Course : Software Engineering and Project Management

Unit 3

Design Engineering

Syllabus

Design Process & quality, Design Concepts, The design Model, Pattern-based Software Design.

Architectural Design : Design Decisions, Views, Patterns, Application Architectures

Modeling Component level Design: component, Designing class based components, conducting component-level design

User Interface Design : The golden rules, Interface Design steps & Analysis, Design Evaluation

Case Study: WebApp Interface Design

Design Process & quality

- ✓ Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.

What is design ?

- ✓ Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system.

- ✓ Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Design Process

- ✓ Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Software Quality Guidelines and Attributes

- ✓ Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews suggests three characteristics that serve as a guide for the evaluation of a good design:

1. The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
2. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

3. 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.

Quality Guidelines

✓ In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design.

✓ **Consider the following guidelines :-**

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics, and (3) can be implemented in an evolutionary fashion, 2 thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

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

8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

- ✓ Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**.

- ✓ The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.

- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

- ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

- ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability —and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

Design Process & quality

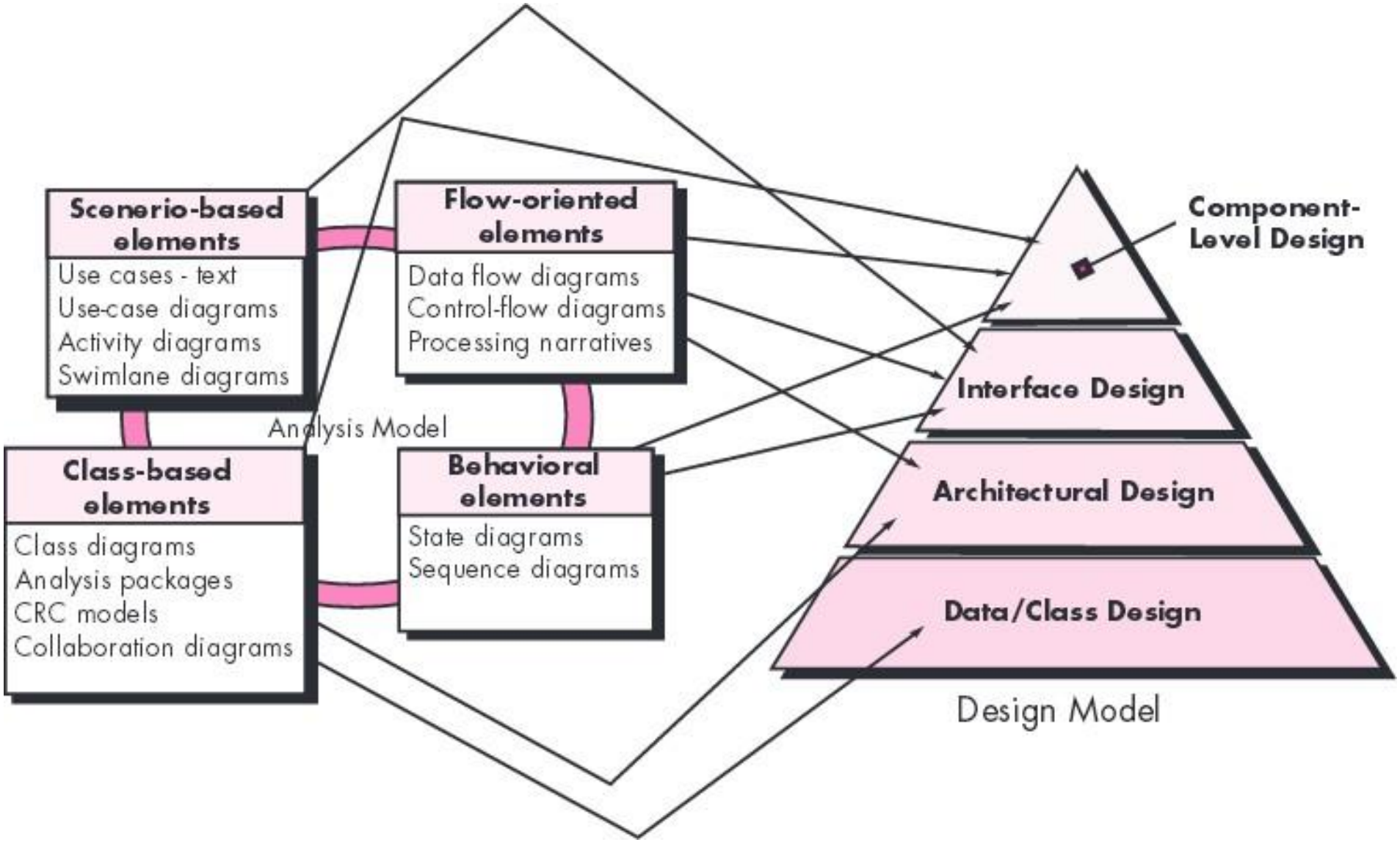

Figure 8.1 : Translating the requirements model into the design mode

Design Concepts

1. Abstraction

- ✓ When you consider a modular solution to any problem, many levels of abstraction can be posed.
- ✓ At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- ✓ At lower levels of abstraction, a more detailed description of the solution is provided.
- ✓ Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

- ✓ **A procedural abstraction** refers to a sequence of instructions that have a specific and limited function.
- ✓ The name of a procedural abstraction implies these functions, but specific details are suppressed.

- ✓ An example of a procedural abstraction would be the word ***open*** for a door.
- ✓ ***Open*** implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

- ✓ **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***.

2. Architecture

- ✓ Software architecture alludes (signals) 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 structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

- ✓ A set of architectural patterns enables a software engineer to solve common design problems.

- ✓ There is a set of properties that should be specified as part of an architectural design.

❖ **Structural properties :** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.

✓ For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

❖ **Extra-functional properties :** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

- **Families of related systems :** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

- ***Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.***

- Structural models represent architecture as an organized collection of program components.

- Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

- Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

- Process models focus on the design of the business or technical process that the system must accommodate.

- Functional models can be used to represent the functional hierarchy of a system.

- ✓ A number of different architectural description languages (ADLs) have been developed to represent these models.

3. Patterns

- ✓ " A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns".

- ✓ A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

- ✓ The intent of each design pattern is to provide a description that enables a designer to determine

  (1) whether the pattern is applicable to the current work,

  (2) whether the pattern can be reused (hence, saving design time),

  (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Separation of Concerns

- ✓ Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- ✓ A concern is a feature or behavior that is specified as part of the requirements model for the software.
- ✓ By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- ✓ For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2 , it follows that the effort required to solve p1 is greater than the effort required to solve p2.

- ✓ As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

- ✓ It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately.

- ✓ This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

5. Modularity

- ✓ Modularity is the most common manifestation of separation of concerns.
- ✓ Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
- ✓ Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
- ✓ The number of control paths, span of reference, number of variables, and over-all complexity would make understanding close to impossible.
- ✓ In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

- ✓ Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.
- ✓ Given the same set of requirements, more modules means smaller individual size.
- ✓ However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.
- ✓ These characteristics lead to a total cost or effort curve shown in the figure.
- ✓ There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

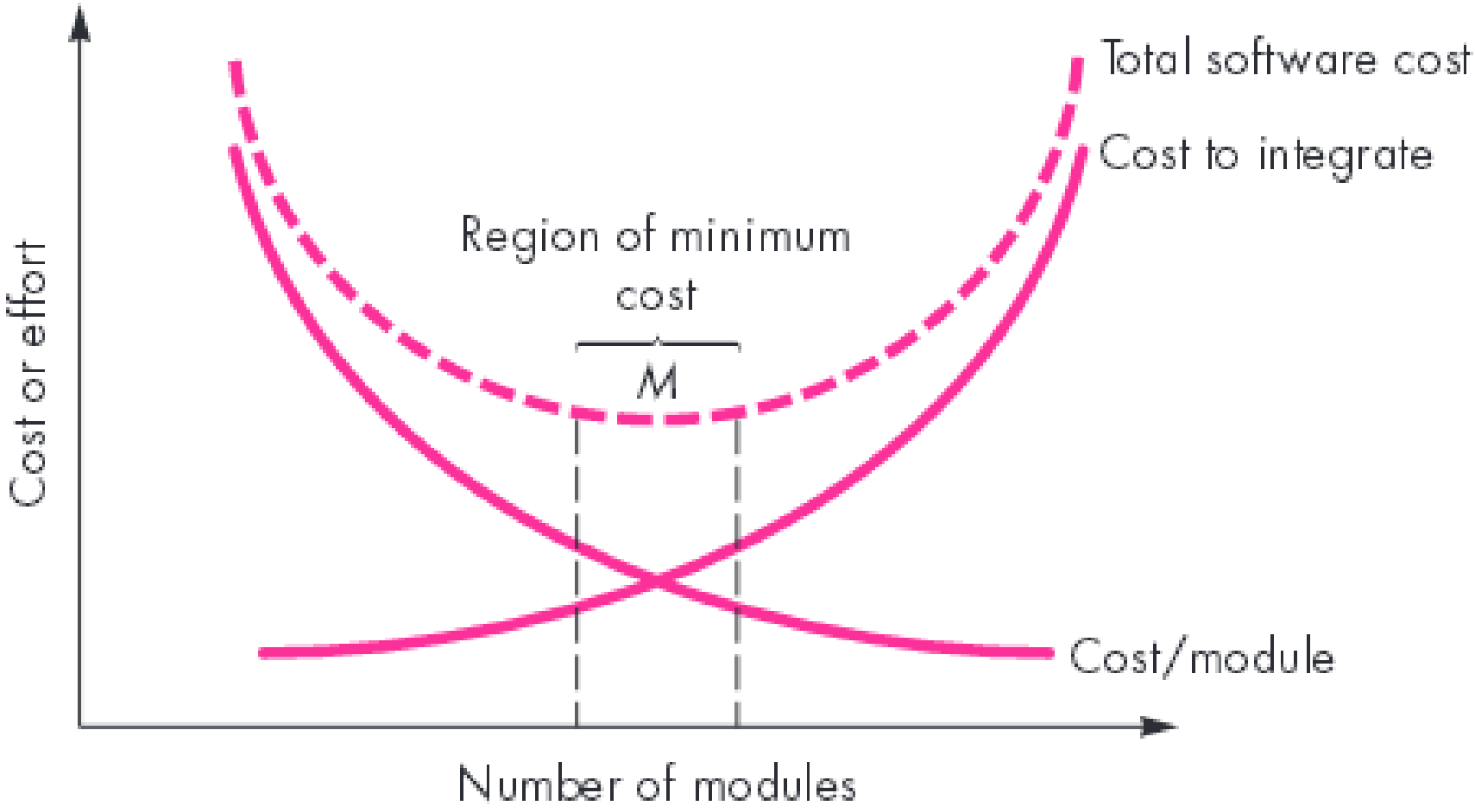

Figure 8.2 : Modularity and software cost

- ✓ You modularize a design (and the resulting program) so that development can be more easily planned;
- ✓ Software increments can be defined and delivered;
- ✓ Changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and
- ✓ Long-term maintenance can be conducted without serious side effects.

6. Information Hiding

- ✓ The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others. "
- ✓ Modules should be specified and designed so that information (algorithms 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.

- ✓ The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing & later during software maintenance.

- ✓ Because most data & procedural detail are hidden from other parts of the software, errors introduced during modification are less likely to prepare to other locations within the software.

7. Functional Independence

- ✓ Functional independence is achieved by developing modules.
- ✓ You should design software so that each module addresses a specific subset of requirements & has a simple interface when viewed from other parts of the program structure.

- ✓ Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized & interfaces are simplified.
- ✓ Independent modules are easier to maintain (& test) because secondary effects caused by design or code modifications are limited, error propagation is reduced, & reusable modules are possible.

- ✓ Independence is assessed using two qualitative criteria : **cohesion & coupling**.

- ***Cohesion*** is an indication of the relative functional strength of a module.

- ***Coupling*** is an indication of the relative interdependence among modules.

- Cohesion is a natural extension of the information-hiding.
- Cohesive module performs a single task, requiring little interaction with other components in other parts of a program.
- Cohesive module should do just one thing.

- Coupling is an indication 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, & what data pass across the interface.

- ✓ In software design, you should strive for the lowest possible coupling.

- ✓ Simple connectivity among modules results in software that is easier to understand & less prone to a "ripple effect", caused when errors occur at one location & propagate throughout a system.

8. Refinement

- ✓ Stepwise refinement is a top-down design strategy.
- ✓ A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached.
- ✓ Refinement is actually a process of **Elaboration**.
- ✓ You begin with a statement of function that is defined at a high level of abstraction.
- ✓ That is, the statement describes function or information conceptually but provides no information.
- ✓ You then elaborate on the original statement, providing more & more detail as each successive refinement (elaboration) occurs.

9. Aspects

- ✓ It is important to identify a aspects so that the design can properly accommodate them as refinement & modularization occur.

10. Refactoring

- ✓ Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- ✓ Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code (design) yet improves its internal structure.

- ✓ When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design iteration might yield a component that exhibits low cohesion.

- ✓ After careful consideration, you may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion.
- ✓ The result will be software that is easier to integrate, easier to test, & easier to maintain.

11. Object Oriented Design Concepts

- ✓ The OO paradigm is widely used in modern software engineering.
- ✓ OO concepts such as Classes & objects, inheritance, message & polymorphism, among others.

12. Design Classes

- ✓ The requirement model defines a set of analysis classes.
- ✓ Each describes some element of the problem domain, focusing on aspects of the problem that are user visible.
- ✓ The level of abstraction of an analysis class is relatively high.
- ✓ As the design model evolves, you will define a set of ***design classes*** that refine the analysis classes by providing design detail that will enable classes to be implemented.

- ✓ 5 different types of design classes, each representing a different layer of the design architecture, can be developed :-

1. **User Interface Classes :-** Define all abstractions that are necessary for Human Computer Interaction (HCI).
2. **Business Domain Classes :-** Are often refinements of the analysis classes defined earlier. The classes identify the attributes & services (methods) that are required to implement some element of the business domain.

3. **Process Classes :-** Implement lower-level business abstractions required to fully manage the business domain classes.

4. **Persistent Classes :-** Represent data stores (e.g. a database) that will persist beyond the execution of the software.

5. **System Classes :-** Implement software management & control functions that enable the system to operate & communicate within its computing environment & with the outside world.

✓ 4 Characteristics of a well-formed design class :-

1. **Complete & sufficient :-** A design class should be complete encapsulation of all attributes & methods that can reasonably be expected to exist for a class.

- For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes & methods that can reasonably be associated with the creation of a video scene.

- Sufficiently ensures that the design class contains only those methods that are sufficiently to achieve the intent of the class, no more & no less.

2. **Primitiveness :-** Methods associated with a design class should be focused on accomplishing one service for the class.

- Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
- For example, the class **VideoClip** for video-editing software might have attributes **start-point** and **end-point** to indicate the start & end points of the clip.
- The methods, **setStartPoint()** & **setEndPoint(),** provide the only means for establishing start & end points for the clip.

3. **High Cohesion :-** A cohesive design class has a small, focused set of responsibilities & single-mindedly applies attributes & methods to implement those responsibilities.

- For example, the class **VideoClip** might contain a set of methods for editing the video clip.

4. **Low Coupling :-** It is necessary for design classes to collaborate with one another.

- However, collaboration should be kept to an acceptable minimum.
- If a design model is highly coupled, the system is difficult to implement, to test, & to maintain over time.
- In general, design classes within a subsystem should have only limited knowledge of other classes.

