Jaipur Engineering College & Research Centre, Jaipur

Notes
Software Engineering
[3CS4 - 07]

Prepared By:
Manju Vyas
Abhishek Jain
Geerija Lavania

VISION AND MISSION OF INSTITUTE

VISION

To become renowned centre of outcome based learning and work towards academic, professional, cultural and social enrichments of the lives of individual and communities"

MISSION

M1. Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2. Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.

M3. Offer opportunities for interaction between academia and industry.

M4. Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION AND MISSION OF DEPARTMENT

VISION

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISSION

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities.

COURSE OUTCOMES

CO1) understand the purpose of designing a system and evaluate the various models suitable as per its requirement analysis

CO2) understand and apply software project management, effort estimation and project scheduling.

CO3) formulate requirement analysis, process behaviour and software designing.

CO4) Implement the concept of object oriented analysis modelling with the reference of UML and advance SE tools

Program Outcomes (PO)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problemsand design system components or processes that meet thespecified needs with appropriate consideration for the public health and safety, andthe cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issuesand the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEO)

1.To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenge

2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.

4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career.

5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge.

MAPPING CO-PO

Cos/POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	3	2	1	2	1	1	2	3
CO2	3	3	3	3	2	2	1	2	2	2	3	3
CO3	3	3	3	2	2	2	1	2	1	2	2	3
CO4	3	3	3	3	3	1	0	1	1	2	2	3

PSO

PSO1: Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2: Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

UNIT -4 SOFTWARE DESIGN

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the past four decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Each helps the software engineer to answer the following questions:

• What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

Abstraction

Each step in the software process is a refinement in the level of abstraction of the

software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

A *data abstraction* is a named collection of data that describes a data object . In the context of the procedural abstraction *open*, we can define a data abstraction

called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Control abstraction is the third form of abstraction used in software design. Like

procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* used to coordinate activities in an operating system.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach. Refinement is actually a process of *elaboration.*We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture (described in Section 13.4.4) embodies modularity; that is, software is divided into separately named and addressable components, often called *modules,* that are integrated to satisfy problem requirements. Another important question arises when modularity is considered. How do wedefine an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

Modular decomposability. If a design method provides a systematic

mechanism for decomposing the problem into subproblems, it will reduce

the complexity of the overall problem, thereby achieving an effective modular

solution.

Modular composability. If a design method enables existing (reusable)

design components to be assembled into a new system, it will yield a modular

solution that does not reinvent the wheel.

Modular understandability. If a module can be understood as a standalone

unit (without reference to other modules), it will be easier to build and

easier to change.

Modular continuity. If small changes to the system requirements result in

changes to individual modules, rather than systemwide changes, the impact

of change-induced side effects will be minimized.

Modular protection. If an aberrant condition occurs within a module and

its effects are constrained within that module, the impact of error-induced

side effects will be minimized.

Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, *components* can be generalized to represent major system elements and their interactions. One goal of software design is to derive an architectural rendering of a system.This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

Control Hierarchy

Control hierarchy, also called *program structure,* represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller .

For example, referring to Figure module *M* is superordinate to modules *a, b,* and *c*. Module *h* is subordinate to module *e* and is ultimately subordinate to module *M*. Width-oriented relationships (e.g., between modules *d* and *e*) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the

software architecture: visibility and connectivity. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module.

Connectivity indicates the set of components that are directly invoked or used as

data by a given component. For example, a module that directly causes another module to begin execution is connected to it

Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 13.4a, *horizontal partitioning* defines separate branches of the modular hierarchy for each major program function. *Control modules,* represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called *processing*) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- software that is easier to test
- software that is easier to maintain
- propagation of fewer side effects
- software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the

representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

Information Hiding

The concept of modularity leads every software designer to a fundamental question:

"How do we decompose a software solution to obtain the best set of modules?"

The principle of *information hiding* suggests that modules be

"characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module

EFFECTIVE MODULAR DESIGN

All the fundamental design concepts described in the preceding section serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

Functional Independence

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules

are possible.To summarize, functional independence is a key to good design, and design is the key to software quality.

