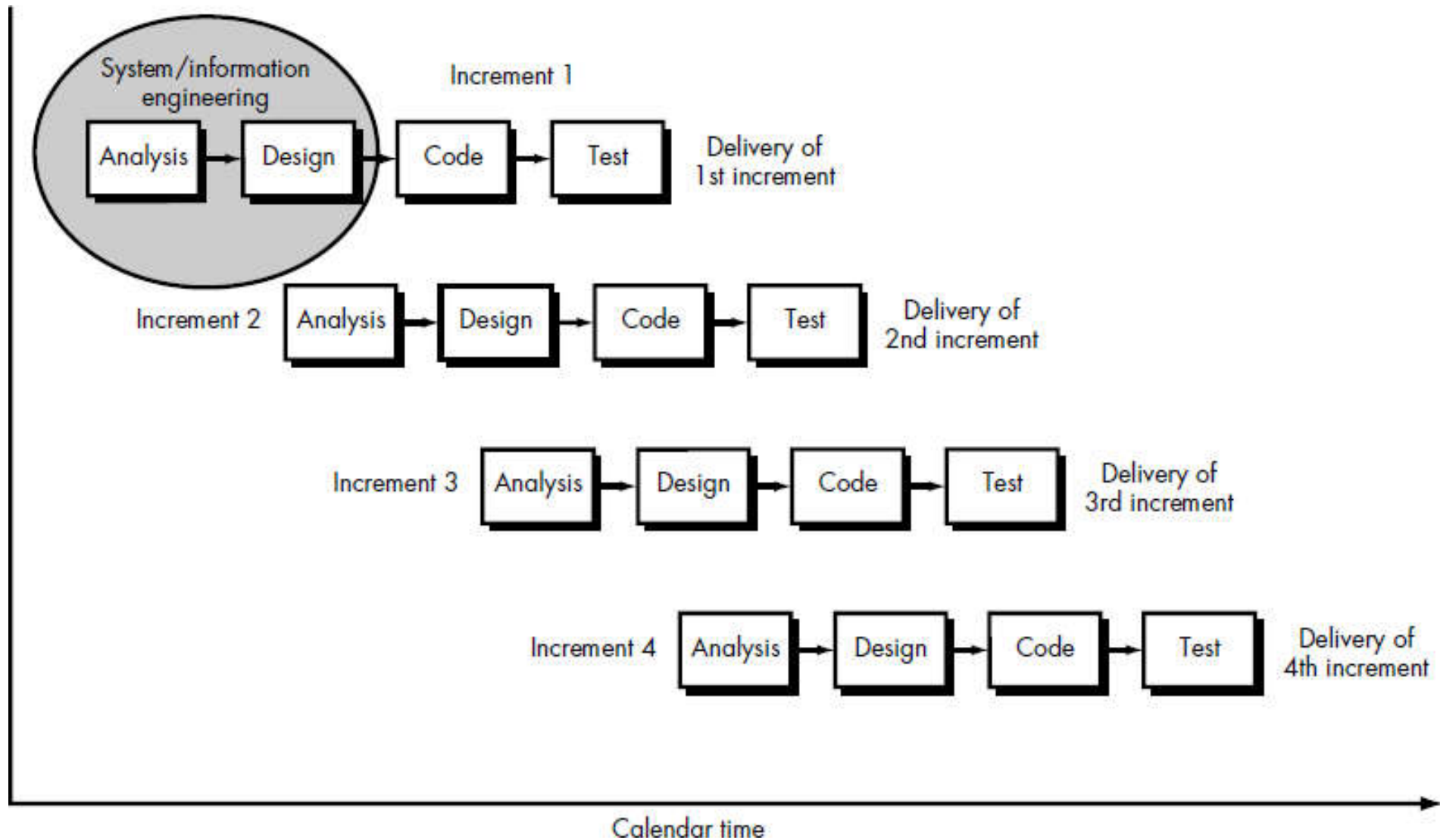
The Incremental Model

- The Model
- Advantages of the model

Incremental Model

- The *incremental model* combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping.
- The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software.
- The incremental model focuses on the delivery of an operational product with each increment

Incremental Model



Incremental Model

- For example, word-processing software developed using the incremental paradigm
 - 1st increment: deliver basic file management, editing, and document production functions;
 - 2nd increment: more sophisticated editing and document production capabilities;
 - 3rd increment: spelling and grammar checking;
 - 4th increment: advanced page layout capability.

Incremental Model

- 1st increment is *core product*, basic requirements are addressed, but many supplementary features 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.

Incremental Model Advantages

- Incremental development is useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Increments can be planned to manage technical risks. E.g., a major system might require the availability of new hardware under development. 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.

Model V

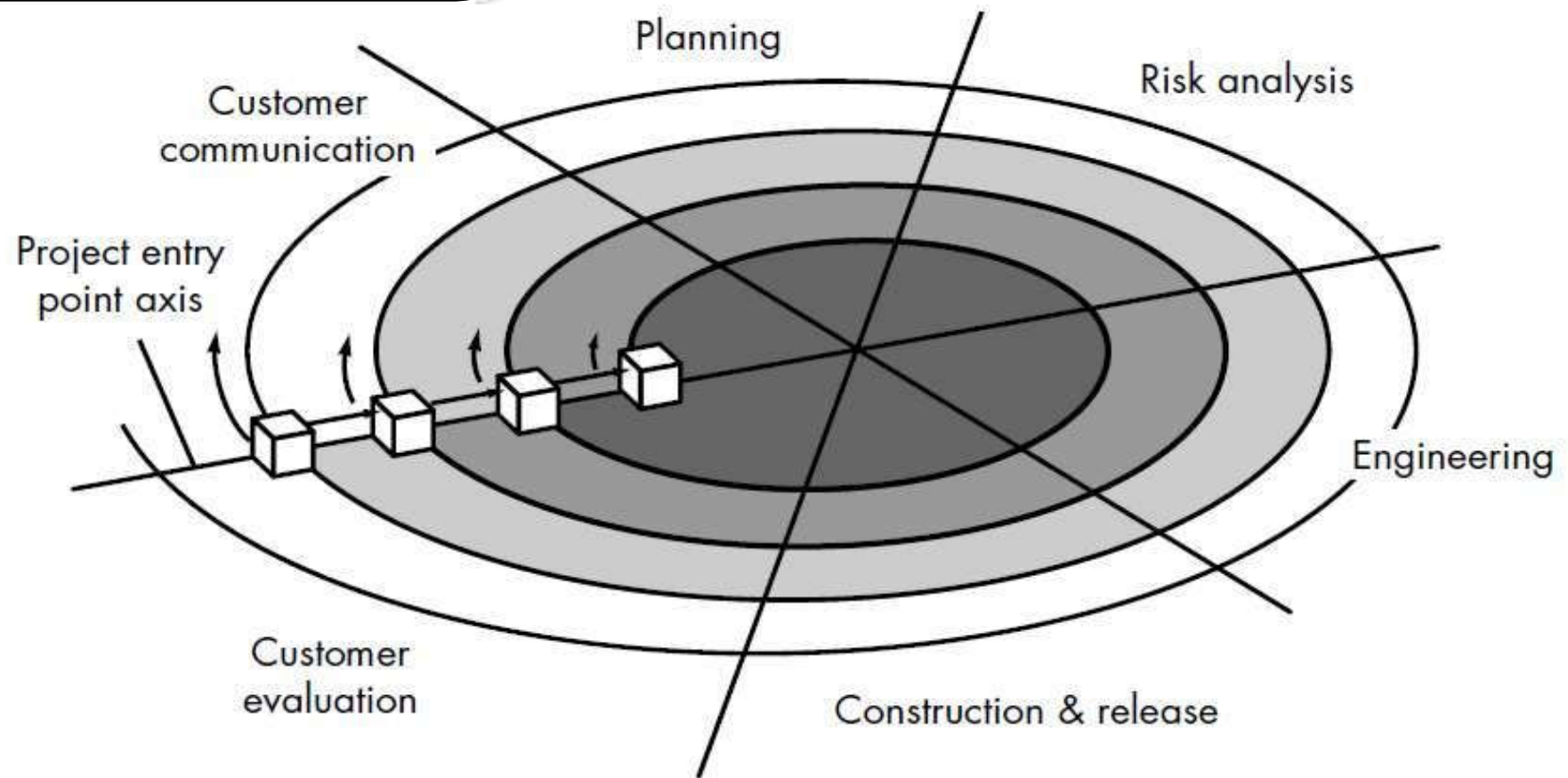
The Spiral Model





- The Model
- Advantages and disadvantages of the model

Spiral Model

- The Spiral Model 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

Spiral Model



-  Product maintenance projects
-  Product enhancement projects
-  New product development projects
-  Concept development projects

Spiral Model

- A spiral model is divided into 6 framework activities
 - **Customer communication** - tasks required to establish effective communication between developer and customer.
 - **Planning** - tasks required to define resources, timelines, and other project related information.
 - **Risk analysis** - tasks required to assess both technical and management risks.