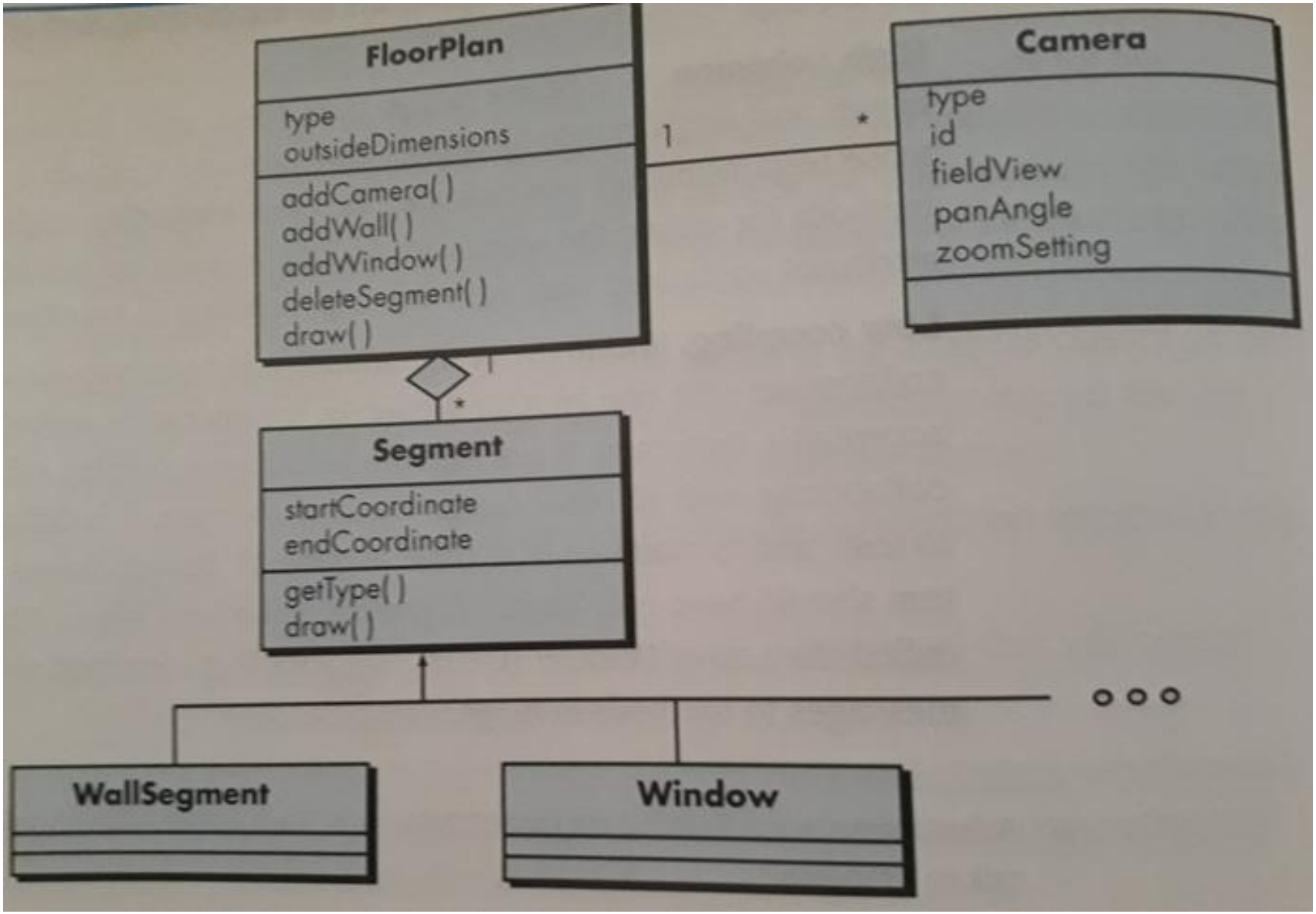

Figure 8.3 : Design class for FloorPlan & composite aggregation for the class

The Design Model

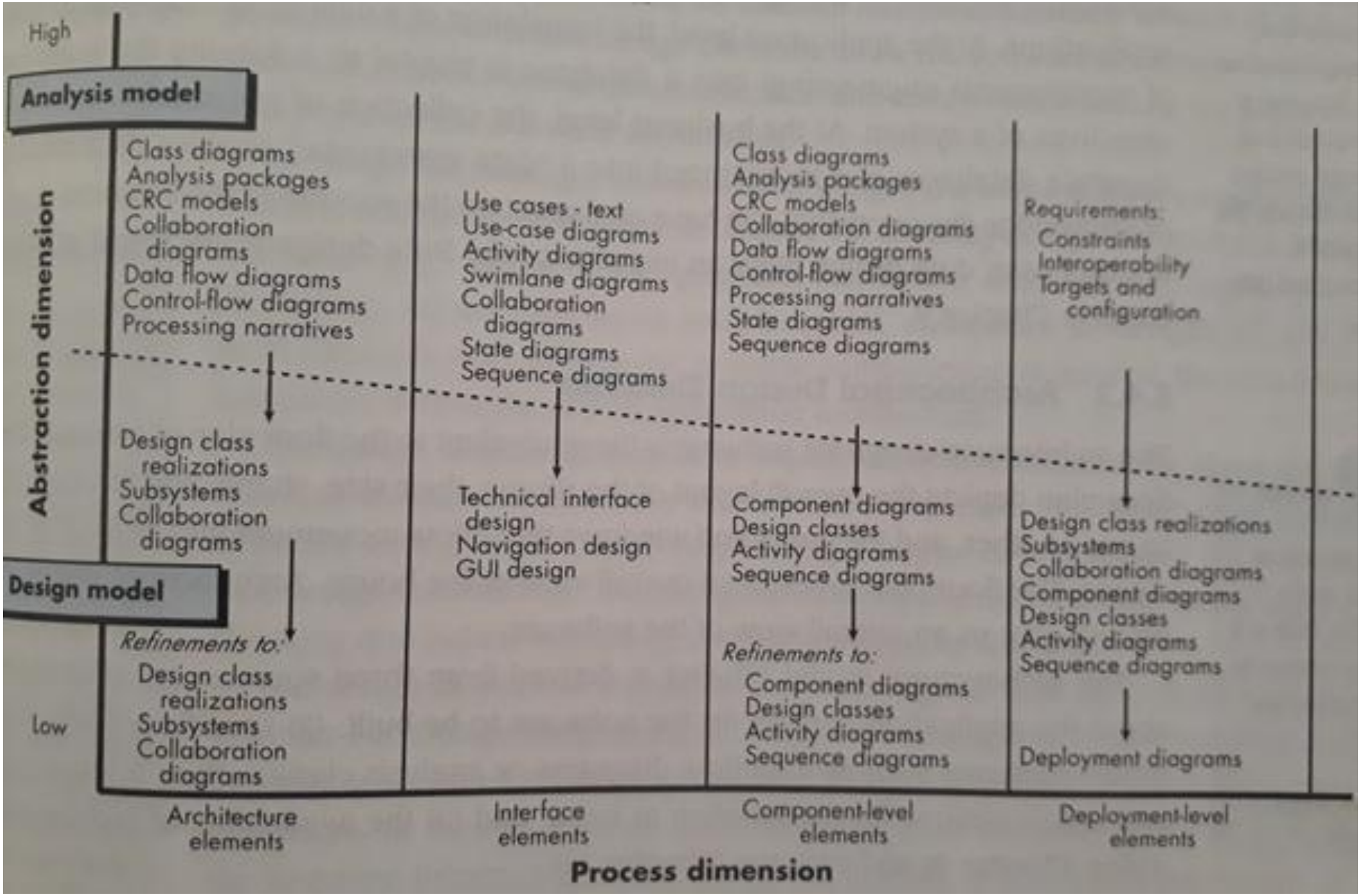

Figure 8.4 : Dimensions of the Design Model

1. Data Design Elements

- ✓ Data design creates a model of data &/or information that is represented at a high level of abstraction (the customer / user's view of data).
- ✓ This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- ✓ The structure of data has always been an important part of software design.
- ✓ At the program component level, the design of data structures & the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- ✓ At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- ✓ At the business level, the collection of information stored in disparate databases & reorganized into a "data warehouse" enables data mining.

2. Architectural Design Elements

- ✓ The architectural design for software is the equivalent to the floor plan of a house.
- ✓ The floor pan depicts the overall layout of the rooms; their size, shape, & relationship to one another; & the doors & windows that allow movement into & out of the rooms.
- ✓ The floor plan gives us an overall view of the house.
- ✓ Architectural design elements give us an overall view of the software.

- ✓ The architectural model is derived from 3 sources:-

1. Information about the application domain for the software to be built.
2. Specific requirements model elements such as data flow diagrams or analysis classes, their relationships & collaborations for the problem at hand.
3. The availability of architectural styles & patterns.

3. Interface Design Elements

- ✓ The interface design for software is analogous to a set of detailed drawings for the doors, windows, & external utilities of a house.

- ✓ There are 3 important elements of interface design:-

1. The user interface (UI).
2. External interfaces to other systems, devices, networks, or other producers or consumers of information.
3. Internal interfaces between various design components.

- ✓ These interface design elements allow the software to communicate externally & enable internal communication & collaboration among the components that populate the software architecture.

- ✓ UI design increasingly called ***Usability Design.***

- ✓ Usability design incorporates aesthetic elements (e.g. layout, color, graphics, metaphors, UI navigation), and technical elements (e.g. UI patterns, reusable components).
- ✓ UI is unique subsystem within overall application architecture.
- ✓ The design of external interfaces requires definitive information about the entity to which information is sent or received.
- ✓ In every case, this information should be collected during requirements engineering.
- ✓ The design of external interfaces should incorporate error checking & appropriate security features.
- ✓ The design of internal interfaces is closely aligned with component-level design.

- For example, the ***SafeHome()*** security function makes use of a control panel that allows a homeowner to control certain aspects of the security function.

- In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone.

- The **ControlPanel** class (as Figure 8.5) provides the behavior associated with a keypad, & therefore, it must implement the operations ***readKeyStroke()*** and ***decodeKey()***.

- If these operations are to be provided to other classes (in this case, ***WirelessPDA*** and ***MobilePhone***), it is useful to define an interface as shown in Figure 8.5.

- ✓ The interface, named ***KeyPad***, is shown as an << interface >> stereotype or as a small, labeled circle connected to the class with a line.
- ✓ The interface is defined with no attributes & the set of operations that are necessary to achieve the behavior of a keypad.

- ✓ The dashed line with an open triangle at its end (Figure 8.5) indicates that the ***controlPanel*** class provides ***KeyPad*** operations as part of its behavior.

- ✓ In UML, this is characterized as a ***Realization.***
- ✓ That is, part of the behavior of **ControlPanel** will be implemented by realizing ***KeyPad*** operations.
- ✓ These operations will be provided to other classes that access the interface.

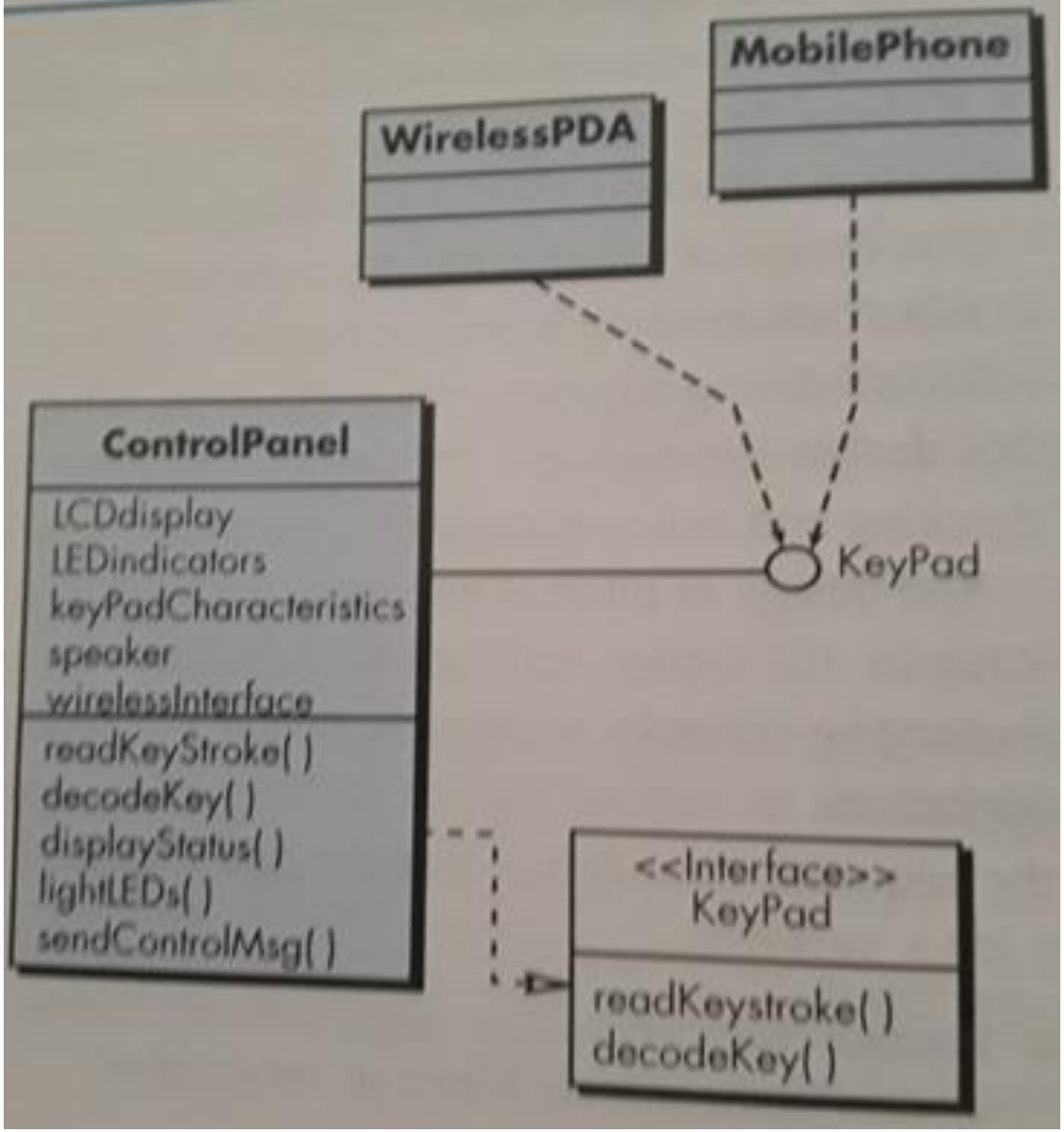

Figure 8.5 : Interface representation for Control-Panel

4. Component-Level Design Elements

- ✓ This design is the equivalent to a set of detailed drawings (& specifications) for each room in a house.
- ✓ These drawings depict writing & plumbing within each room, the location of wall switches, faucets, sinks showers, tubs, drains, cabinets, & closets.

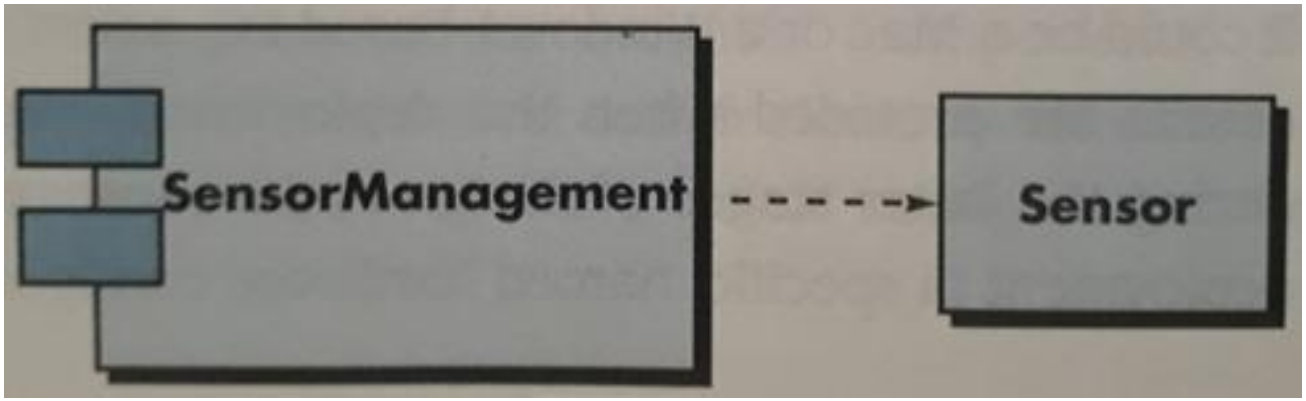

Figure 8.6 : A UML Component Diagram

- ✓ The component-level design for software fully describes the internal detail of each software component.

- ✓ To accomplish this, the component-level design defines data structures for all local data objects & algorithmic detail for all processing that occurs within a component & an interface that allows access to all component operations (behaviors).

- ✓ A component is represented in UML diagrammatic form as shown in Figure 8.6.
- ✓ In this Figure 8.6, a component named ***SensorManagament*** (part of ***Safehome*** security function) is represented.

- ✓ A dashed arrow connects the component to a class named ***Sensor*** that is assigned to it.
- ✓ The ***SensorManagement*** component performs all functions associated with ***SafeHome*** sensors including monitoring & configuring them.

4. Deployment-Level Design Elements

- ✓ It indicates how software functionality & subsystems will be allocated within the physical computing environment that will support the software.
- ✓ For example, the elements of the ***SafeHome*** product are configured to operate within 3 primary computing environments – a home based PC, the ***SafeHome*** control panel, & a server housed at CPI corp. (providing internet based access to the system).

- ✓ During design, a UML deployment diagram is developed & then refined as Figure 8.7.
- ✓ The subsystems (functionality) housed within each computing element are indicated.
- ✓ For example, personal computer houses subsystems that implement security, surveillance, home management, & communication features.

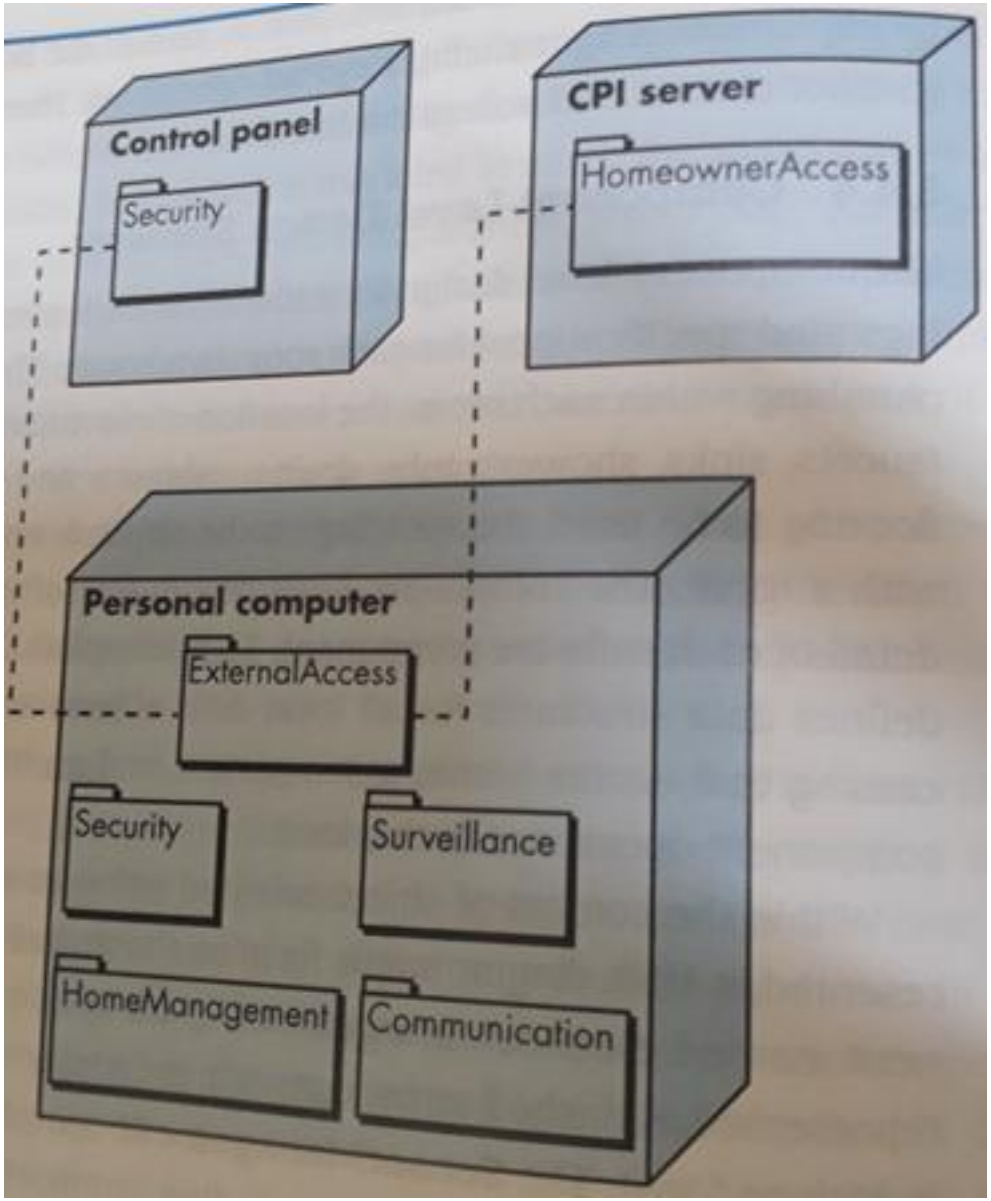

Figure 8.7 : A UML deployment diagram

- ✓ The diagram shown in Figure 8.7 is in *descriptor form*.
- ✓ This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details.
- ✓ For example, the “personal computer” is not further identified.
- ✓ It could be a Mac or Windows-based PC, a Sun workstation, or a Linux-box.