Cohesion

Cohesion is a natural extension of the information hiding concept described earlier. A cohesive module performs a single task within a software procedure,

requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module.

Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed. At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed *coincidentally cohesive.* A module that performs tasks that are related logically (e.g.a module that produces all output regardless of type) is *logically cohesive.* When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion.*

Coupling

Coupling is a measure of interconnection among modules in a software structure.

Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity

among modules results in software that is easier to understand and less prone to a "ripple effect" , caused when errors occur at one location and propagate

through a system.

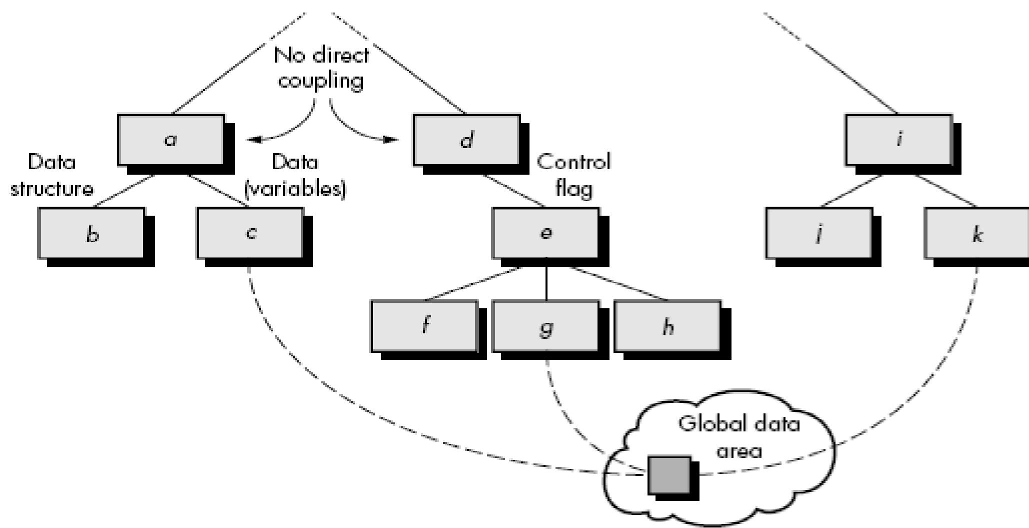


Figure provides examples of different types of module coupling. Modules *a*

and *d* are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module *c* is subordinate to module *a* and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called *data coupling*) is exhibited in this portion of structure. A variation of data coupling, called *stamp coupling,* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

“ The designer's goal is to produce a model or representation of an entity that will later be built “

Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer’s requirements and at the same time assessed for quality against a set of predefined criteria for “good” design. In the

software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.

Software engineers design computerbased systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface

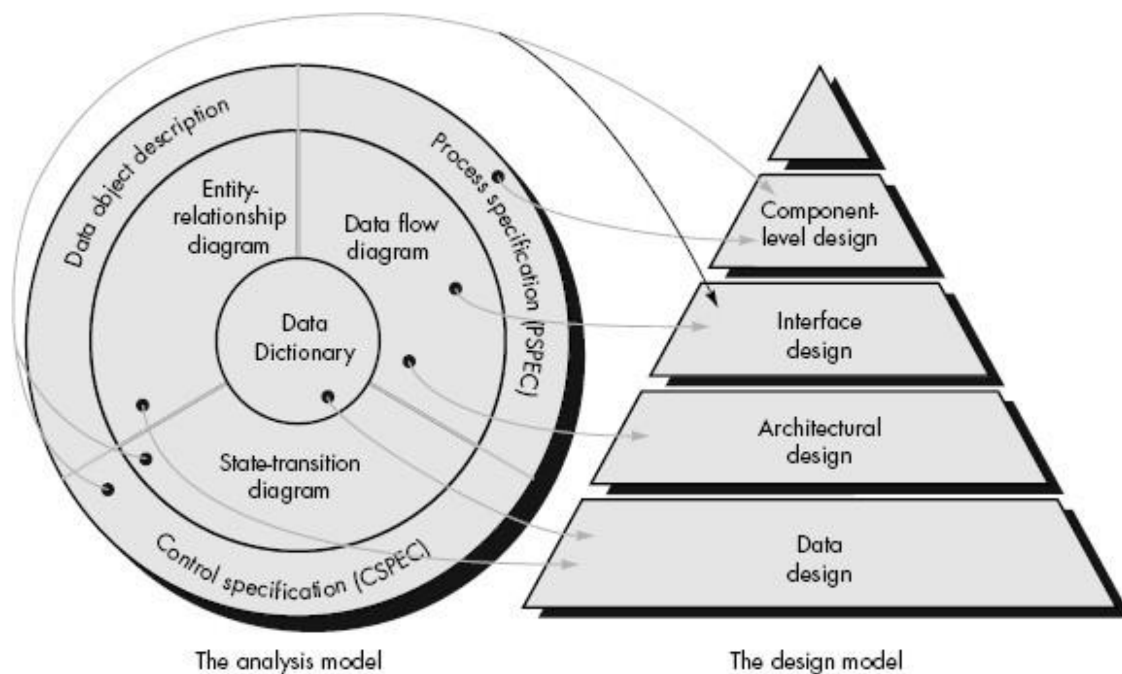
level, human ergonomics often dictate our design approach. At the component level, a “programming approach” leads us to effective data and procedural designs.