Spiral Model

- **Engineering** - tasks required to build one or more representations of the application.
- **Construction and release** - tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- **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.

Spiral Model

- Beginning at the center, moves around the spiral in a clockwise direction
- The first circuit might result in the development of a product specification; subsequent passes might be used to develop a prototype, 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 from customer
- In addition, the project manager adjusts the planned number of iterations required to complete the software

Spiral Model

- Each cube placed along the axis can be used to represent the starting point for different types of projects.
- A “concept development project” starts at the core of the spiral and will until concept development is complete.
- If the concept is to be developed into an actual product, the process proceeds through the next cube (new product development project entry point) and a “new development project” is initiated.

Spiral Model Advantages

- The spiral model is a realistic approach to the development of large-scale systems and software.
- The spiral model uses prototyping as a risk reduction mechanism, enables the developer to apply the prototyping approach at any stage in the evolution of the product.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

Spiral Model Disadvantages

- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable
- 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

Model VI

Fourth Generation Techniques (4GT)

- The paradigm
- Advantages and disadvantages of the model

Fourth Generation Techniques

- 4GT encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level, then the tool automatically generates source code based on the developer's specification.
- 4GT paradigm focuses on the ability to specify software using *specialized language forms* or a *graphic notation* that describes the problem to be solved in terms that the customer can understand.

Fourth Generation Techniques

- A software development environment that supports the 4GT paradigm includes some or all of the following tools:
 - nonprocedural languages for database query,
 - report generation,
 - data manipulation,
 - screen interaction and definition,
 - code generation;
- 4GT tools have high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for Web-site creation using advanced software tools.

Fourth Generation Techniques

- 4GT begins with a requirements gathering step.
- For small applications, it may be possible to move directly from the requirements gathering step to implementation using a nonprocedural **fourth generation language (4GL)** or a model composed of a network of graphical icons.
- For larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used. The use of 4GT without design (for large projects) will cause the difficulties (poor quality, poor maintainability, poor customer acceptance)

Fourth Generation Techniques

- To transform a 4GT implementation into a product, the developer must
 - conduct through testing,
 - develop meaningful documentation,
 - and perform all other solution integration activities
- The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

4GT Advantages

- The use of 4GT is a viable approach for many different application areas. Coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problems.
- Data collected from companies that use 4GT indicate that the time required to produce software is greatly reduced for small and intermediate applications, and that the amount of design and analysis for small applications is also reduced.

4GT Disadvantages

- Current 4GT tools are not much easier to use than programming languages, the resultant source code produced by such tools is "inefficient," and the maintainability of large software systems developed using 4GT is open to question.
- The use of 4GT for large software development efforts demands as much or more analysis, design, and testing (software engineering activities) to achieve substantial time savings that result from the elimination of coding.

Chapter II

Requirements Analysis & Software Specification

- Requirements Engineering
- Requirements Analysis
- Analysis Principles
- Software Prototyping
- Specification

Requirements Engineering

- The requirements engineering process can be described in five distinct steps:
 - requirements elicitation
 - requirements analysis and negotiation
 - requirements specification
 - requirements validation
 - requirements management

Requirements Elicitation

- Requirements elicitation is difficult
 - *Problems of scope.* The boundary of the system is ill-defined or the customers specify unnecessary technical detail that may confuse,
 - *Problems of understanding.* The customers are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
 - *Problems of volatility.* The requirements change over time.

Requirements Elicitation

- Detailed guidelines for requirements elicitation:
 - Assess the business and technical feasibility for the proposed system.
 - Identify the people who will help specify requirements and understand their organizational bias.
 - Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.
 - Identify “domain constraints” (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.

Requirements Elicitation

- Detailed guidelines for requirements elicitation (cont.):
 - Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).
 - Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.
 - Identify ambiguous requirements as candidates for prototyping.
 - Create usage scenarios to help customers/users better identify key requirements.

Requirements Elicitation

- The work products include:
 - A statement of need and feasibility.
 - A bounded statement of scope for the system.
 - A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.
 - A description of the system's technical environment.
 - A list of requirements (organized by function) and the domain constraints that apply to each.
 - A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
 - Any prototypes developed to better define requirements.

Requirements Analysis & Negotiation

- Analysis

- categorizes requirements and organizes them into related subsets;
- explores each requirement in relationship to others;
- examines requirements for consistency, omissions, and ambiguity;
- and ranks requirements based on the needs of customers/users.

Requirements Analysis & Negotiation

- The following questions are asked and answered:
 - Is each requirement consistent with the overall objective for the system/product?
 - Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
 - Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?

Requirements Analysis & Negotiation

- The following questions are asked and answered (cont.):
 - Is each requirement bounded and unambiguous?
 - Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
 - Do any requirements conflict with other requirements?
 - Is each requirement achievable in the technical environment that will house the system or product?
 - Is each requirement testable, once implemented?

Requirements Specification

- A specification can be:
 - a written document,
 - a graphical model,
 - a formal mathematical model,
 - a collection of usage scenarios,
 - a prototype,
 - or any combination of these.
- For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.
- For smaller systems, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

Requirements Specification

- The *Software Requirement Specification (SRS)* is the final work product produced by the system and requirements engineer.
- It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element.
- The *System Specification* also describes the information (data and control) that is input to and output from the system.

System Modelling

- In order to fully specify what is to be built, you would need a meaningful model.
- It is important to evaluate the system's components in relationship to one another, to determine how requirements fit into this picture, and to assess the “aesthetics” of the system as it has been conceived.

Requirements Validation

- *Requirements validation* examines the specification to ensure that
 - all system requirements have been stated unambiguously;
 - that inconsistencies, omissions, and errors have been detected and corrected;
 - and that the work products conform to the standards established for the process, the project, and the product.

Requirements Validation

- The primary requirements validation mechanism is the *formal technical review*.
- The review team includes system engineers, customers, users, and other stakeholders.
- The review team examine the system specification, look for:
 - errors in content or interpretation,
 - areas where clarification may be required,
 - missing information,
 - inconsistencies,
 - conflicting requirements,
 - or unrealistic (unachievable) requirements.

Requirements Management

- *Requirements management* is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- Each requirement is assigned a unique identifier that might take the form:

<requirement type><requirement #>

where *requirement type* takes on values such as *F* = *functional requirement*, *D* = *data requirement*, *B* = *behavioral requirement*, *I* = *interface requirement*, and *P* = *output requirement*.

E.g., a requirement identified as F09 indicates a functional requirement assigned requirement number 9.

Requirements Management

- Once requirements have been identified, traceability tables are developed.

Requirement	Specific aspect of the system or its environment								
	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

Requirements Management

- Among many possible traceability tables are the following:
 - *Features traceability table*. Shows how requirements relate to important customer observable system/product features.
 - *Source traceability table*. Identifies the source of each requirement.
 - *Dependency traceability table*. Indicates how requirements are related to one another.
 - *Subsystem traceability table*. Categorizes requirements by the subsystem(s) that they govern.
 - *Interface traceability table*. Shows how requirements relate to both internal and external system interfaces.

Software Requirements Analysis

- Requirements analysis allows the software engineer to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.
- Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs.
- Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Software Requirements Analysis

- Software requirements analysis may be divided into five areas of effort:
 - (1) problem recognition,
 - (2) evaluation and synthesis,
 - (3) modeling,
 - (4) specification,
 - and (5) review

Software Requirements Analysis

- The analyst must
 - define all externally observable data objects,
 - evaluate the flow and content of information,
 - define and elaborate all software functions,
 - understand software behavior in the context of events that affect the system, establish system interface characteristics,
 - and uncover additional design constraints.

Software Req. Analysis Principles

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behavior of the software (as a consequence of external events) must be represented.
4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis process should move from essential information toward implementation detail.

Software Req. Analysis Guidelines

- *Understand the problem before you begin to create the analysis model*
- *Develop prototypes that enable a user to understand how human/machine interaction will occur.*
- *Record the origin of and the reason for every requirement.*
- *Use multiple views of requirements.*
- *Rank requirements.*
- *Work to eliminate ambiguity.*

Software Models

- **Functional models.** Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem specific functions.
- **Behavioral models.** Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioral model. A computer program always exists in some state—an externally observable mode of behavior (e.g., waiting, computing, printing, polling) that is changed only when some event occurs.