Pattern Based Software Design

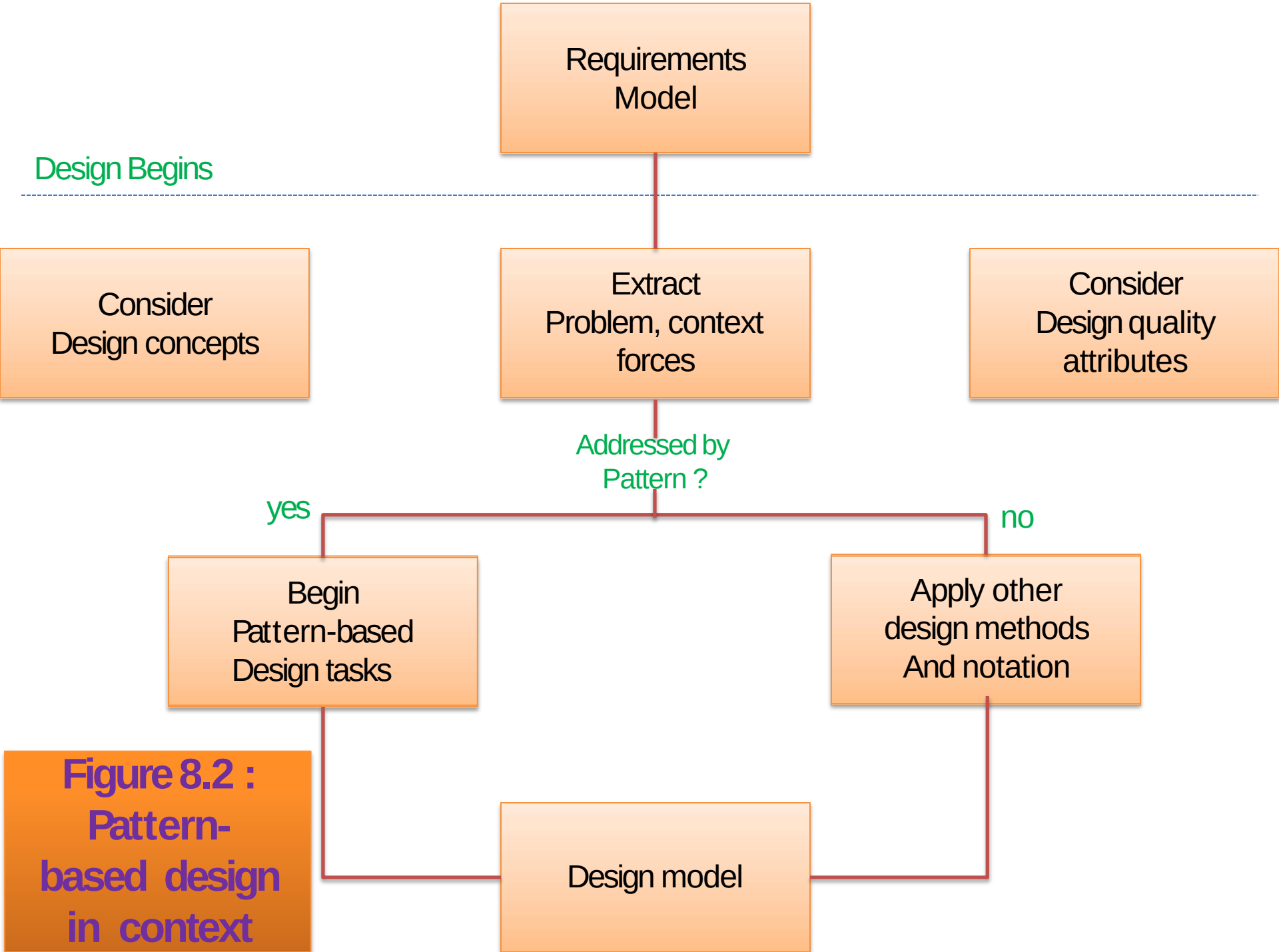

Figure 8.2 : Pattern-based design in context

- ✓ The role of Pattern based design in all of this is as Figure 8.2
- ✓ A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
- ✓ The requirements model describes the problem set, establishes the context, & identifies the system of forces that hold sway (influence).

Design Tasks

✓ The following design tasks are applied when a pattern based design philosophy is used :-

1. Examine the requirements model & develop a problem hierarchy :- Describe each problem & sub problem by isolating the problem, the context, & the system of forces that apply.

- Work from broad problems (high level of abstraction) to smaller sub problems (at lower levels of abstraction).

2. Determine if a reliable pattern language has been developed for the problem domain :- A Pattern language addresses problems associated with a specific application domain.

- If that level of pattern language specifically could not be found, the team would partition the ***SafeHome*** software problem into a series of generic problem domains.

3. Beginning with a broad problem, determine whether one or more architectural patterns is available for it :-

- If an architectural pattern is available, be certain to examine all collaborating patterns.

4. Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems & search for appropriate patterns to address them.

- It may be necessary to search through other pattern responsible as well as the list of patterns that corresponds to the architectural solution.
- If an appropriate pattern is found, adapt the design solution proposed & build a design model element that adequately represents.

5. **Repeat steps 2 through 5 until all broad problems have been addressed :-**

- The implication is to begin with big picture & elaborate to solve problems at increasingly more detailed levels.

6. **If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns :-**

- Proceed in a manner similar to steps 3,4 & 5.

7. **Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual patterns show promise , compare the problem to be solved against the existing patterns(s) presented:-**

- Be certain to examine context & forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.

8. **Be certain to refine the design as it is derived from patterns using design quality criteria as a guide :-**

Building Pattern Organizing Table

- ✓ As pattern-based design proceeds, you may encounter trouble in organizing & categorizing candidate patterns from multiple pattern languages & repositories.

- ✓ To help organize your evaluation of candidate patterns, Microsoft suggests the creation of a ***Pattern-Organizing table*** that takes the general form show in Figure 12.2

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem Statement …	PatternName (s)		PatternName (s)	
Problem Statement …		PatternName (s)		PatternName (s)
Problem Statement …	PatternName (s)			PatternName (s)
Architecture				
Problem Statement …		PatternName (s)		
Problem Statement …		PatternName (s)		PatternName (s)
Problem Statement …				
Component-level				
Problem Statement …		PatternName (s)	PatternName (s)	
Problem Statement …				PatternName (s)
Problem Statement …		PatternName (s)	PatternName (s)	
User Interface				
Problem Statement …		PatternName (s)	PatternName (s)	
Problem Statement …		PatternName (s)	PatternName (s)	
Problem Statement …		PatternName (s)	PatternName (s)	

Figure 12.2 :- A pattern organizing table

- ✓ A pattern organizing table can be implemented as a spreadsheet model using the form shown in the Figure 12.2.
- ✓ An abbreviated list of problem statements, organized by data/content, architecture, component-level, & user interface issues, is presented in the left-hand (shaded) column.

- ✓ Four Pattern types – database, application, implementation, & infrastructure – are listed across the top row.
- ✓ The names of candidate patterns are noted in the cells of the table.

- ✓ To provide entries for the organizing table, you'll search through pattern languages & repositories for patterns that address a particular problem statement.
- ✓ When one or more candidate patterns is found, it is entered in the row corresponding to the problem statement & the column corresponds to the pattern type.

Common Design Mistakes

- ✓ In some cases, not enough time has been spent to understand the underlying problem & as a consequence, you select a pattern that looks right but is inappropriate for the solution required.
- ✓ Once the wrong pattern is selected, you refuse to see your error & force-fit the pattern.
- ✓ In other cases, the problem has forces that are not considered by the pattern you have chosen, resulting in poor or erroneous fit.
- ✓ Sometimes a pattern is applied too literally & the required adaptions for your problem space are not implemented.

Architectural Design :

- ✓ **Architectural design** is concerned with understanding how a system should be organized and designing the overall structure of that system.

- ✓ The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

- ✓ You can design software architectures at two levels of abstraction, which called as ***architecture in the small and architecture in the large:***

1. Architecture in the small is concerned with the architecture of individual programs.

- At this level, we are concerned with the way that an individual program is decomposed into components.

2. Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

- These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

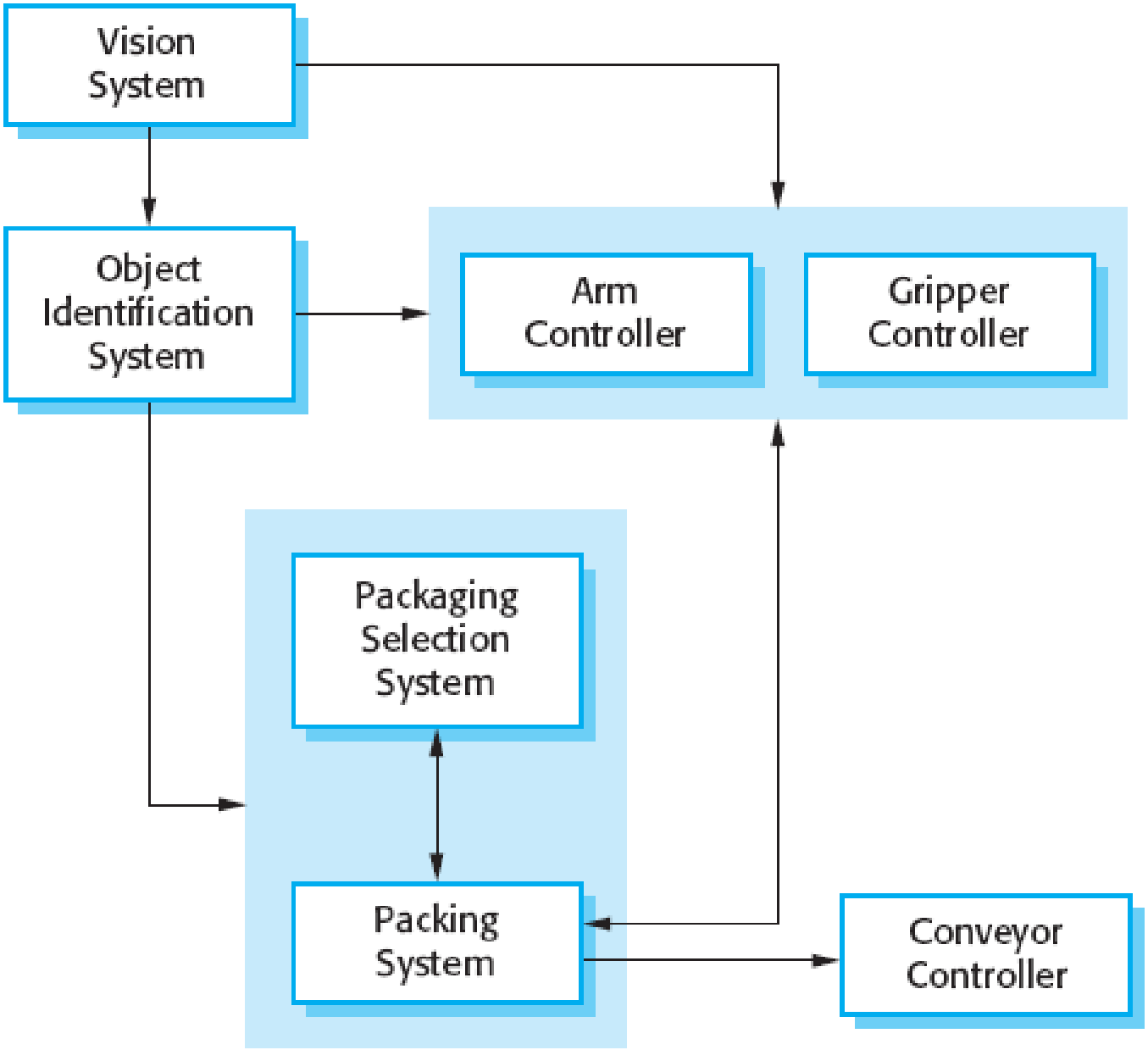

Figure 6.1 : The architecture of a packing robot control system

- ✓ System architectures are often modeled using simple block diagrams, as in **Figure 6.1**.
- ✓ **Each box** in the diagram represents a component.
- ✓ **Boxes** within boxes indicate that the component has been decomposed to sub-components.
- ✓ **Arrows** mean that data and or control signals are passed from component to component in the direction of the arrows.

- ✓ Software architecture is important because it affects the performance, robustness, three advantages of explicitly designing and documenting

- ✓ Software architecture: distributability, and maintainability of a system.

1. *Stakeholder communication*
2. *System analysis*
3. *Large-scale reuse*

Design Decisions

- ✓ Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

1. Performance

- ➢ If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network.

2. Security

- ➢ If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.

3. Safety

- If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components.
- This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.

4. Availability

- If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system.

5. Maintainability

- If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Architectural views

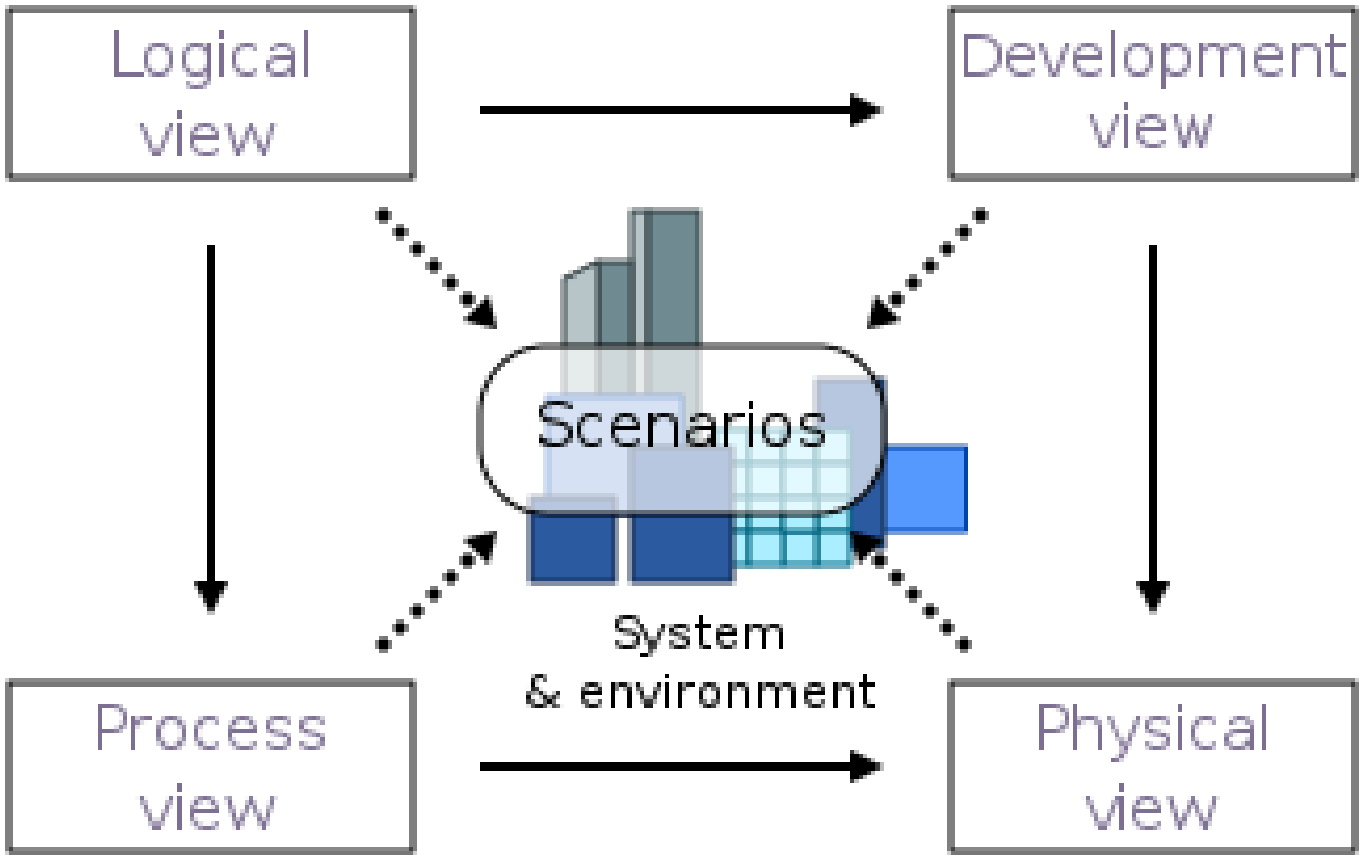

Figure 2 : 4+1 Architectural View Model

- ✓ The views that he suggests are:

1. **A Logical View**, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. **A Process View**, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about nonfunctional system characteristics such as performance and availability.

3. **A Development View**, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.

4. **A Physical View**, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

- ✓ **Conceptual view** is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused.

- ✓ In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making.

- ✓ They are a way of communicating the essence of a system to different stakeholders.

Architectural patterns

- ✓ The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems.
- ✓ Architectural patterns were proposed in the 1990s under the name ‘architectural styles’.
- ✓ Next point describes the well-known **Model-View-Controller pattern**.
- ✓ This pattern is the basis of interaction management in many web-based systems.
- ✓ The stylized pattern description includes the pattern name, a brief description (with an associated graphical model), and an example of the type of system where the pattern is used.

❖ **Pattern Name : MVC**

❖ **Description :** Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other.

- ✓ The Model component manages the system data and associated operations on that data.
- ✓ The View component defines and manages how the data is presented to the user.
- ✓ The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.

❖ **Example :** Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.

- **When used :** Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.

- **Advantages :** Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.

- **Disadvantages :** Can involve additional code and code complexity when the data model and interactions are simple.

- ✓ You should also include information about when the pattern should be used and its advantages and disadvantages.
- ✓ Graphical models of the architecture associated with the MVC pattern are shown in Figures 6.3 and 6.4.
- ✓ These present the architecture from different views—Figure 6.3 is a conceptual view and Figure 6.4 shows a possible run-time architecture when this pattern is used for interaction management in a web-based system.