SOFTWARE DESIGN AND SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture.More detailed data design occurs as each software component is designed.

The *architectural design* defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied [SHA96]. The architectural design representation the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.



THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Design and Software Quality

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in earler chapters.

McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

• The design must implement all of the explicit requirements contained in the

analysis model, and it must accommodate all of the implicit requirements

desired by the customer.

• The design must be a readable, understandable guide for those who generate

code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing

the data, functional, and behavioral domains from an implementation

perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this chapter, we discuss design quality criteria in some detail. For the time being, we present the following guidelines:

A design should exhibit an architectural structure that

(1) has been created using recognizable design patterns

(2) is composed of components that exhibit good design characteristics

(3) can be implemented in an evolutionary fashion, thereby facilitating

implementation and testing.

A design should be modular; that is, the software should be logically partitioned

into elements that perform specific functions and subfunctions.

A design should contain distinct representations of data, architecture, interfaces,

and components (modules).

A design should lead to data structures that are appropriate for the objects to

be implemented and are drawn from recognizable data patterns.

A design should lead to components that exhibit independent functional

characteristics.

A design should lead to interfaces that reduce the complexity of connections

between modules and with the external environment.

A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

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. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design. The design *model* is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software

provides a variety of different views of the computer software.

Basic design principles enable the software engineer to navigate the design process. Davis suggests a set of principles for software design, which have been

adapted and extended in the following list:

• **The design process should not suffer from "tunnel vision."** A good

designer should consider alternative approaches, judging each based on the

requirements of the problem, the resources available to do the job, and the

design concepts .

• **The design should be traceable to the analysis model.** Because a single

element of the design model often traces to multiple requirements, it is necessary

to have a means for tracking how requirements have been satisfied by

the design model.

• **The design should not reinvent the wheel.** Systems are constructed using

a set of design patterns, many of which have likely been encountered before.

These patterns should always be chosen as an alternative to reinvention.

Time is short and resources are limited! Design time should be invested in

representing truly new ideas and integrating those patterns that already exist.

• **The design should "minimize the intellectual distance"**

between the software and the problem as it exists in the real world.

That is, the structure of the software design should (whenever possible)

mimic the structure of the problem domain.

• **The design should exhibit uniformity and integration.** A design is uniform

if it appears that one person developed the entire thing. Rules of style

and format should be defined for a design team before design work begins. A

design is integrated if care is taken in defining interfaces between design

components.

• **The design should be structured to accommodate change.** The design

concepts discussed in the next section enable a design to achieve this principle.

• **The design should be structured to degrade gently, even when aberrant**

data, events, or operating conditions are encountered. Welldesigned

software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

• **Design is not coding, coding is not design.** Even when detailed procedural

designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

• **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.

• **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors [MEY88]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed,reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.