Prototyping Approach

- The prototyping paradigm can be either close-ended or open-ended.
 - The close-ended approach is often called *throwaway prototyping*;
 - An open-ended approach, called *evolutionary prototyping*, uses the prototype as the first part of an analysis activity that will be continued into design and construction.

Prototyping Approach

- It is necessary to determine whether the system to be built is amenable to prototyping.
 - application area,
 - application complexity,
 - customer characteristics,
 - and project characteristics.

Prototyping Approach

- It is necessary to determine whether the system to be built is amenable to prototyping.
 - application area,
 - application complexity,
 - customer characteristics,
 - and project characteristics.

Question	Throwaway prototype	Evolutionary prototype	Additional preliminary work required
Is the application domain understood?	Yes	Yes	No
Can the problem be modeled?	Yes	Yes	No
Is the customer certain of basic system requirements?	Yes/No	Yes/No	No
Are requirements established and stable?	No	Yes	Yes
Are any requirements ambiguous?	Yes	No	Yes
Are there contradictions in the requirements?	Yes	No	Yes

Prototyping Approach

- Prototyping Methods and Tools
 - Fourth generation techniques
 - Reusable software components
 - Formal specification and prototyping environments

Software Specification

- Specification may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to successful software implementation.

Software Specification Principles

- Specification principles:
 - Separate functionality from implementation.
 - Develop a model of the desired behavior of a system that encompasses data & the functional responses of a system to various stimuli from the environment.
 - Establish the context in which software operates by specifying the manner in which other system components interact with software.
 - Define the environment in which the system operates

Software Specification Principles

- Specification principles (cont.):
 - Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.
 - Recognize that “the specifications must be tolerant of incompleteness and augmentable.” A specification is always a model of some real situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.
 - Establish the content and structure of a specification in a way that will enable it to be amenable to change.

Chapter III

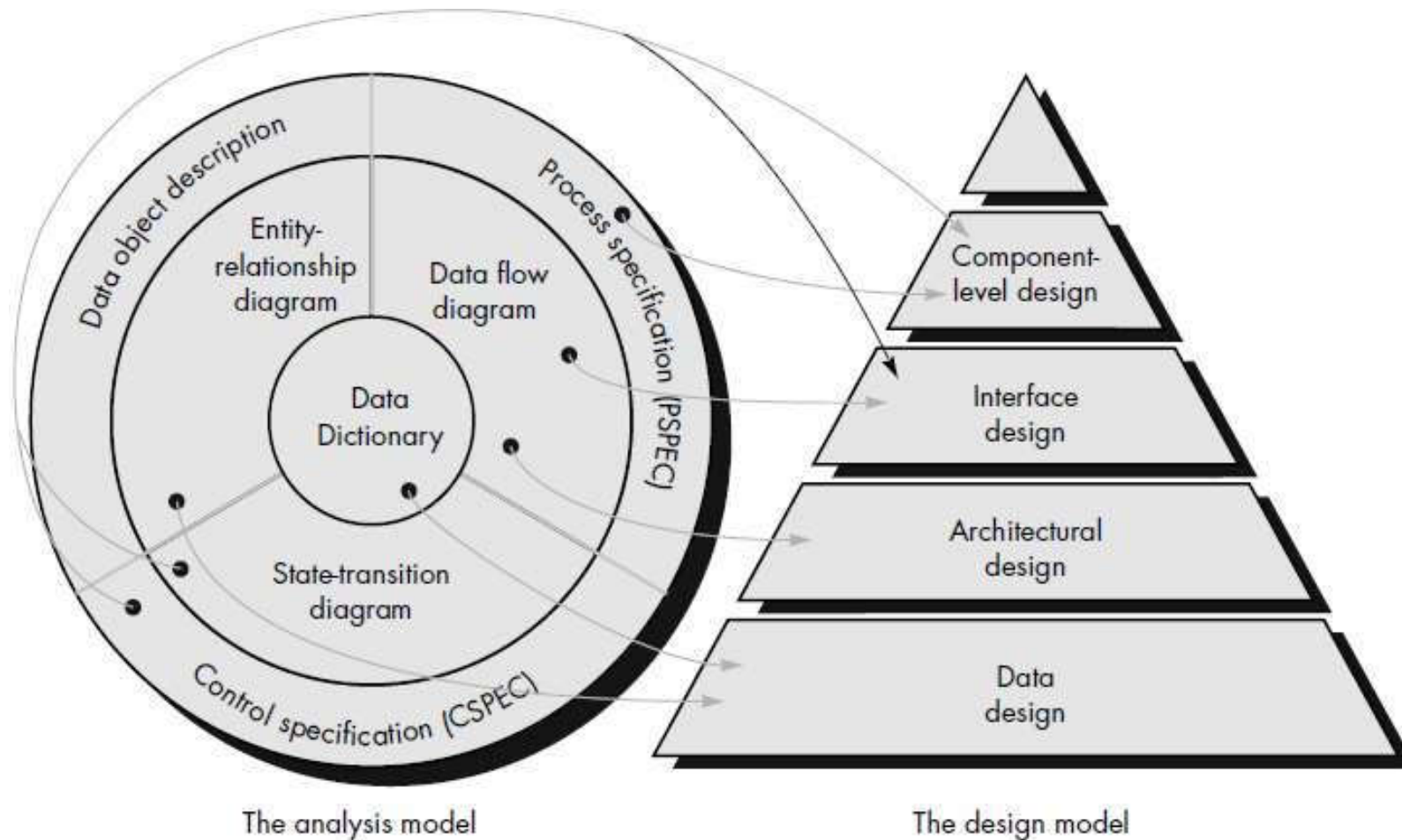
Design Concepts & Principles

- Design Principles
- Architectural Design
- Interface Design

Overview

- The designer's goal is to produce a model or representation of an entity that will later be built.
- Design is the first of three technical activities (design, code generation, and test) that are required to build and verify the software
- Software requirements, manifested by the data, functional, and behavioral models, feed the design task
- The design task produces a data design, an architectural design, an interface design, and a component design

Overview



Translating analysis model into a software design

Overview

- The design task produces a data design, an architectural design, an interface design, and a component design.
 - The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software.
 - The *architectural design* defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements.

Overview

- The *interface design* describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it.
- The *component-level design* transforms structural elements of the software architecture into a procedural description of software components.

Overview

- Design provides us with representations of software that can be assessed for quality.
- Design is the only way that we can accurately translate a customer's requirements into a finished software product.

Design Principles

- Software design is both a process and a model.
 - The design *process* is a sequence of steps that enable the designer to describe all aspects of the software to be built (creative skill, past experience, a sense of what makes “good” software)
 - The design *model* begins by representing the totality of the thing to be built and slowly refines the thing to provide guidance for constructing each detail

Design Principles

- The design process should not suffer from “tunnel vision.”
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.

Design Principles

- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Data Design

- Data Design

- *Data design* creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
- The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level.
- The data objects defined during software requirements analysis are modeled using entity/relationship diagrams.

Architectural Design

- What is Software Architecture?
 - The software architecture is the structure(s) of the system, which comprise *software components*, the externally visible properties of those components, and the relationships among them.
 - A *software component* can be a program module, databases and “middleware” that enable the configuration of a network of clients and servers

Interface Design

- Interface design focuses on three areas of concern:
 - (1) the design of interfaces between software components,
 - (2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities),
 - (3) the design of the interface between a human (i.e., the user) and the computer.

User Interface Design

- The three “golden rules”:
 1. Place the user in control.
 2. Reduce the user’s memory load.
 3. Make the interface consistent.

User Interface Design Golden Rules

1. Place the user in control.

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions (An interaction mode is the current state of the interface).
- Provide for flexible interaction
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user
- Design for direct interaction with objects that appear on the screen.