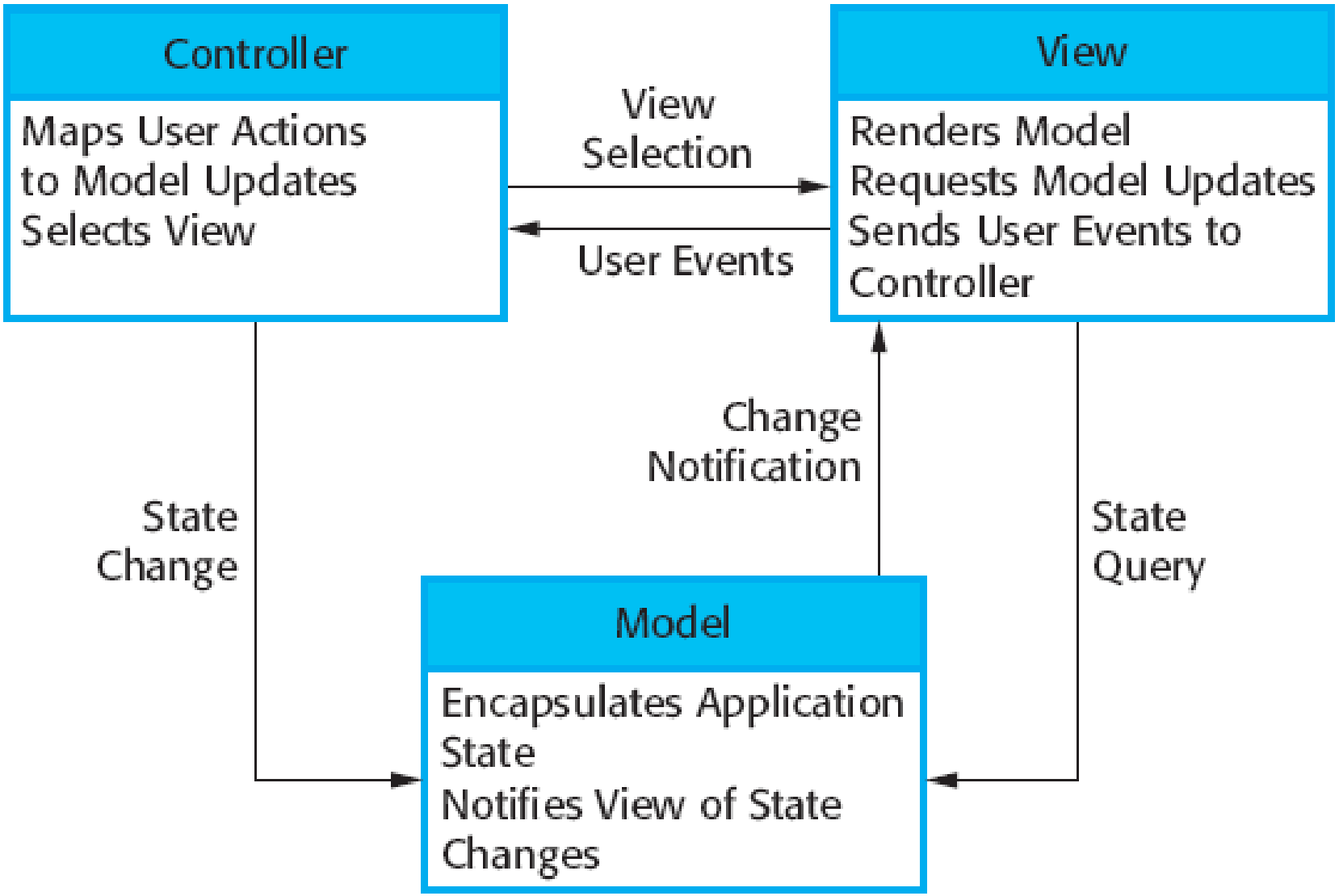

Figure 6.3 : The organization of the MVC

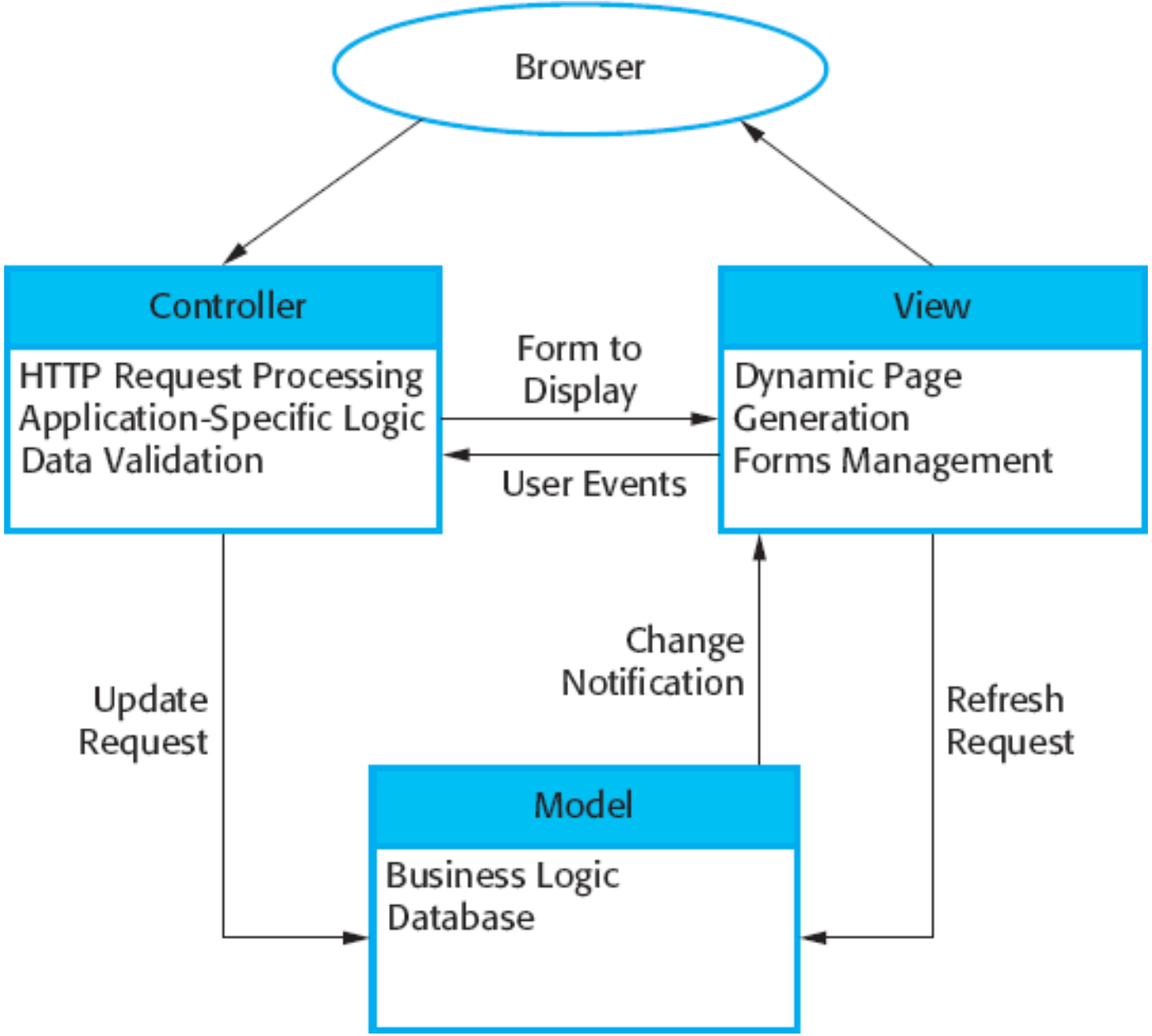

Figure 6.4 : Web application architecture using the MVC pattern

1. Layered Architecture

- ✓ The notions of separation and independence are fundamental to architectural design because they allow changes to be localized.
- ✓ The MVC pattern separates elements of a system, allowing them to change independently.
- ✓ For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model.
- ✓ The layered architecture pattern is another way of achieving separation and independence.
- ✓ This pattern is shown as below :-

- **Name of Pattern : Layered Architecture Pattern**

- **Description :** Organizes the system into layers with related functionality associated with each layer.
  - ✓ A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.

- **Example :** A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.

- **When Used :** Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.

- **Advantages :** Allows replacement of entire layers so long as the interface is maintained.

- ✓ Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.

- **Disadvantages :** In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it.

- ✓ Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

- ✓ This layered approach supports the incremental development of systems.
- ✓ As a layer is developed, some of the services provided by that layer may be made available to users.
- ✓ The architecture is also changeable and portable.
- ✓ So long as its interface is unchanged, a layer can be replaced by another, equivalent layer.
- ✓ When layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected.
- ✓ **Figure 6.6** is an example of a layered architecture with four layers.

- ✓ The lowest layer includes system support software—typically database and operating system support.
- ✓ The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components.
- ✓ The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities.
- ✓ the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.

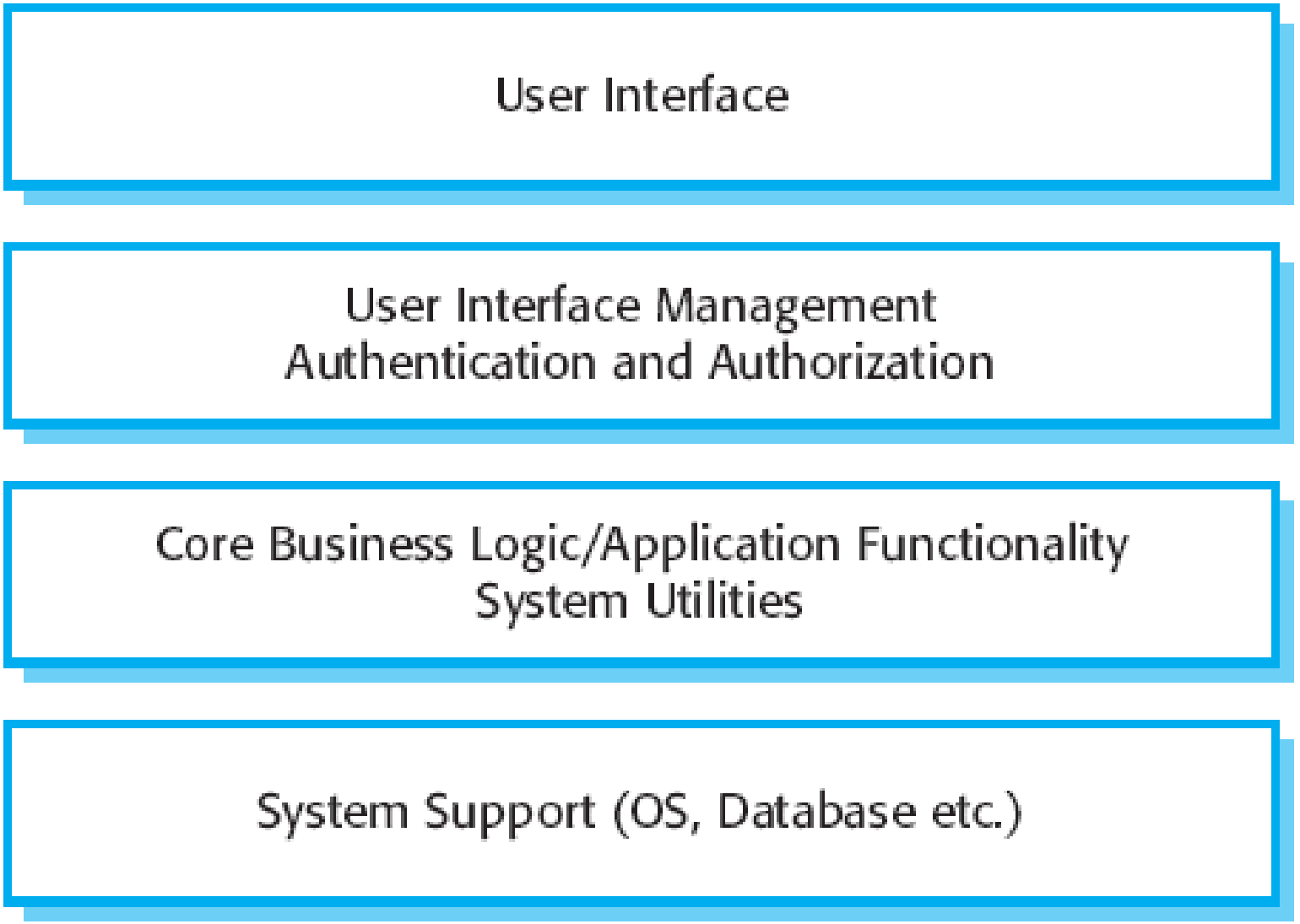

Figure 6.6 : A generic layered architecture

- ✓ Figure 6.7 is an example of how this layered architecture pattern can be applied to a library system called LIBSYS, which allows controlled electronic access to copyright material from a group of university libraries.

- ✓ This has a five-layer architecture, with the bottom layer being the individual databases in each library.

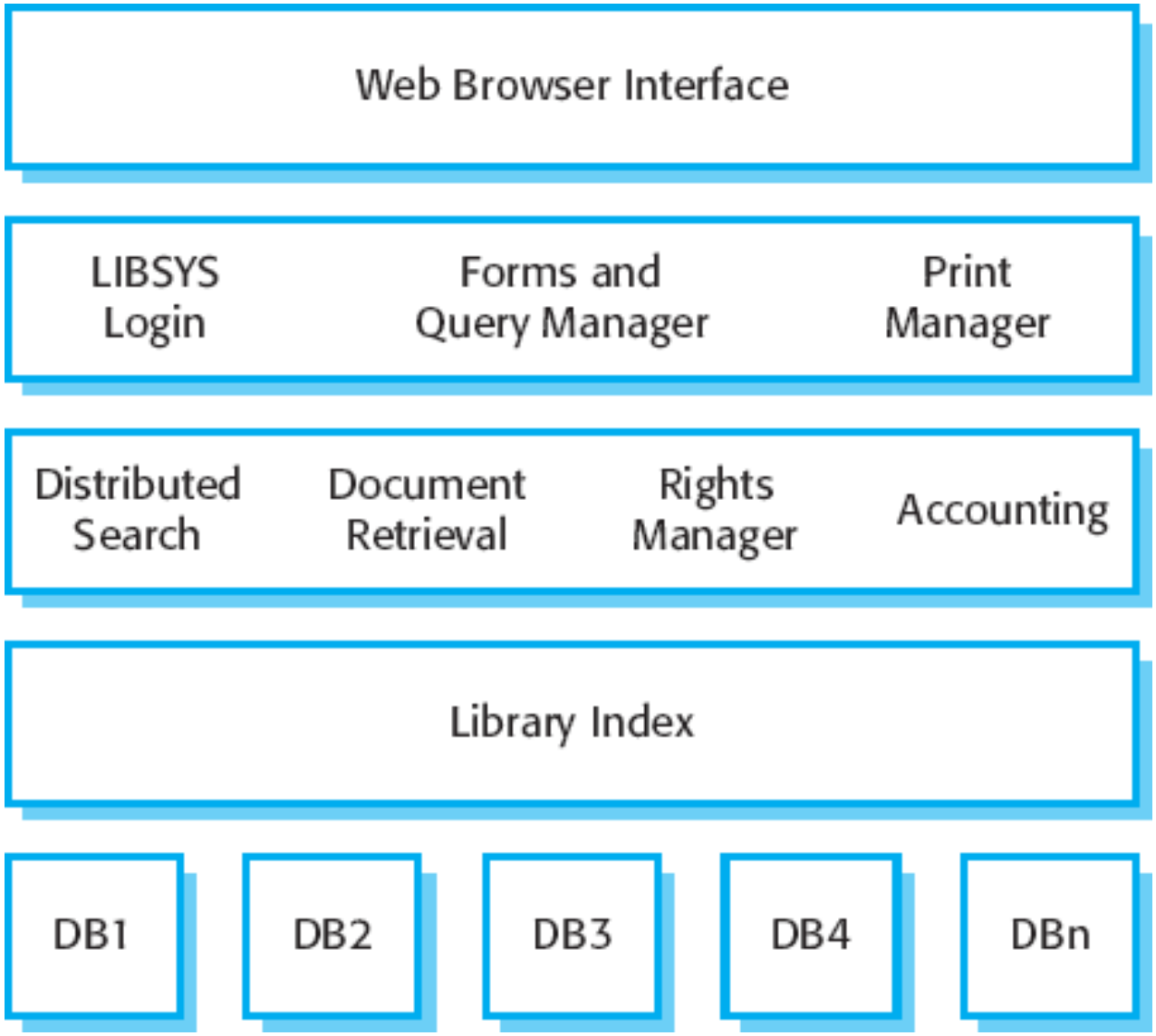

Figure 6.7 : The architecture of the LIBSYS system

2. Repository Architecture

- ✓ The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system.
- ✓ The Repository pattern describes how a set of interacting components can **share data.**
- ✓ The majority of systems that use large amounts of data are organized around a shared database or repository.
- ✓ **This model is therefore suited to applications in which data is generated by one component and used by another.**
- ✓ **Examples** of this type of system include command and control systems, management information systems, CAD systems, and interactive development environments for software.

- **Name of Pattern : The Repository Pattern**

- **Description :** All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.

- **Example :** Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.

- **When Used :** You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time.
- ✓ You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.

- **Advantages :** Components can be independent—they do not need to know of the existence of other components.

- ✓ Changes made by one component can be propagated to all components.
- ✓ All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.

- **Disadvantages :** The repository is a single point of failure so problems in the repository affect the whole system.

- ✓ May be inefficiencies in organizing all communication through the repository.

- ✓ Distributing the repository across several computers may be difficult.

- ✓ **Figure 6.9** is an illustration of a situation in which a repository might be used.
- ✓ This diagram shows an IDE that includes different tools to support model-driven development.
- ✓ The repository in this case might be a version-controlled environment that keeps track of changes to software and allows rollback to earlier versions.

- ✓ Organizing tools around a repository is an efficient way to share large amounts of data.
- ✓ There is no need to transmit data explicitly from one component to another.

- ✓ In the example shown in **Figure 6.9**, the repository is passive and control is the responsibility of the components using the repository.

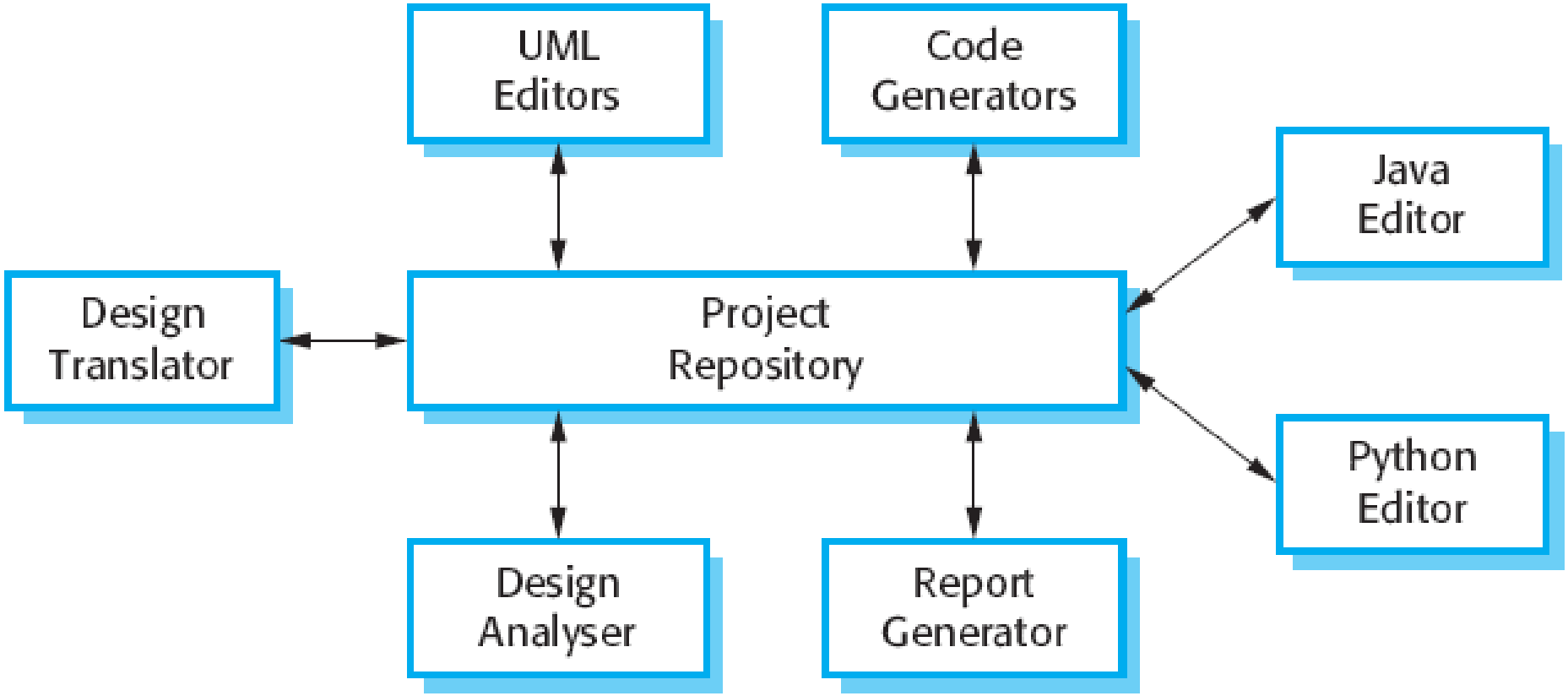

Figure 6.9 : A repository architecture for an IDE

3. Client - Server Architecture

- ✓ The repository pattern is concerned with the static structure of a system and does not show its run-time organization.
- ✓ Client server architecture is used for run-time organization for **distributed systems**.
- ✓ A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services.
- ✓ The major components of this model are:

1. A set of servers that offer services to other components.

✓ Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.

2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.

3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

- **Name of Pattern : The Client–server**

- **Description :** In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

- **Example :** Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.

- **When Used :** Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.

- **Advantages :** The principal advantage of this model is that servers can be distributed across a network.
- ✓ General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.

- **Disadvantages :** Each service is a single point of failure so susceptible to denial of service attacks or server failure.
- ✓ Performance may be unpredictable because it depends on the network as well as the system.
- ✓ May be management problems if servers are owned by different organizations.

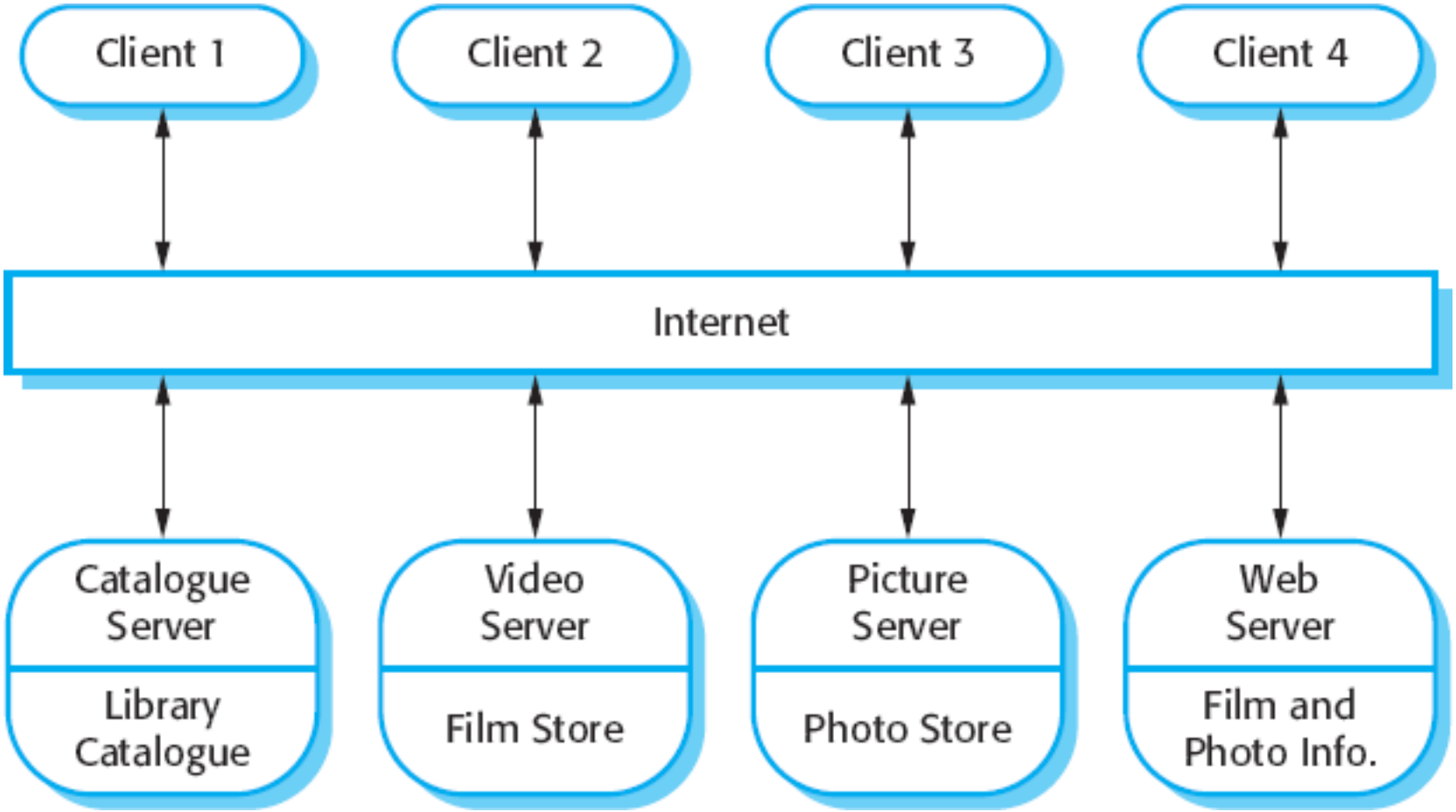

Figure 6.11 : A client—server architecture for a film library

4. Pipe and Filter Architecture

- ✓ This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs.
- ✓ Data flows from one to another and is transformed as it moves through the sequence.
- ✓ Each processing step is implemented as a transform. Input data flows through these transforms until converted to output.
- ✓ The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

- **Name of Pattern : The Pipe and filter**

- **Description :** The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.

- **Example :** Figure 6.13 is an example of a pipe and filter system used for processing invoices.

- **When Used :** Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.

- **Advantages :** Easy to understand and supports transformation reuse.
- ✓ Workflow style matches the structure of many business processes.
- ✓ Evolution by adding transformations is straightforward.
- ✓ Can be implemented as either a sequential or concurrent system.

- **Disadvantages :** The format for data transfer has to be agreed upon between communicating transformations.
- ✓ Each transformation must parse its input and unparse its output to the agreed form.
- ✓ This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

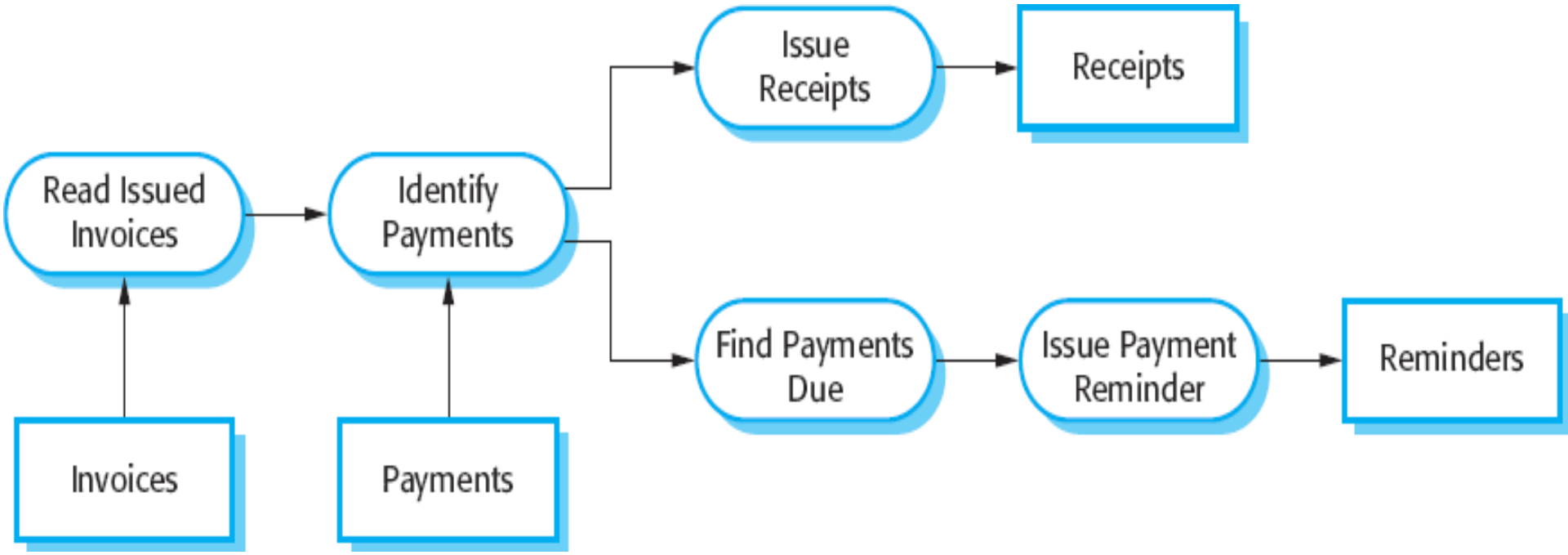

Figure 6.13 : An example of the pipe and filter architecture

Application Architectures

- ✓ The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without reimplementation.
- ✓ As a software designer, you can use models of application architectures in a number of ways:

1. ***As a starting point for the architectural design process :-***

- If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture.

2. ***As a design checklist :-***

- If you have developed an architectural design for an application system, you can compare this with the generic application architecture.
- You can check that your design is consistent with the generic architecture.

3. As a way of organizing the work of the development team :-

- The application architectures identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel.
- You can assign work to group members to implement different components within the architecture.

4. As a means of assessing components for reuse :-

- If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. *As a vocabulary for talking about types of applications :-*

- If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

1. Transaction Processing Systems

- ✓ Transaction processing (TP) systems are designed to process user requests for information from a database, or requests to update a database.
- ✓ Technically, a database transaction is sequence of operations that is treated as a single unit (an atomic unit).
- ✓ All of the operations in a transaction have to be completed before the database changes are made permanent.
- ✓ This ensures that failure of operations within the transaction does not lead to inconsistencies in the database.

- ✓ From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as 'find the times of flights from London to Paris'.
- ✓ If the user transaction does not require the database to be changed then it may not be necessary to package this as a technical database transaction.

- ✓ An example of a transaction is a customer request to withdraw money from a bank account using an ATM.

- ✓ This involves getting details of the customer's account, checking the balance, modifying the balance by the amount withdrawn, and sending commands to the ATM to deliver the cash.

- ✓ Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

- ✓ Transaction processing systems are usually interactive systems in which users make asynchronous requests for service.

- ✓ **Figure 6.14** illustrates the conceptual architectural structure of TP applications.

- ✓ First a user makes a request to the system through an I/O processing component.
- ✓ The request is processed by some application specific logic.
- ✓ A transaction is created and passed to a transaction manager, which is usually embedded in the database management system.
- ✓ After the transaction manager has ensured that the transaction is properly completed, it signals to the application that processing has finished.

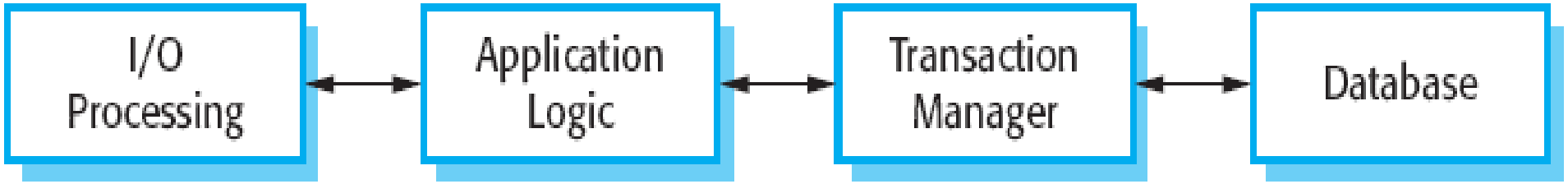

Figure 6.14 :The Database structure of transaction processing applications

- ✓ Transaction processing systems may be organized as a 'pipe and filter' architecture with system components responsible for input, processing, and output.

- ✓ For example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM.

- ✓ The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server.
- ✓ The input and output components are implemented as software in the ATM and the processing component is part of the bank's database server.

- ✓ **Figure 6.15** shows the architecture of this system, illustrating the functions of the input, process, and output components.

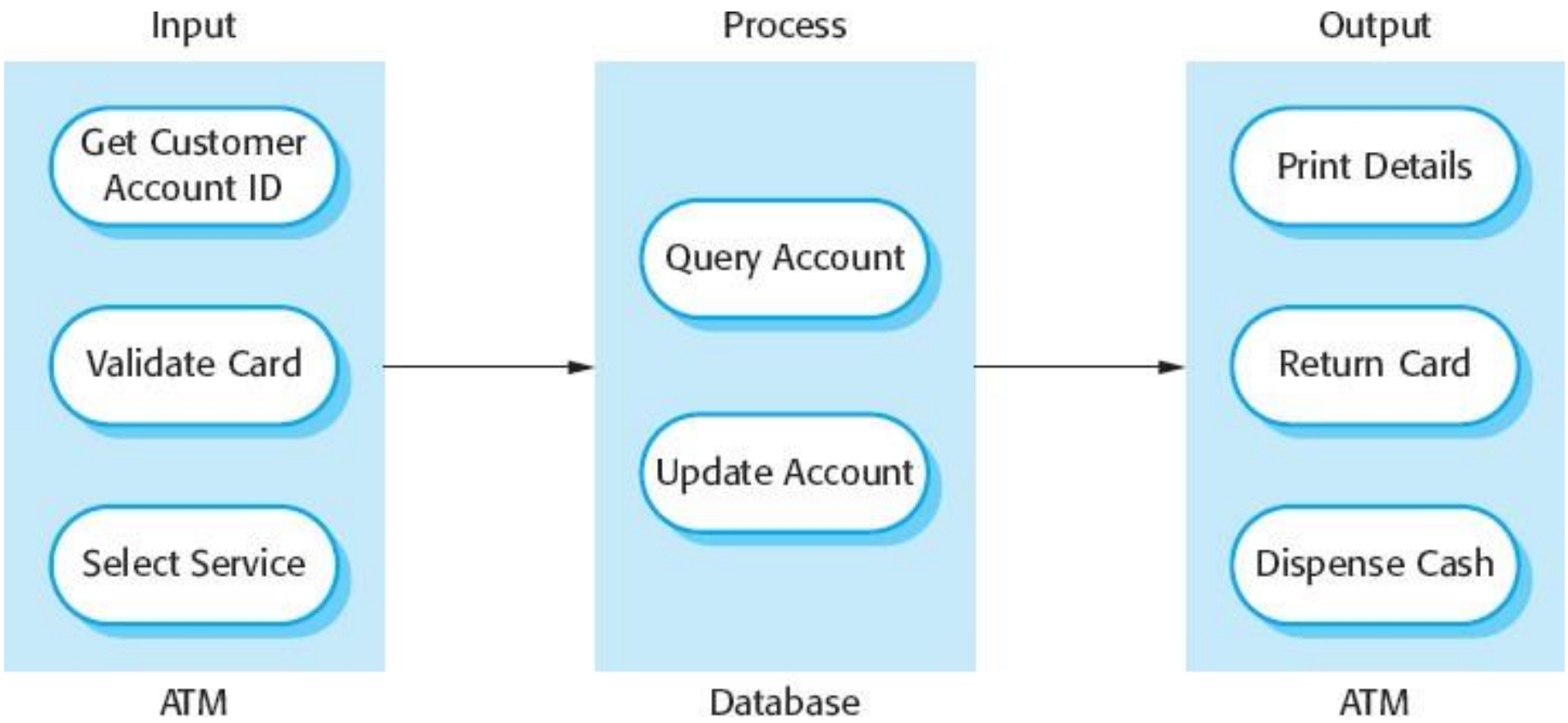

Figure 6.15 :The software architecture of an ATM system

2. Information systems

- ✓ All systems that involve interaction with a shared database can be considered to be transaction-based information systems.
- ✓ An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital.
- ✓ Increasingly, information systems are web-based systems that are accessed through a web browser.

- ✓ Figure 6.16 a very general model of an information system.

- ✓ The system is modeled using a layered approach where the top layer supports the user interface and the bottom layer is the system database.

- ✓ The user communications layer handles all input and output from the user interface, and the information retrieval layer includes application-specific logic for accessing and updating the database.