User Interface Design Golden Rules

2. Reduce the user's memory load.

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive
- The visual layout of the interface should be based on a real world metaphor
- Disclose information in a progressive

User Interface Design Golden Rules

3. Make the interface consistent.

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

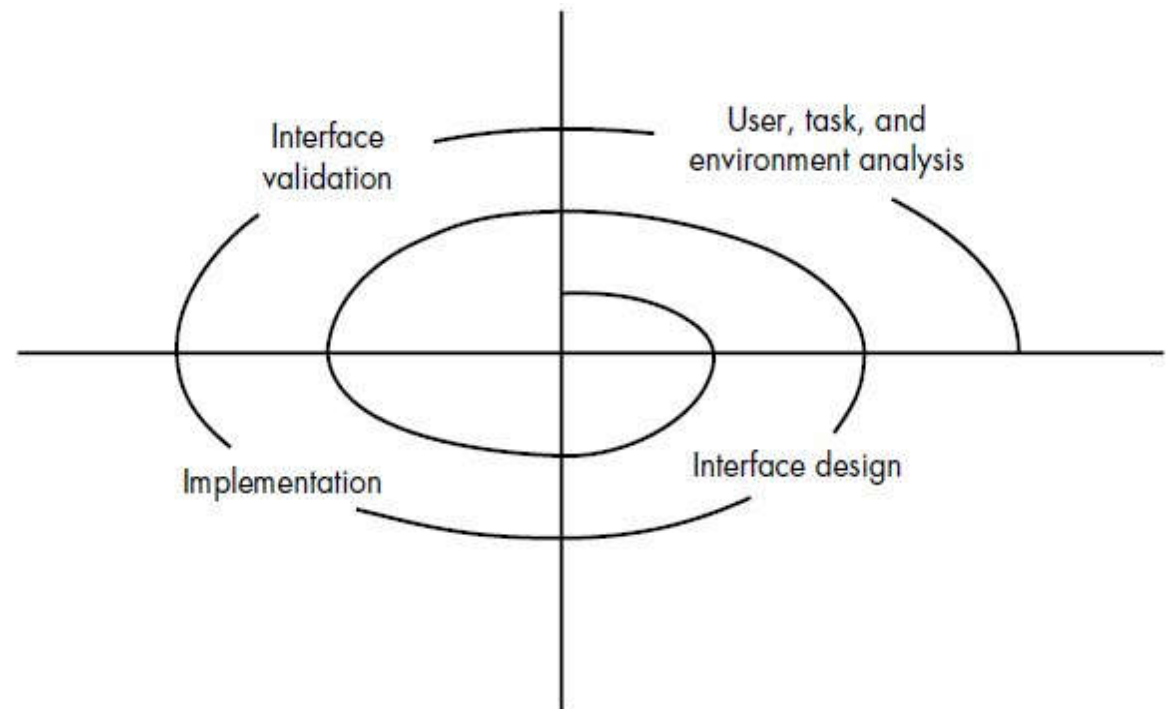
User Interface Design Models

- Interface Design Models

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

User Interface Design Process

- The User Interface Design Process
 - encompasses four distinct framework activities:
 - User, task, and environment analysis and modeling
 - Interface design
 - Interface construction
 - Interface validation



Chapter IV

Software Testing

- Software Testing Techniques
- Software Testing Strategies

Software Testing

● Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A successful test is one that uncovers an as-yet-undiscovered error.

Software Testing

- Attributes of a “good” test
 - A good test has a high probability of finding an error.
 - A good test is not redundant.
 - A good test should be “best of breed”.
 - A good test should be neither too simple nor too complex.

Software Testing Principles

● Testing Principles

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The *Pareto* principle applies to software testing.
- Testing should begin “in the small” and progress toward testing “in the large.”
- Exhaustive testing is not possible.
- To be most effective, testing should be conducted by an independent third party.

Software Testing Techniques

- Test Case Design Methods
 - *White-Box Testing*: Knowing the internal workings of a product, tests can be conducted to ensure that internal operations are performed according to specifications and all internal components have been adequately exercised.
 - *Black-Box Testing*: Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

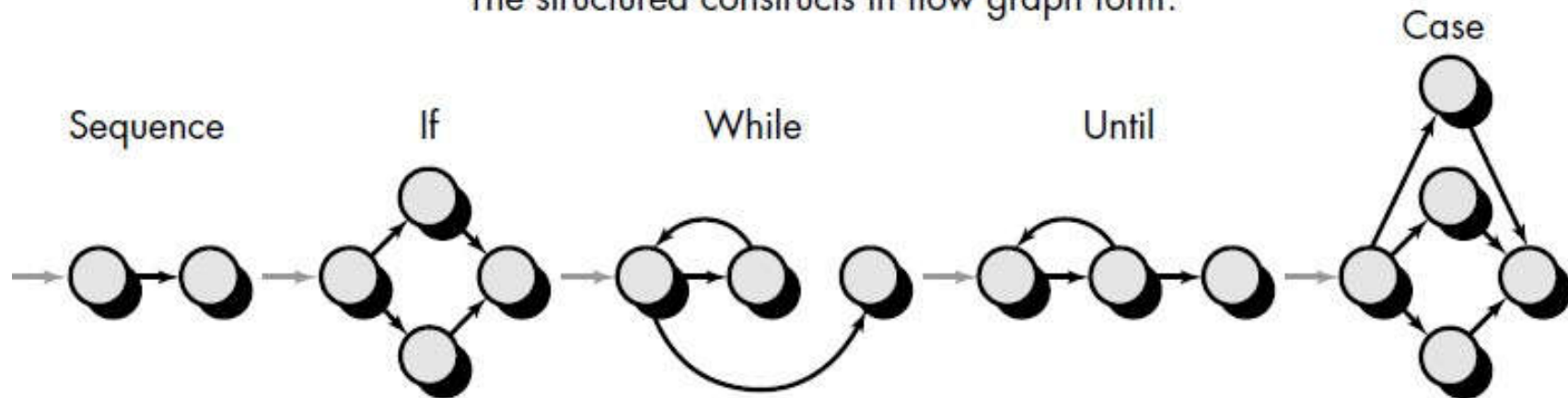
White-Box Testing

- White-box testing (*glass-box testing*) uses the control structure of the procedural design to derive test cases, in such a way that
 - guarantee that all independent paths within a module have been exercised at least once,
 - exercise all logical decisions on their true and false sides,
 - execute all loops at their boundaries and within their operational bounds,
 - exercise internal data structures to ensure their validity.

White-Box Testing Techniques

- *Basis Path Testing* (Tom McCabe)
 - enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

The structured constructs in flow graph form:

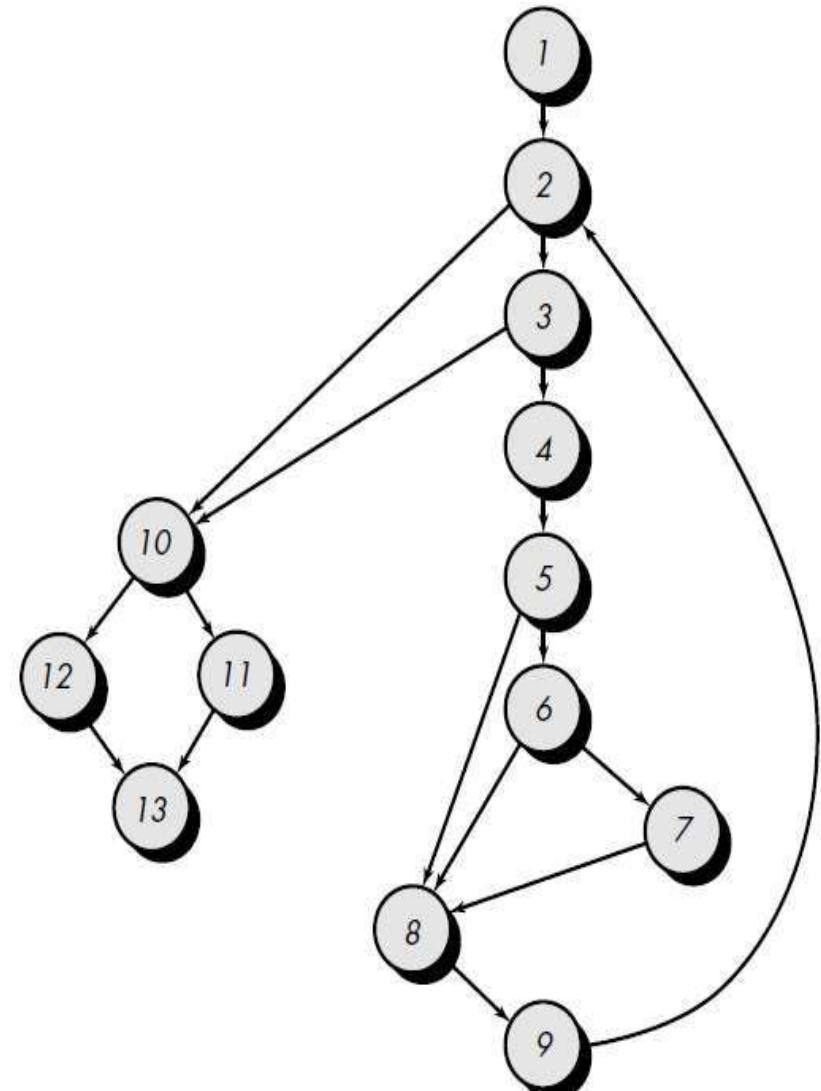
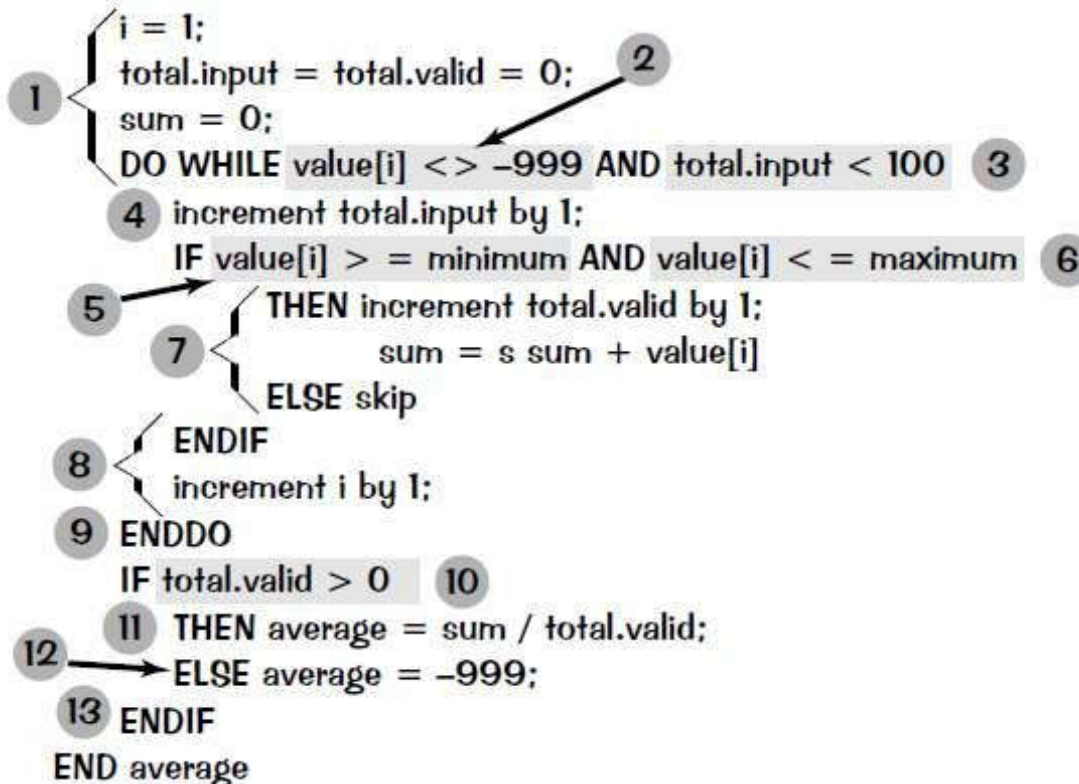


White-Box Testing Techniques

● Basis Path Testing (cont.)

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;



White-Box Testing Techniques

● Basis Path Testing (cont.)

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

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

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

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

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

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

Path 1 test case:

value(k) = valid input, where $k < i$ for $2 \leq i \leq 100$

value(i) = -999 where $2 \leq i \leq 100$

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

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

Path 2 test case:

value(1) = -999

Expected results: Average = -999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

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

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

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

Path 5 test case:

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

value(k) > maximum where $k \leq i$

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

Path 6 test case:

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

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

Black-Box Testing

- *Black-box testing (behavioral testing)* focuses on the functional requirements of the software, enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
 - incorrect or missing functions,
 - interface errors,
 - errors in data structures or external database access,
 - behavior or performance errors,
 - initialization and termination errors.

Black-Box Testing

- Graph-Based Testing Methods

- the first step is to understand the objects that are modeled in software and the relationships that connect these objects.
- next step is to define a series of tests that verify “all objects have the expected relationship to one another”

⇒(1) creating a *graph* of important objects and their relationships

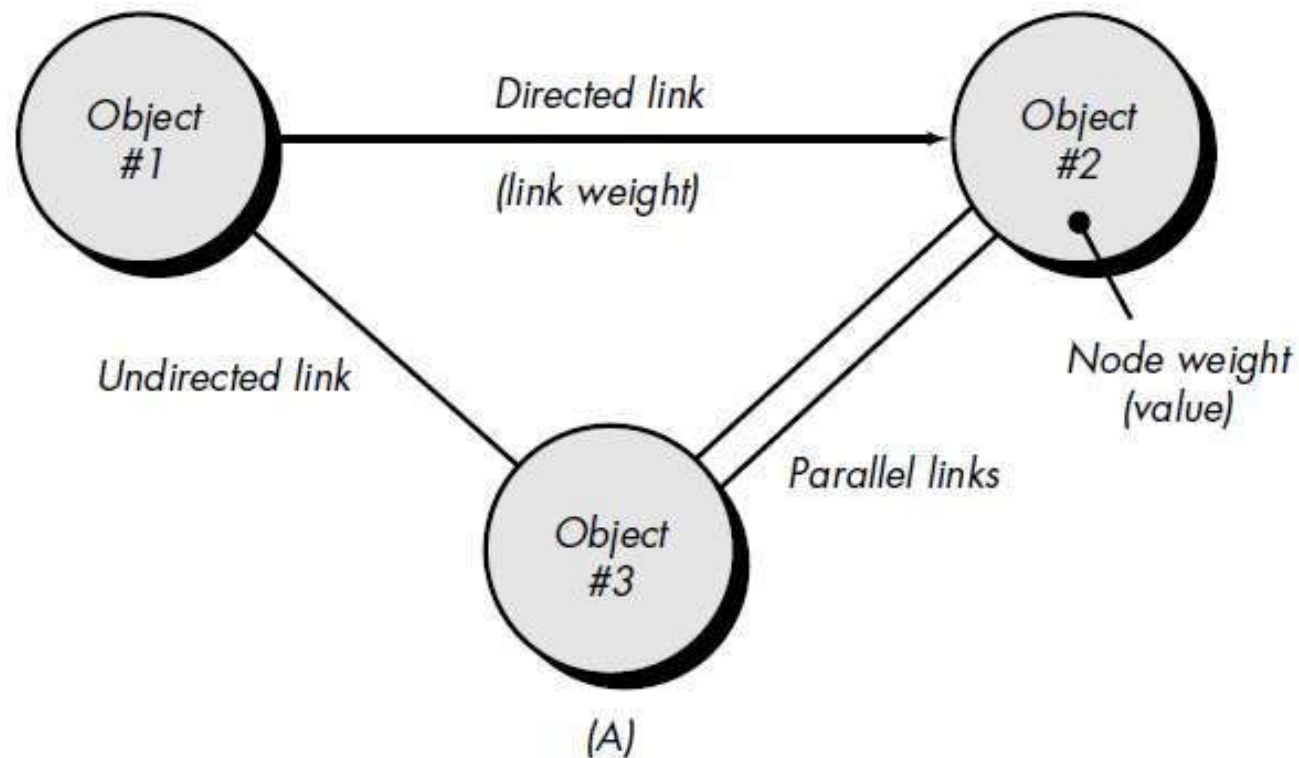
⇒(2) devising a series of *tests* that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Black-Box Testing

- Graph-Based Testing Methods (*cont.*)
 - A *graph* is a collection of
 - *nodes* that represent objects;
 - *links* that represent the relationships between objects;
 - *node weights* that describe the properties of a node (e.g., a specific data value or state behavior);
 - *link weights* that describe some characteristic of a link.

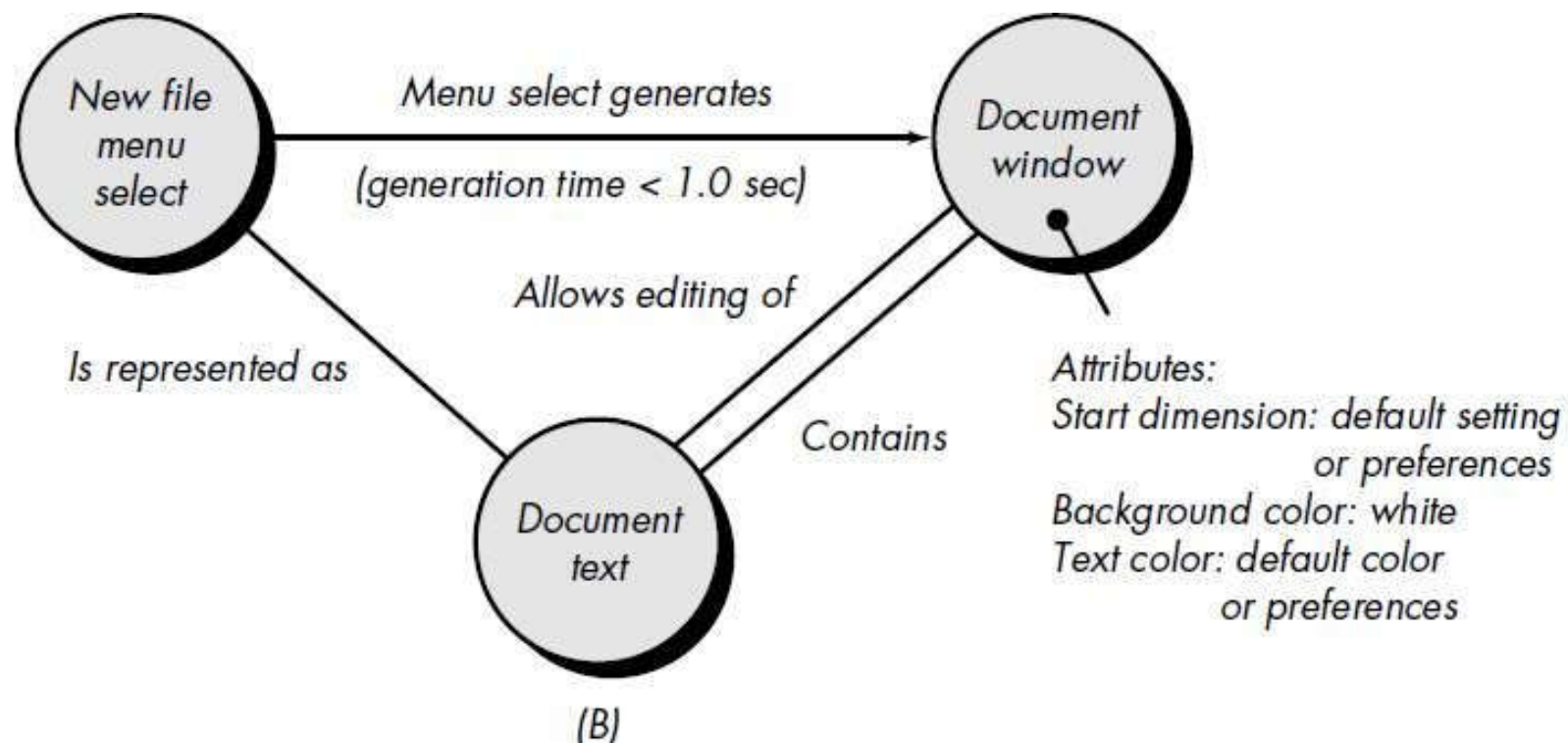
Black-Box Testing

- Graph-Based Testing Methods



Black-Box Testing

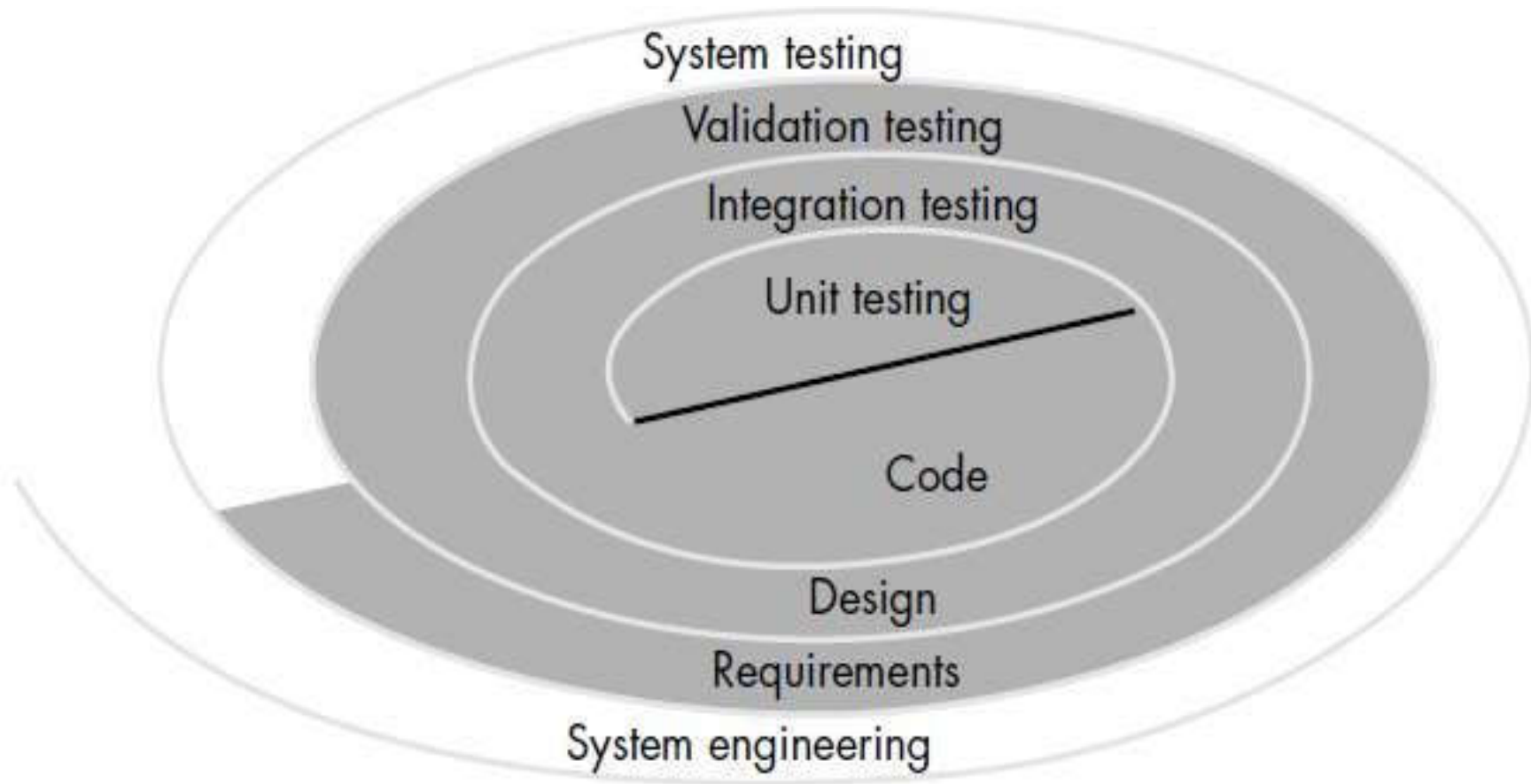
● Graph-Based Testing Methods



Software Testing Strategies

- The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.
- Testing strategy must incorporate
 - test planning,
 - test case design,
 - test execution,
 - resultant data collection and evaluation.