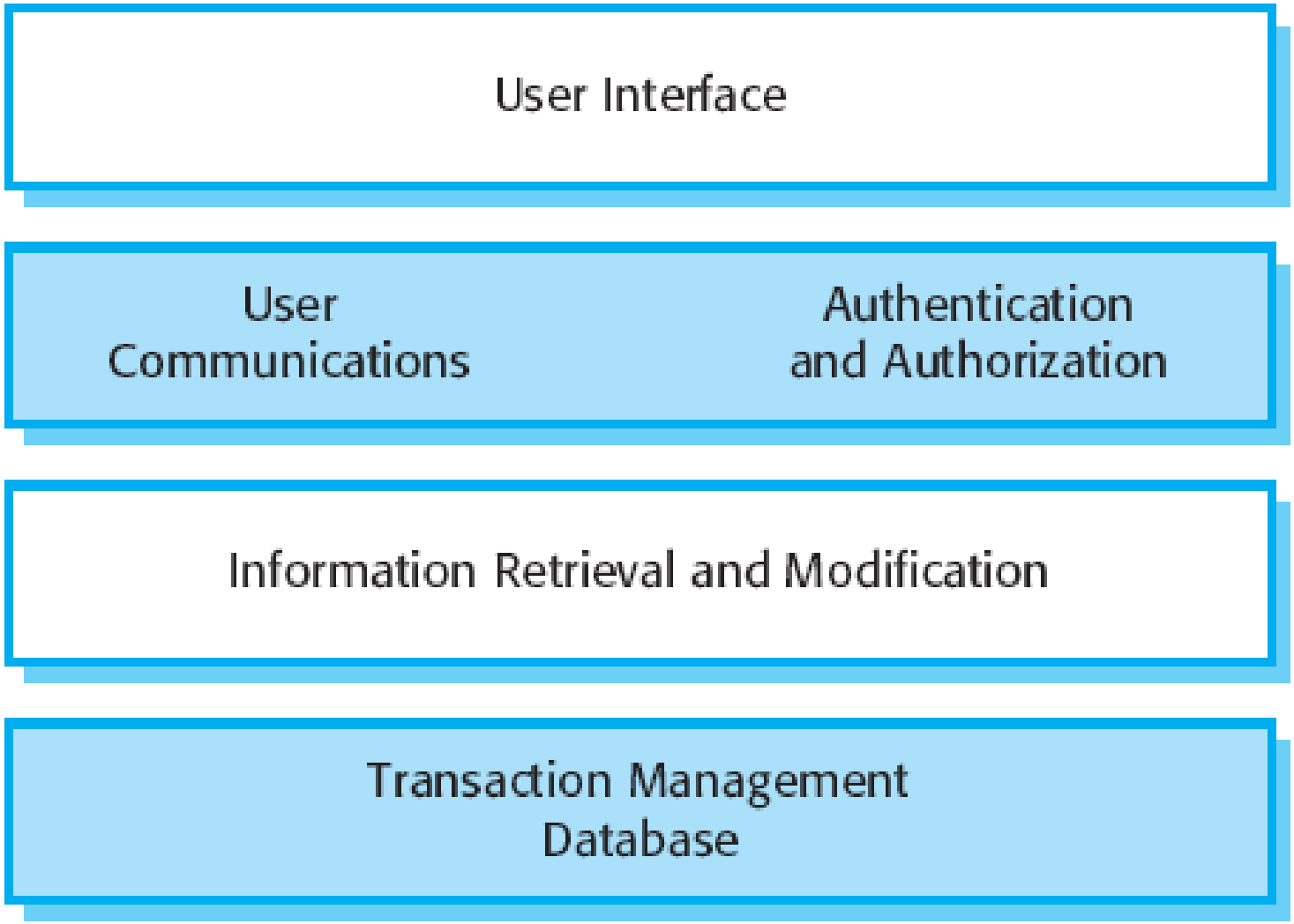

Figure 6.16 : Layered information system architecture

- ✓ As an example of an instantiation of this layered model, **Figure 6.17** shows the architecture of the MHC-PMS.

- ✓ Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems.
- ✓ Have added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:

1. The top layer is responsible for implementing the user interface. In this case, the UI has been implemented using a web browser.

2. The second layer provides the user interface functionality that is delivered through the web browser.

- It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role.
- This layer includes form and menu management components that present information to users, and data validation components that check information consistency.

3. The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
4. Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

3. Language processing systems

- ✓ Language processing systems translate a natural or artificial language into another representation of that language and, for programming languages, may also execute the resulting code.
- ✓ In software engineering, compilers translate an artificial programming language into machine code.
- ✓ Other language-processing systems may translate an XML data description into commands to query a database or to an alternative XML representation.
- ✓ Natural language processing systems may translate one natural language to another e.g., French to Norwegian.

- ✓ A possible architecture for a language processing system for a programming language is illustrated in Figure 6.18.

- ✓ The source language instructions define the program to be executed and a translator converts these into instructions for an abstract machine.

- ✓ These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment.

- ✓ The output of the process is the result of interpreting the instructions on the input data.

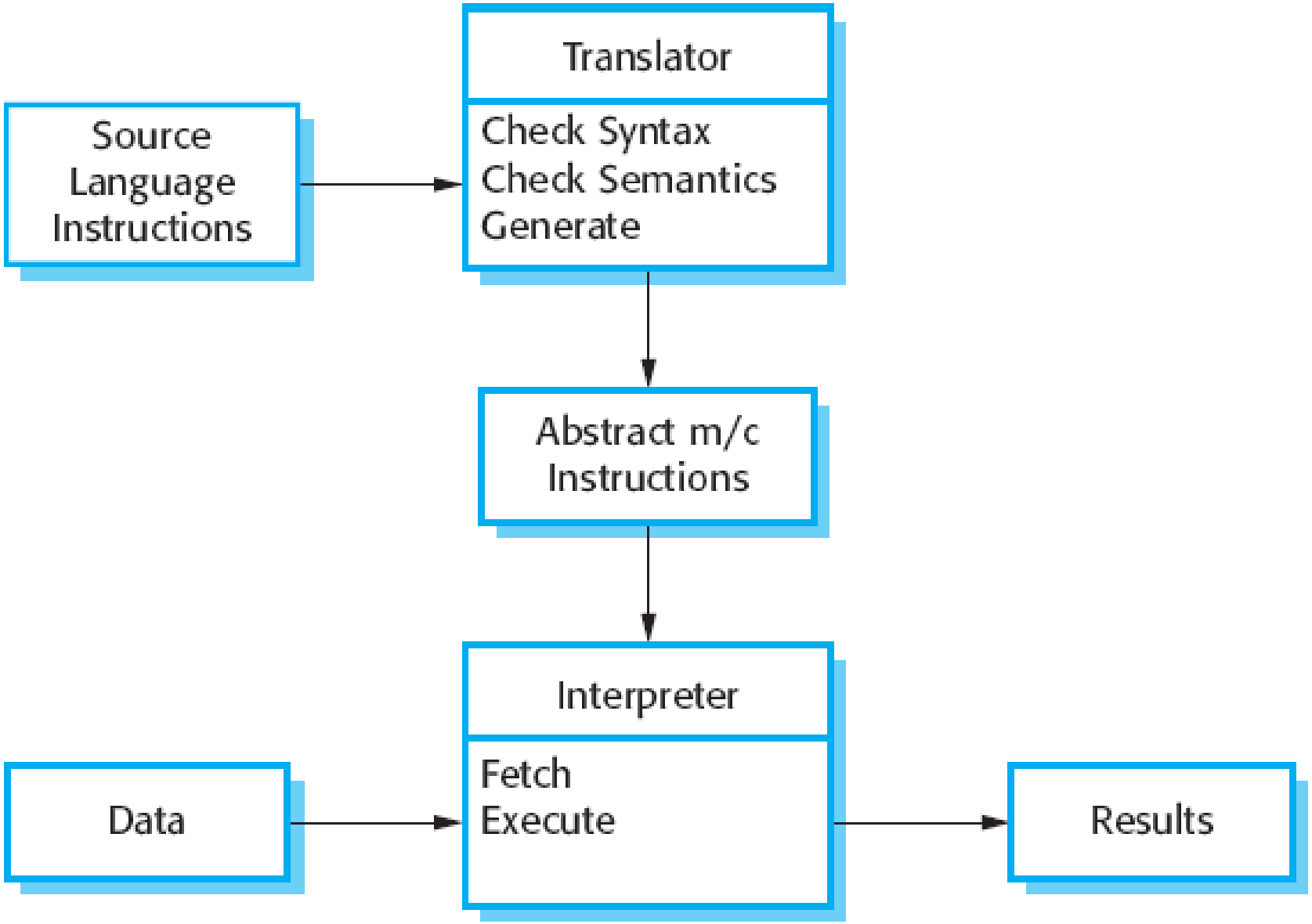

Figure 6.18 : The architecture of a language processing system

- ✓ Of course, for many compilers, the interpreter is a hardware unit that processes machine instructions and the abstract machine is a real processor.

- ✓ However, for dynamically typed languages, such as Python, the interpreter may be a software component.

- ✓ Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.19) that includes the following components:

1. A **lexical analyzer**, which takes input language tokens and converts them to an internal form.

2. A **symbol table**, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.

3. A **syntax analyzer**, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.

4. A **syntax tree**, which is an internal structure representing the program being compiled.

5. A **semantic analyzer** that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.

6. A **code generator** that 'walks' the syntax tree and generates abstract machine code.

- ✓ This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another.

- ✓ It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger or a program prettyprinter.

- ✓ In this situation, changes from one component need to be reflected immediately in other components.

- ✓ It is better, therefore, to organize the system around a repository, as shown in Figure 6.20.

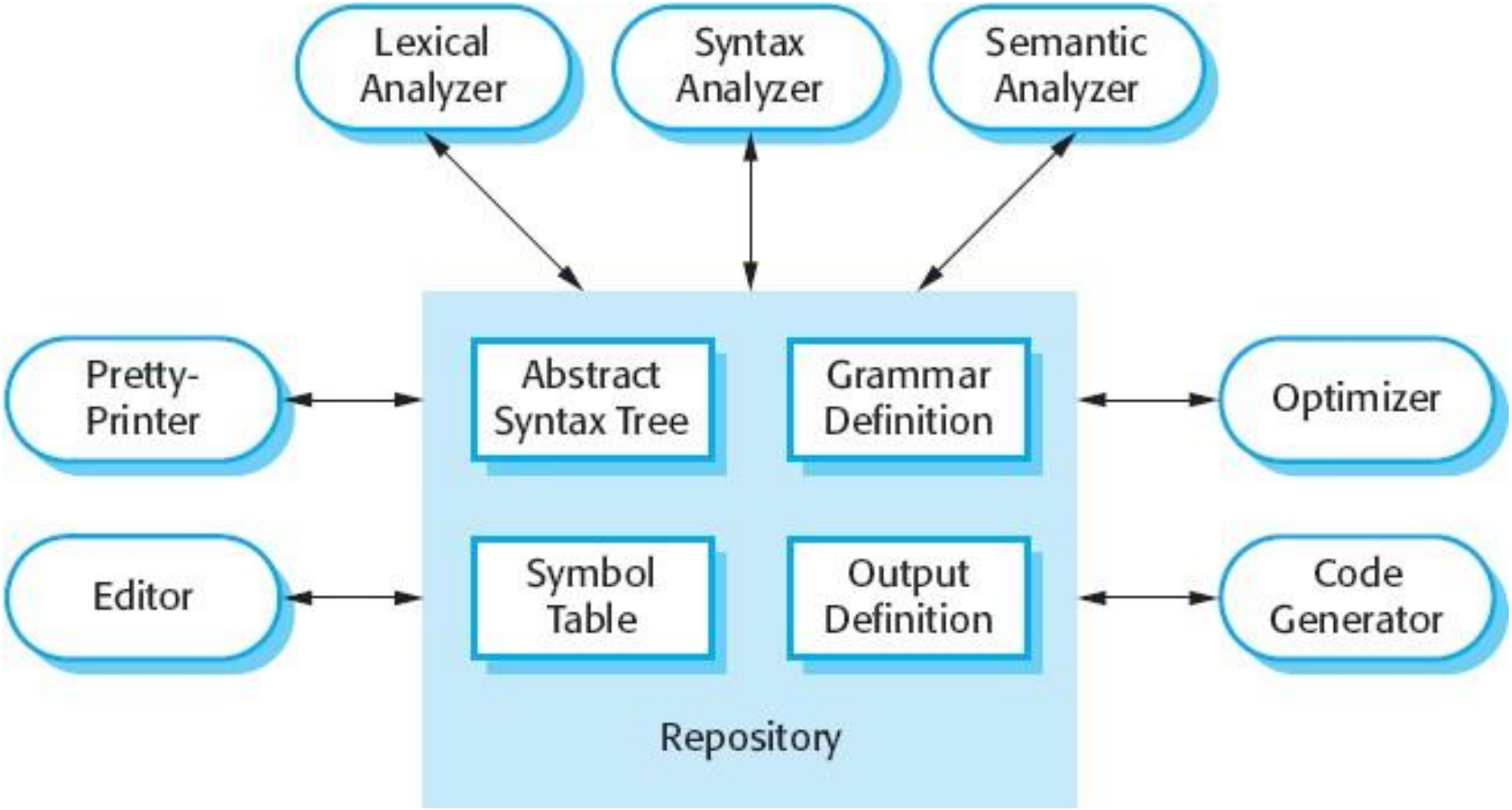

Figure 6.20 : A repository architecture for a language processing system

- ✓ This figure illustrates how a language processing system can be part of an integrated set of programming support tools.

- ✓ In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it.

- ✓ Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository.

- ✓ Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed and a prettyprinter can create listings of the program in a format that is easy to read.

Component level Design

What is a Component ?

- A component is a modular building block for computer software.
- A component is a modular, deployable, and replaceable part of a system that encapsulates implementation & exposes a set of interfaces.
- Components reside within a software architecture, they must communicate and collaborate with other components & with entities (e.g. other systems, devices, people) that exist outside the boundaries of the software.

An Object Oriented View

- ✓ A component contains a set of collaborating classes.
- ✓ Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
- ✓ As part of design elaboration, all interfaces that enable the classes to communicate & collaborate with other design classes must also be defined.
- ✓ To accomplish this, you begin with the requirements model & elaborate, analysis classes (for components that relate to the problem domain) & infrastructure classes (for components that provide support services for the problem domain).
- ✓ To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop.

- ✓ The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, & then pass the job on to an automated production facility.

- ✓ During requirements engineering, an analysis class called **PrintJob** was derived.
- ✓ The attributes & operations defined during analysis are noted at the top of Figure 10.1.

- ✓ During architectural design, **PrintJob** is defined as a component within the software architecture & is represented using shorthand UML notation shown in middle right of the Figure 10.1.

- ✓ Note that **PrintJob** has 2 interfaces, ***computeJob***, which provides job costing capability, & ***initiateJob***, which passes the job along to the production facility.

- ✓ These are represented using the "lollipop" symbols shown to the left of the component box.
- ✓ Component level design begins at this point.
- ✓ The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation.
- ✓ The original analysis class is elaborated to flesh out all attributes & operations required to implement the class as the component **PrintJob**.
- ✓ Referring to the lower right portion of Figure 10.1, the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component.
- ✓ The interfaces ***computeJob & initiateJob*** imply communication & collaboration with other components (not shown here).

- ✓ For example, the operation **computePageCost()** (part of the ***computeJob*** interface) might collaborate with a **PricingTable** component that contains job pricing information.
- ✓ The ***checkPriority()*** operation (part of the ***initiateJob*** interface) might collaborate with a **JobQueue** component to determine the types & priorities of jobs currently awaiting production.
- ✓ This elaboration activity is applied to every component defined as part of the architectural design.
- ✓ Once it is completed, further elaboration is applied to each attribute, operation, & interface.
- ✓ The data structures appropriate for each attribute must be specified.

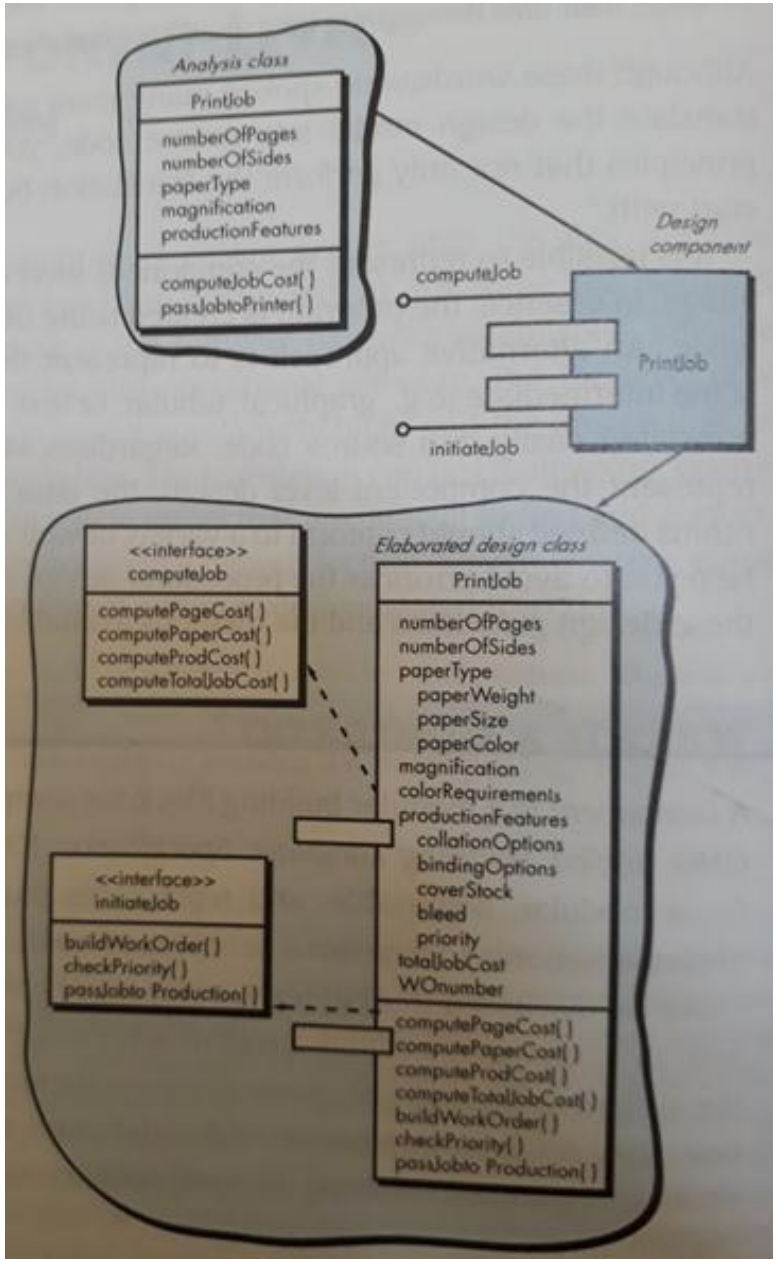

Figure 10.1 : Elaboration of a design component

Traditional View

- ✓ In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, & an interface that enables the component to be invoked & data to be passed to it.

- ✓ A traditional component, also called a **Module**, resides within the software architecture & serves one of 3 important roles :-

1. A Control Component that coordinates the invocation of all other problem domain components.
2. A Problem Domain component that implements a complete or partial function that is required by the customer.
3. An Infrastructure component that is responsible for functions that support the processing required in the problem domain.

- ✓ Like OO components, traditional software components are derived from the analysis model.

- ✓ As Figure 10.2, Each box represents a software component.
- ✓ Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class.
- ✓ In this case, however, each operation is represented as a separate module that is invoked as shown in Figure 10.2. Other modules are used to control processing & are therefore control components.

- ✓ During component level design, each module in Figure 10.2 is elaborated.

- ✓ The module interface is defined explicitly.
- ✓ That is, each data or control object that flows across the interface is represented.

- ✓ The data structures that are used internal to the module are defined.

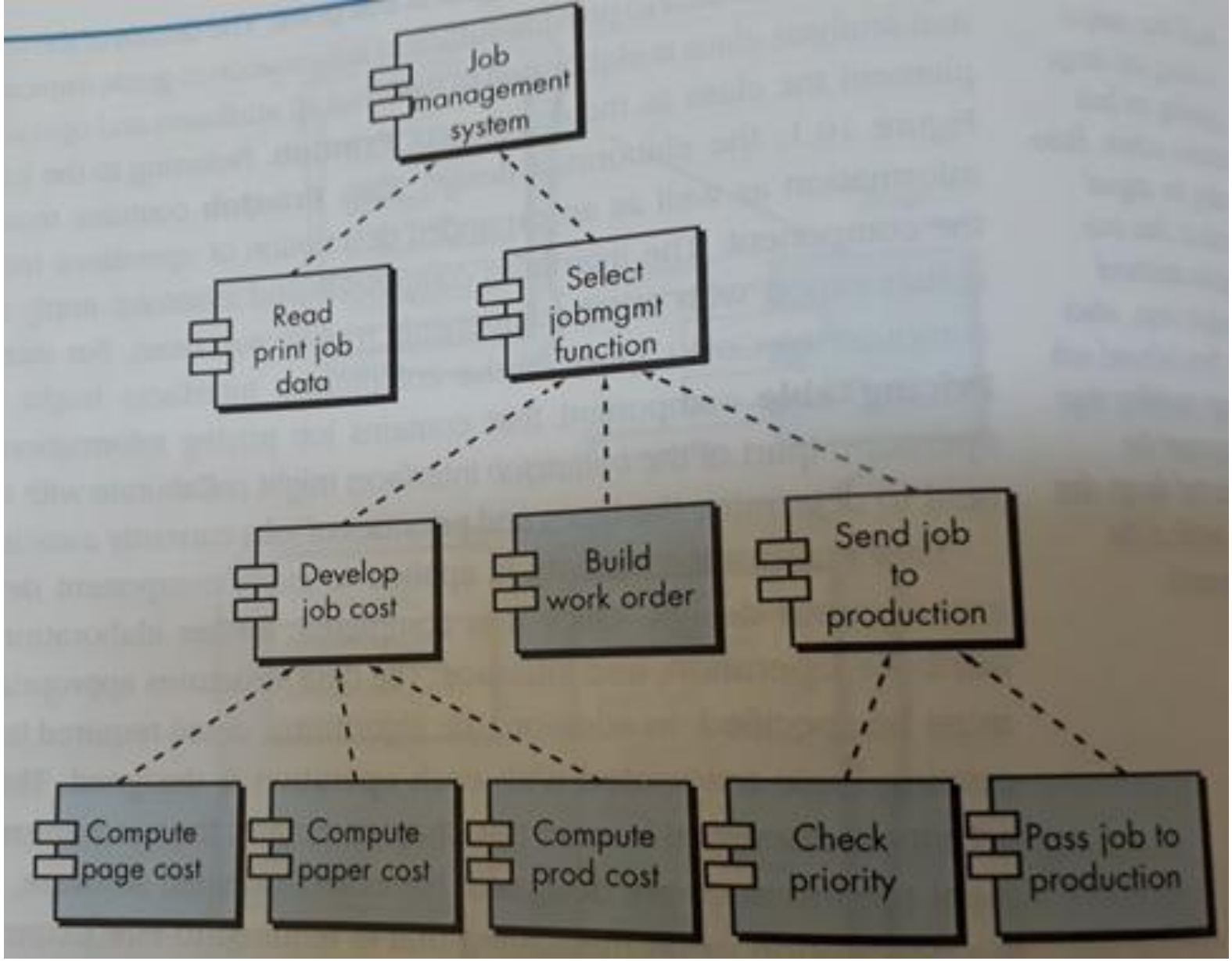

Figure 10.2 : Structure Chart for a Traditional System

- ✓ To illustrate this process, consider the module ***ComputePageCost***. The intent of this module is to compute the printing cost per page based on specifications provided by the customer.

- ✓ Data required to perform this function are :- **number of pages in the document, total number of documents to be produced, one-or-two-side printing, color requirements, & size requirements.**

- ✓ These data are passed to ***ComputePageCost*** via the module's interface.

- ✓ ***ComputePageCost*** uses these data to determine a page cost that is based on the interface.

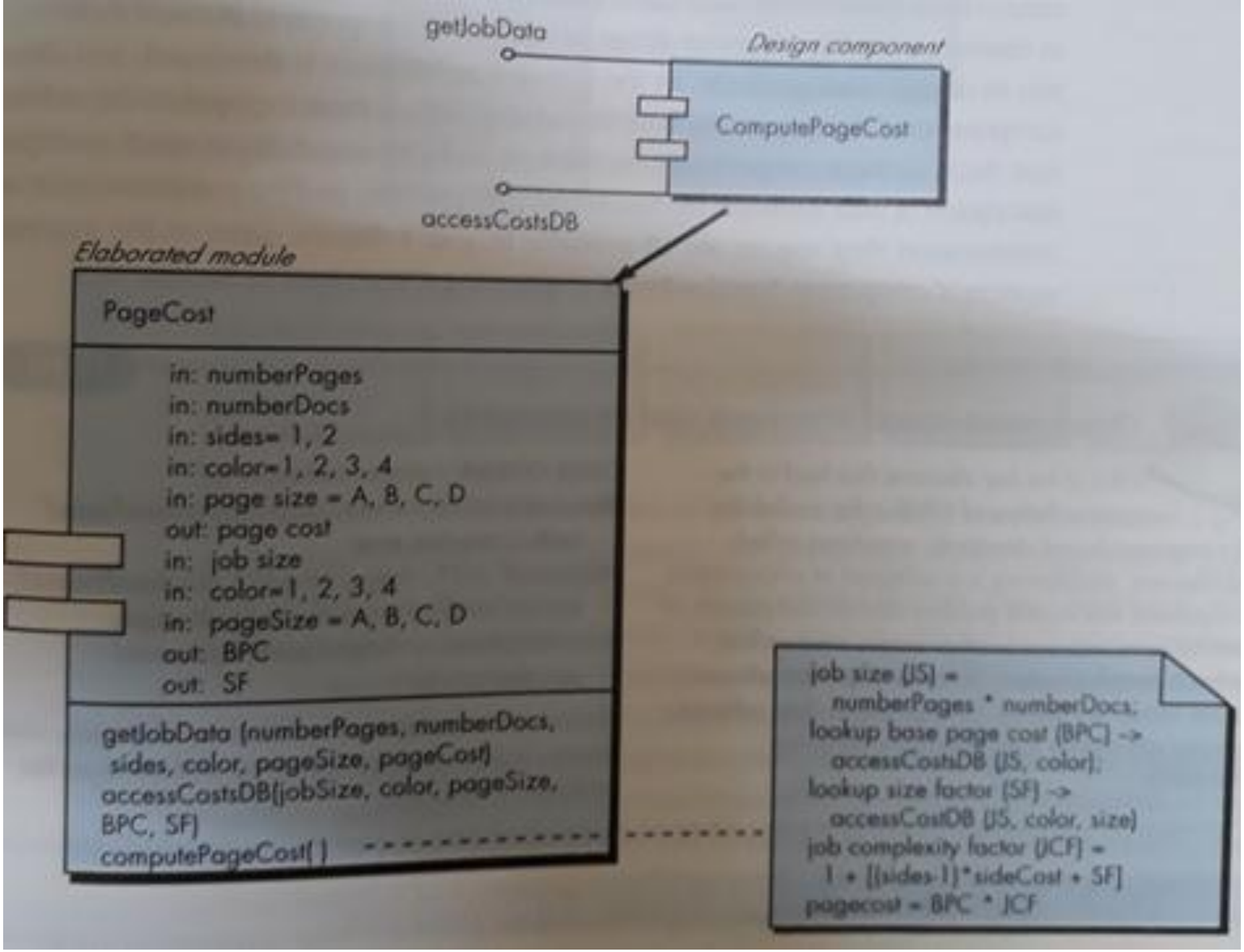

Figure 10.3 : Component-Level design for ComputePageCost

- ✓ Figure 10.3 represents the component-level design using a modified UML notation.
- ✓ The **ComputePageCost** module accesses data by invoking the module **getJobData,** which allows all relevant data to be passed to the component, & a database interface, **accessCostDB,** which enables the module to access a database that contains all printing costs.

- ✓ As design continues, the **ComputePageCost** module is elaborated to provide algorithm detail & interface detail (Figure 10.3).
- ✓ Algorithm detail can be represented using the pseudocode text shown in figure or with a UML activity diagram.

- ✓ The interfaces are represented as a collection of input & output data objects or items.
- ✓ Design elaboration continues until sufficient detail is provided to guide construction of the component.

Process-Related View

- ✓ The software community has emphasized the need to build systems that make use of existing software components or design patterns.
- ✓ In essence, a catalog of proven design or code-level components is made available to you as design work proceeds.
- ✓ As the software architecture is developed, you chose components or design patterns from the catalog & use them to populate the architecture.
- ✓ Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, & the communication & collaboration they require are all available to you.

Designing Class Based Components

Basic Design Principles

1. The Open-Closed Principle (OCP)

- ✓ *"A module *component+ should be open for extension but closed for modification".*

- ✓ You should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.

- ✓ To accomplish this, you create abstractions that serve as buffer between the functionality that is likely to be extended & the design class itself.

- ✓ For example, assume that assume that the ***SafeHome*** security makes use of a **Detector** class that must check the status of each type of security sensor.
- ✓ It is likely that as time passes, the number & types of security sensors will grow.
- ✓ If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else).
- ✓ This is a violation of OCP.

- ✓ One way to accomplish OCP for the **Detector** class is as Figure 10.4.
- ✓ The **sensor** interface presents a consistent view of sensors to the detector component.
- ✓ If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is Preserved.

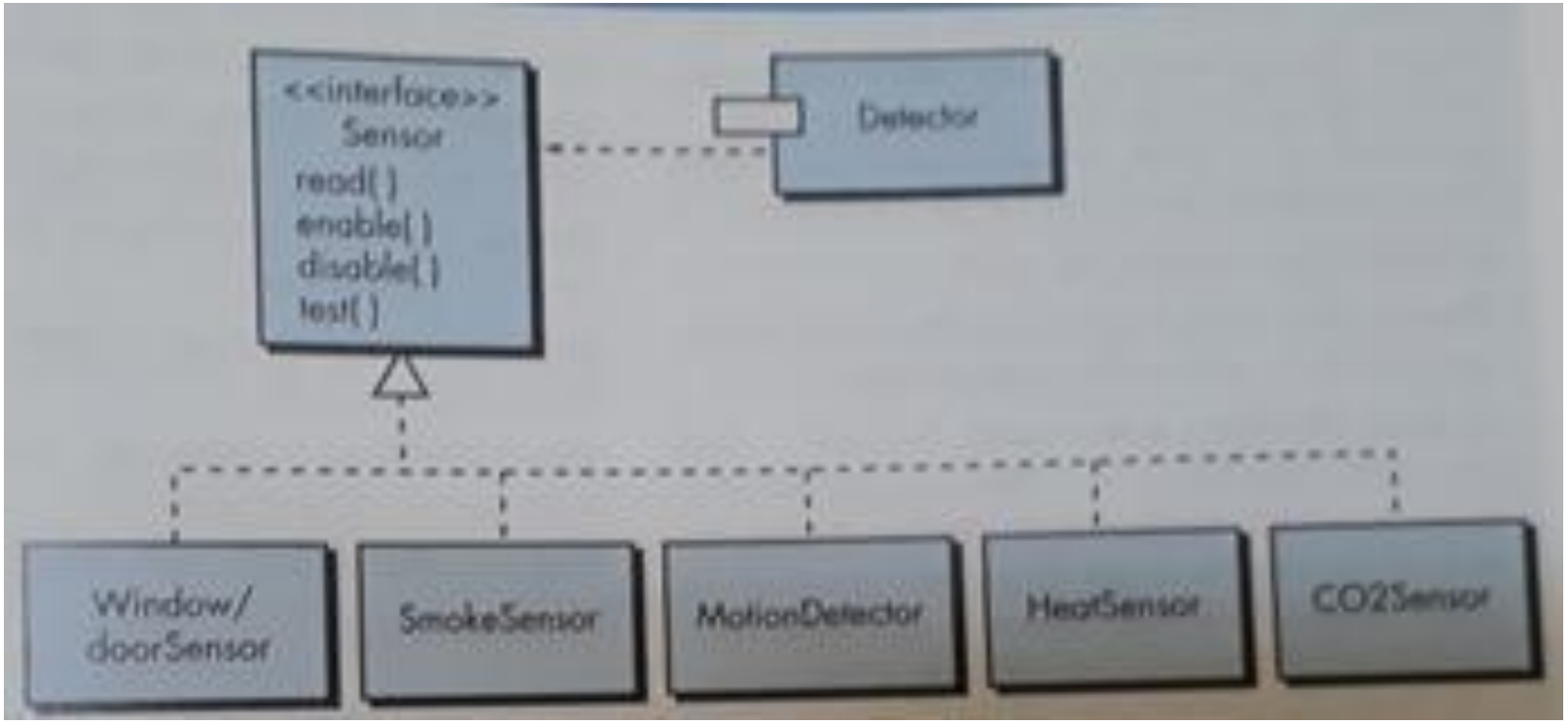

Figure 10.4 : Following the OCP

2. The Liskov Substitution Principle (LSP)

- ✓ *"Subclasses should be substitutable for their base classes".*
- ✓ A base class should continue to function properly if a class derived from the base class is passed to the component instead.
- ✓ LSP demands that any class derived from a base class must honor any implied contract between the base class & the components that use it.
- ✓ A "contract" is a *precondition* that must be true before the component uses a base class & a *postcondition* that should be true after the component uses a base class.
- ✓ When you create derived classes, be sure they conform to the pre- & postconditions.

3. Dependency Inversion Principle (DIP)

- ✓ *"Depend on abstractions. Do not depend on concretions."*
- ✓ Abstractions are the place where a design can be extended without great complication.
- ✓ The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

4. The Interface Segregation Principle (ISP)

- ✓ *"Many client-specific interfaces are better than one general purpose interface".*
- ✓ There are many instances in which multiple client components use the operations provided by a server class.
- ✓ ISP suggests that you should create a specialized interface to serve each major category of clients.
- ✓ Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.
- ✓ If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

- ✓ As an example, consider the FloorPlan class that is used for the ***SafeHome*** security & surveillance functions.

- ✓ For the security functions, ***FloorPlan*** is used only during configuration activities & uses the operations ***placeDevice()***, ***showDevice(), groupDevice() & remoteDevice()*** to place, show, group, & remove sensors from the floor plan.

- ✓ The ***SafeHome*** surveillance function uses the four operations for security, but also requires special operations to manage cameras :- ***showFOV() & showDeviceID().***
- ✓ Hence, the ISP

5. The Release Reuse Equivalency Principle (REP)

- ✓ *"The granule (small seed) of reuse is the granule of release".*
- ✓ When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity & the people who will use it.
- ✓ The developer commits to establish a release control system that supports & maintains older versions of the entity while the users slowly upgrade to the most current version.
- ✓ Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed & controlled as newer versions evolve.

6. The Common Closure Principle (CCP)

- ✓ *"Classes that change together belong together".*
- ✓ Classes should be packed cohesively, i.e. they should address the same functional or behavioral area.
- ✓ When some characteristic of that area must change, it is likely that only those classes within the package will require modification.
- ✓ This leads to more effective change control & release management.

7. The Common Reuse Principle (CRP)

- ✓ *"Classes that aren't reused together should not be grouped together".*
- ✓ When one or more classes within a package changes, the release number of the package changes.
- ✓ All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package & be tested to ensure that the new release operates without incident.
- ✓ If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

Component-Level Design Guidelines

- **Components :-** Naming conventions should be established for components that are specified as part of the architectural model & then refined & elaborated as part of the component-level model.
- ✓ Architectural component names should be drawn from the problem domain & should have meaning to all stakeholders who view the architectural model.
- ✓ For example, the class name ***FloorPlan*** is meaningful to everyone reading it regardless of technical background.

- **Interfaces :-**
- ✓ Interface provide important information about communication & collaboration.

➢ Dependencies & Inheritance :-

- ✓ For improved reliability, it is a good idea to model dependencies from left to right & inheritance from bottom (derived classes) to top (base classes).
- ✓ In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency.

Cohesion

- ✓ Cohesion implies that a component or class encapsulates only attributes & operations that are closely related to one another & to the class or component itself.

- ✓ Different types of cohesion :-

- ❖ **Functional :-** It shows primarily by operations, this level of cohesion occurs when a component performs a targeted computation & then return a result.

- ❖ **Layer :-** It shows by packages, components, & classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
- For example, the ***SafeHome*** security function requirement to male an outgoing phone call if an alarm is sensed.

- It might be possible to define a set of layered packages as shown in Figure 10.5.
- The shaded packages contain infrastructure components.
- Access is from the control panel package downward.

❖ **Communicational :-** All operations that access the same data are defined within one class.