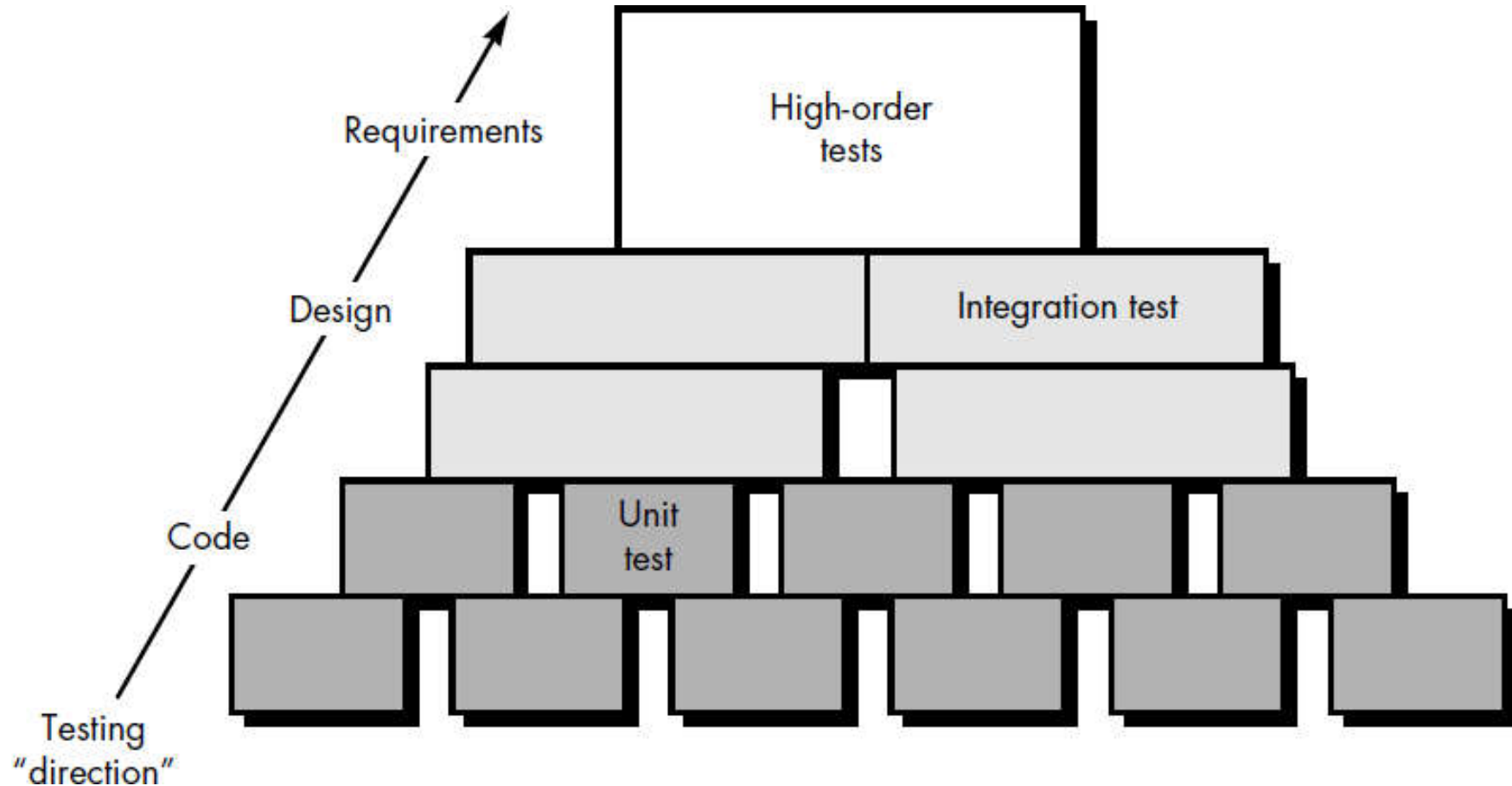
Software Testing Strategies



Software Testing Strategies

- *Unit testing* concentrates on each unit (i.e., component) of the software as implemented in source code.
- *Integration testing* focuses on design and the construction of the software architecture.
- *Validation testing*, where requirements established as part of software requirements analysis are validated against the software that has been constructed.
- *System testing*, where the software and other system elements are tested as a whole.

Software Testing Strategies



Software Testing Strategies

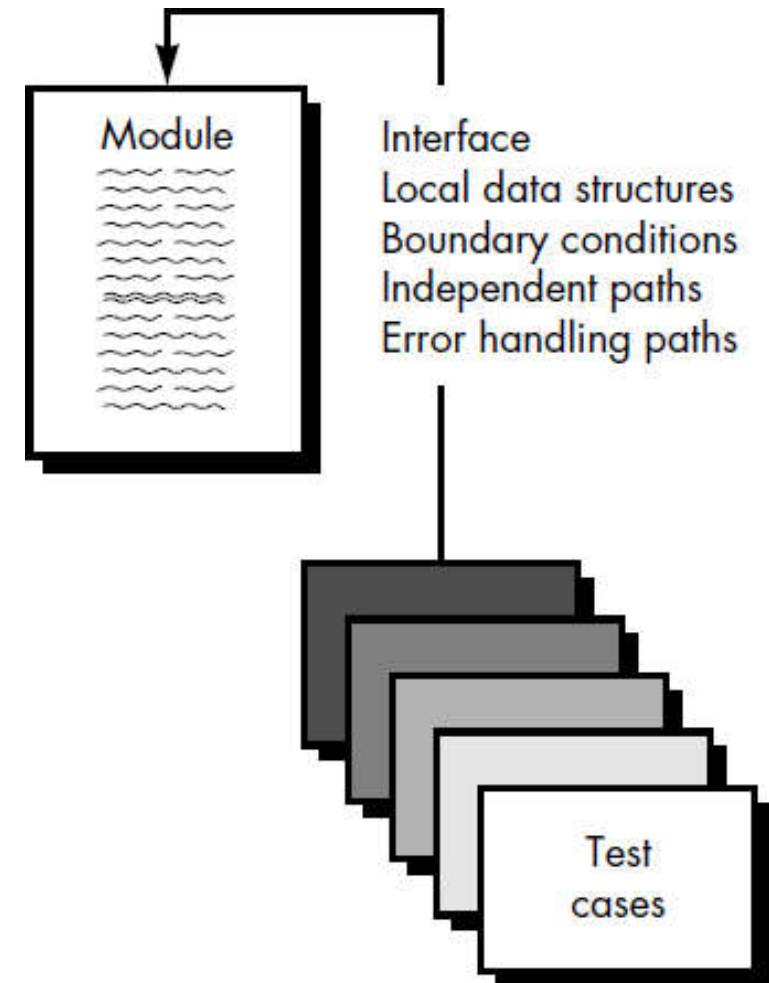
● Unit Testing

- Unit testing focuses on the smallest unit of software design, i.e. the software component or module.
- The unit test is white-box oriented, and the step can be conducted in parallel for multiple components

Software Testing Strategies

● Unit Testing

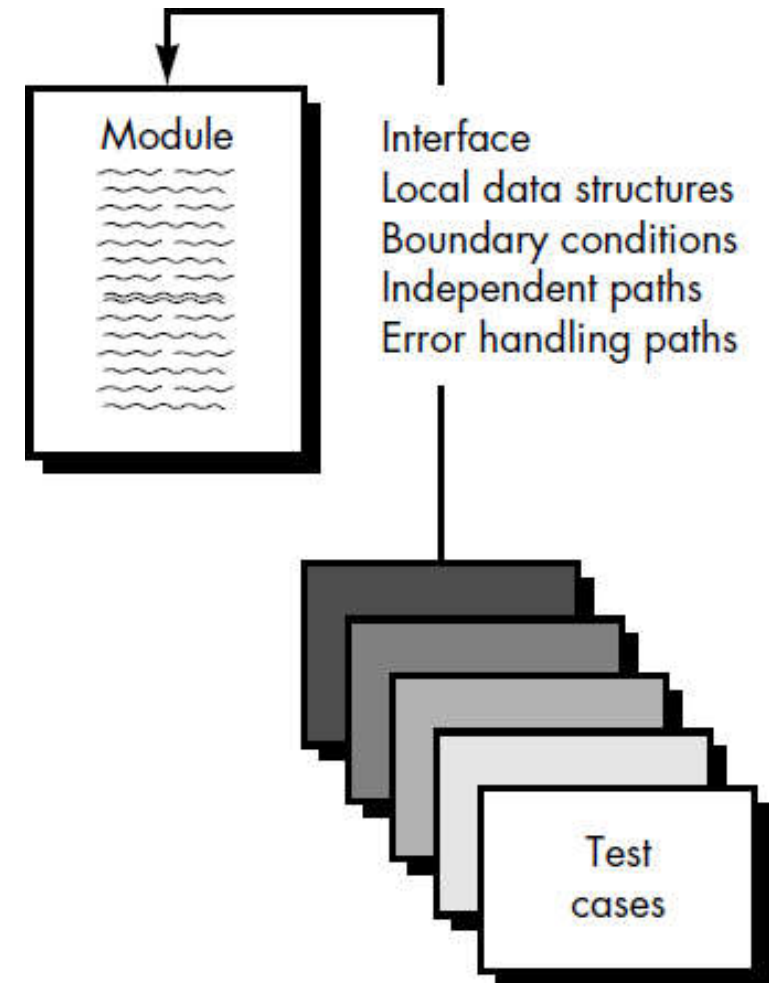
- The *module interface* is tested to ensure that information properly flows into and out of the program unit under test.
- The *local data structure* is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.



Software Testing Strategies

● Unit Testing

- *Boundary conditions* are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- All *independent paths* (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Finally, all *error handling* paths are tested.



Software Testing Strategies

● Integration Testing

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

Software Testing Strategies

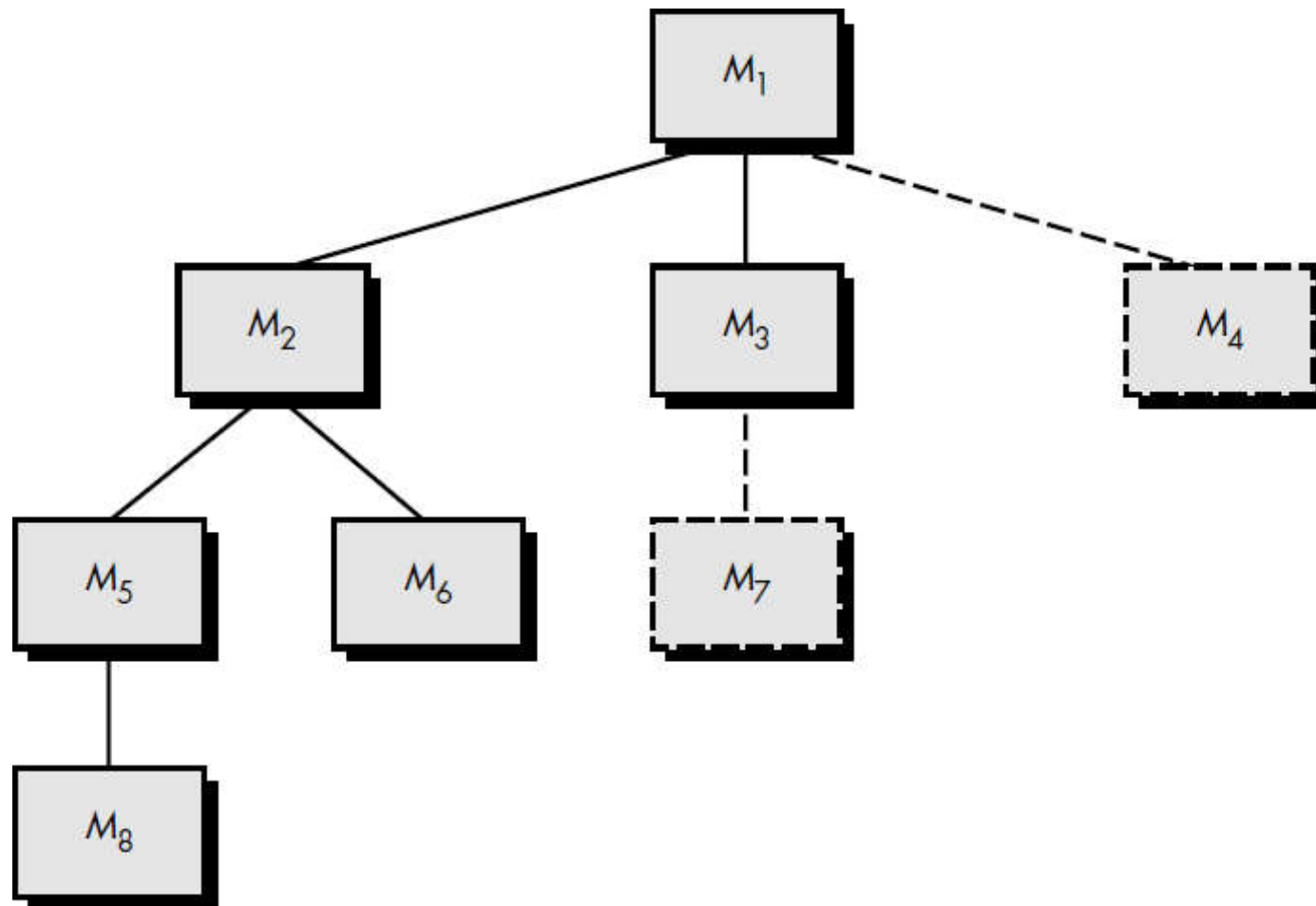
● Integration Testing

- *Top-down integration testing* is an incremental approach to construction of program structure.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Software Testing Strategies

- Integration Testing

- Top-down Integration: *Depth-first integration*



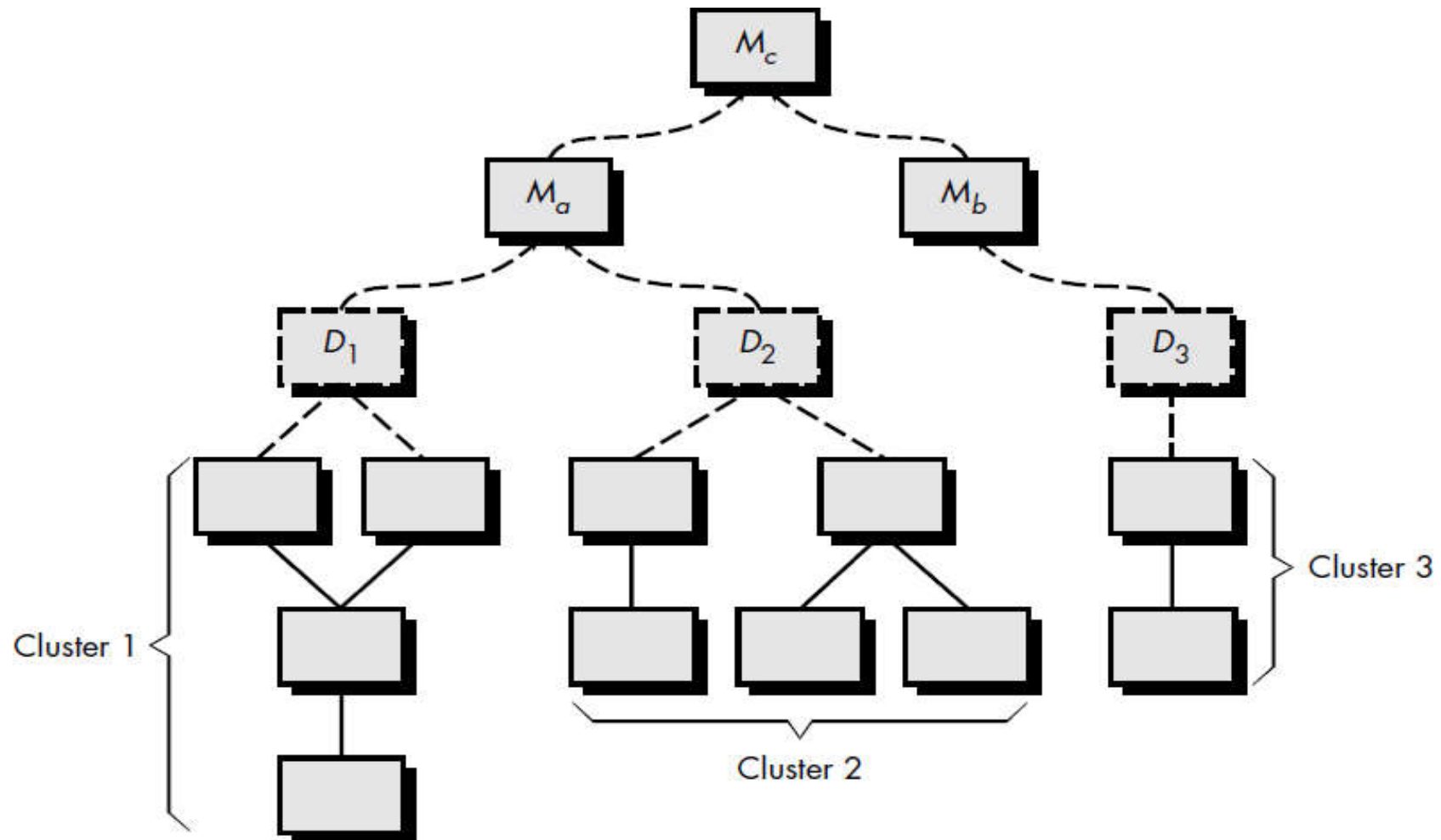
Software Testing Strategies

● Integration Testing

- *Bottom-up Integration Testing* begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure).
- Components are integrated from the bottom up, therefore processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

Software Testing Strategies

- Integration Testing
 - Bottom-up Integration



Software Testing Strategies

- **Validation Testing**

- Validation succeeds when software functions in a manner that can be reasonably expected by the customer

Software Testing Strategies

● Validation Testing

- *Alpha Testing* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.
- *Beta Testing* is conducted at one or more customer sites by the end-user of the software. The developer is generally not present. The beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems that are encountered during beta testing and reports these to the developer at regular intervals.

Software Testing Strategies

● System Testing

- *Recovery testing* forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process.

Debugging Process