- In general, such classes focus solely on the data in question, accessing & storing it.

✓ Classes & components that exhibit functional, layer, & communicational cohesion are relatively easy to implement, test, & maintain.

✓ You should strive to achieve these levels of cohesion whenever possible.

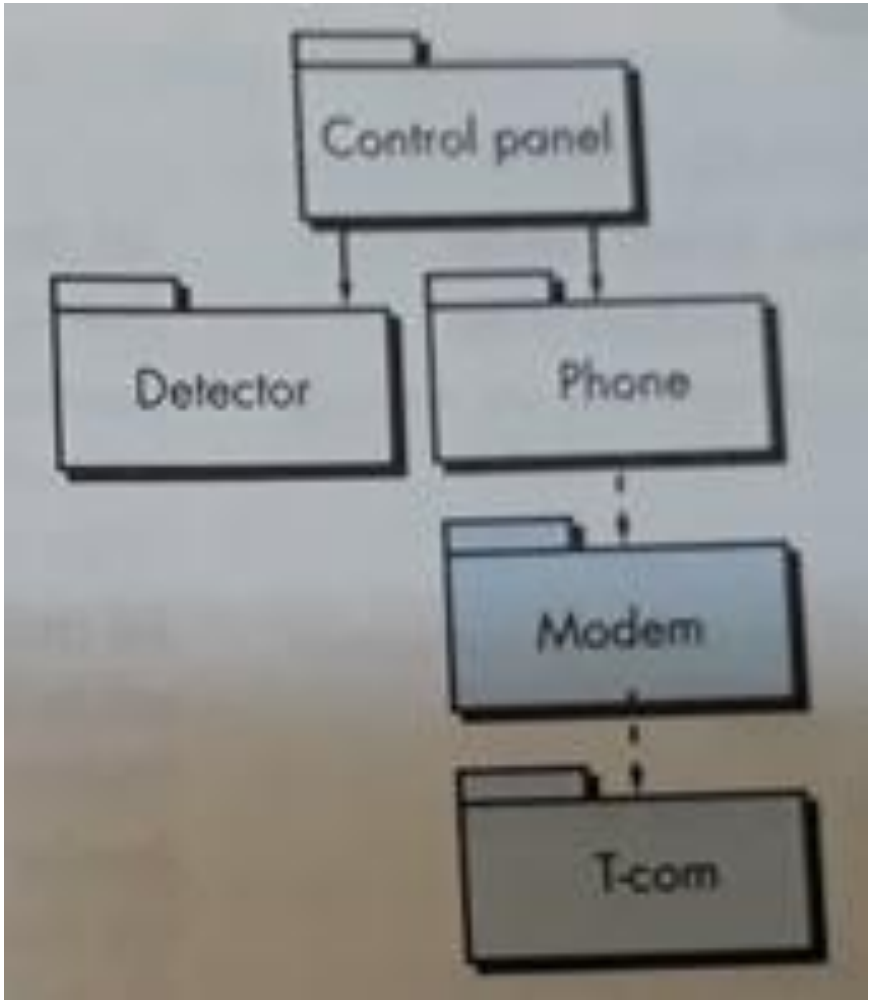

Figure 10.5 : Layer Cohesion

Coupling

- ✓ Coupling is a qualitative measure of the degree to which classes are connected to one another.
- ✓ As classes (& components) become more interdependent, coupling increases.
- ✓ Objective in component-level design is to keep Coupling as low as possible.

- ✓ Different coupling categories:-

- ❖ **Content Coupling :-** Occurs when one component modifies data that is internal to another component. This violates information hiding – a basic design concept.

- ❖ **Common Coupling :-** Occurs when a no. of components all make use of a global variable.

- **Control Coupling :-** Occurs when operation A() invokes operation B() & passes a control flag to B. The control flag then "directs" logical flow within B.
  - The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.

- **Stamp Coupling :-** Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

Conducting Component Level Design

✓ Following are the component-Level Design steps for an object oriented systems :-

1. **Identify those designs classes that corresponds to the problem domain.**

- Using the requirements & architectural model, each analysis class & architectural component should be elaborated.

2. **Identify all design classes that are corresponds to infrastructure domain.**

- These classes are not described in the requirements model & are often missing from the architecture model, but they must be described at this point.

3. Elaborate all design classes that are not acquired as reusable components.

- Elaboration requires that all interfaces, attributes, & operations necessary to implement the class be described in detail.

3.a. Specify message details when classes or components collaborate.

The requirement model makes use of a collaboration diagram to show how analysis classes collaborate with one another.

As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system.

Figure 10.6 shows a simple collaboration diagram for the printing system. 3 objects, **ProductionJob, WorkOrder, & JobQueue,** collaborate to prepare a print job for submission to the production stream.

Messages are passed between objects as shown by the arrows in the Figure 10.6.

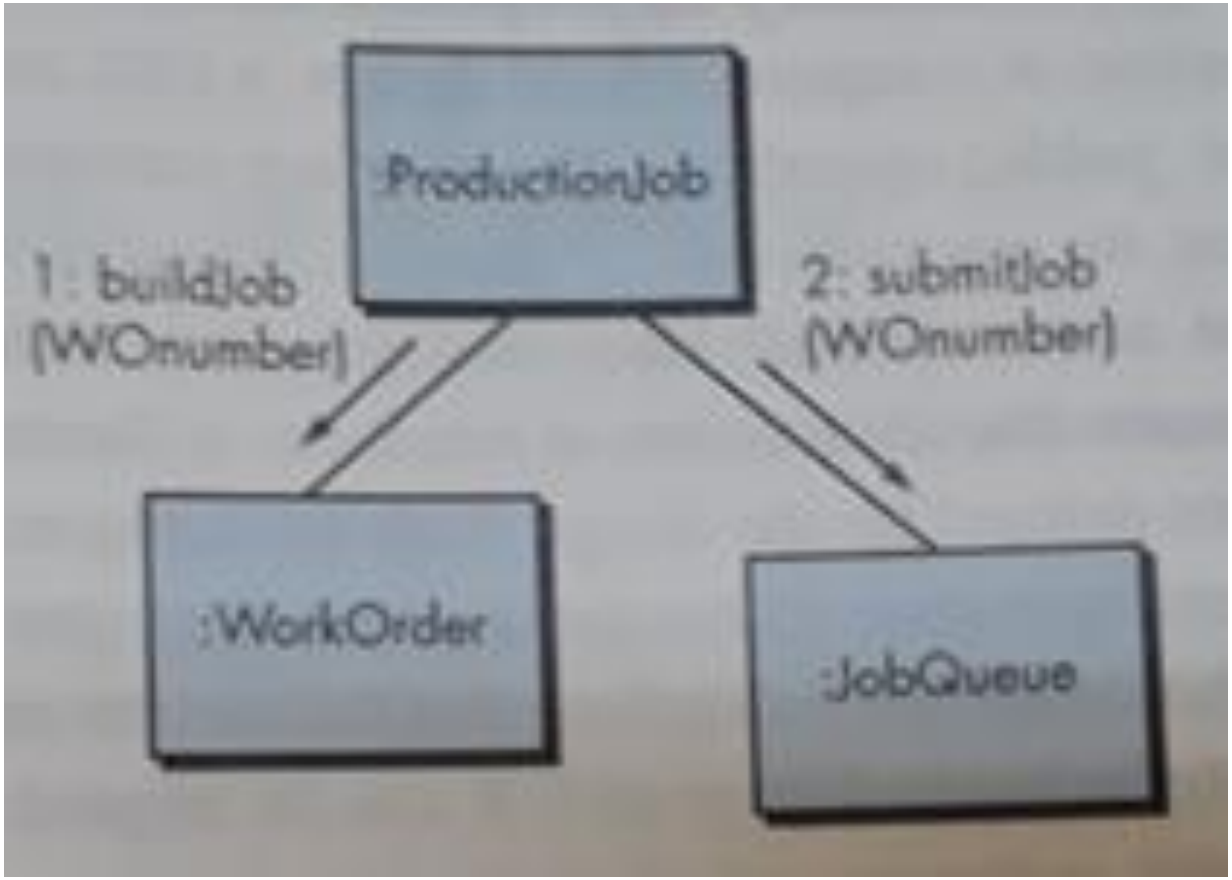

Figure 10.6 : Collaboration diagram with messaging

b.Identify appropriate interfaces for each component.

c.Elaborate attributes & define data types & data structures required to implement them.

d.Describe processing flow within each operation in detail.

This may accomplished using a programming language-based pseudo code or with a UML activity diagram.

Each software component is elaborated through a no. of iterations that apply the stepwise refinement concept.

For example, the operation **computePaperCost()** can be expanded in the following manner :-

computePaperCost (weight, size, color) : numeric

This indicates that ***computePapercost()*** requires the attributes **weight, size, & color** as input & returns a value that is numeric (actually a dollar value) as output.

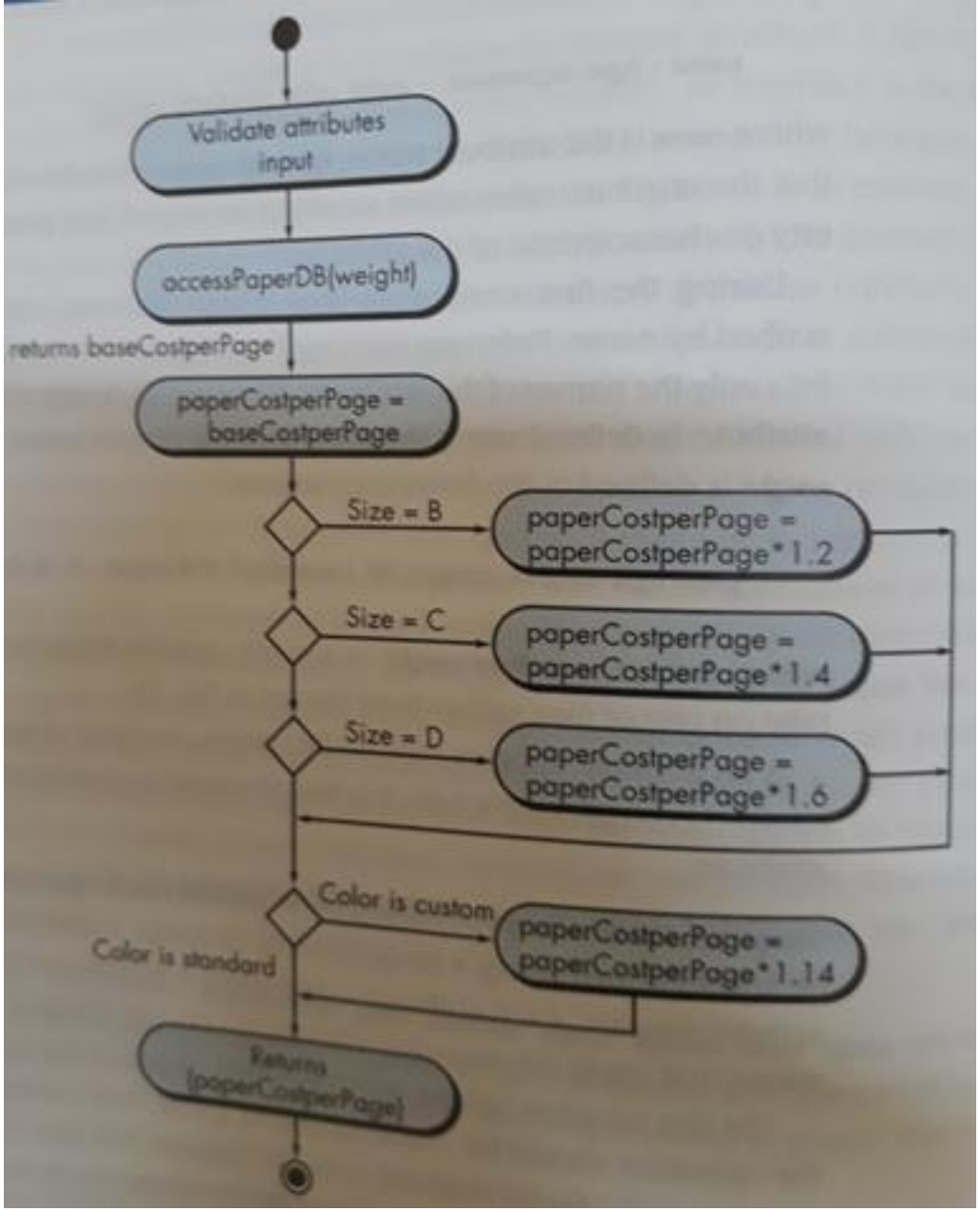

Figure 10.8 : UML activity diagram for ***computePaperCost()***

4. Describe persistent data sources (databases & files) & identify the classes required to manage them.

- In most cases, these persistent data stores are initially specified as part of architectural design.
- However, as design elaboration proceeds, it is often useful to provide additional detail about the structure & organization of these persistent data sources.

5. Develop & elaborate behavioral representations for a class or component.

- UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system.
- During component-level design, it is sometimes necessary to model the behavior of a design class.

6. Elaborate deployment diagrams to provide additional implementation detail.

- Deployment diagrams are used as part of architectural design & are represented in descriptor form.
- Deployment diagram can be elaborated to represent the location of key packages of components.
- However, components generally are not represented individually within a component diagram.

7. Refactor every component-level design representation & always consider alternatives.

User Interface Design

FIGURE 15.1
The user interface design process

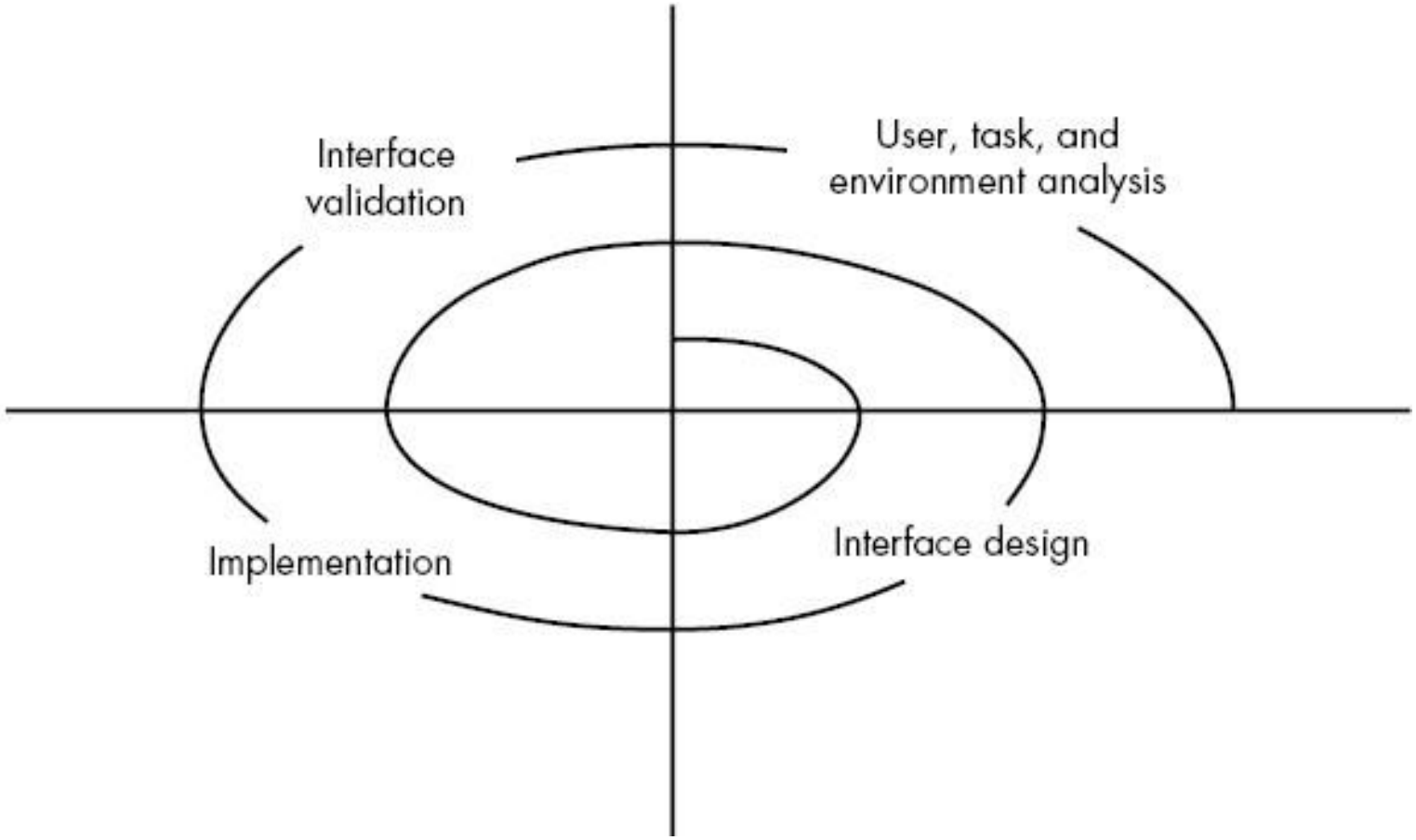

Figure 15.1 : The User Interface Design Process

- ✓ The design process for user interfaces is iterative and can be represented using a spiral model Referring to Figure 15.1, the user interface design process encompasses four distinct framework activities [MAN97]:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

1. The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined.

The information gathered as part of the analysis activity is used to create an analysis model for the interface.

2. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

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

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

Types of User Interface

- ✓ It means how user can communicate with computer systems.
- ✓ Two fundamental approaches are used to interface with the approaches are used to interface with computer system.
- ✓ Use can communicate with computer using different commands provided by the system.
- ✓ In another approach user interact by using user friendly graphical user interface. Graphical user interface provides the menu & icon based facility which can be used using keyboard & mouse.

1. Command Interpreter

- ✓ In Command Interpreter, user communicates with computer system with the help of commands.
- ✓ In Unix/Linux system the command interpreter is known as Shell.
- ✓ Shell is nothing but interface between user & OS.

- ✓ All the shells give similar facility with only minor changes.
- ✓ Selecting the particular shell totally depends on user's requirement.
- ✓ CI just accept the commands, call the corresponding program & execute it.
- ✓ When we type the 'cd' command, OS search the program for the 'cd' command & get it by executed.
- ✓ Many commands are provided by OS.
- ✓ The kind of command is depends upon the OS.

- ✓ The commands are interpreted in different manner.

- ✓ In the first approach, the command interpreter itself contains the code for the command.
- ✓ Whenever we type any command it will execute directly. The size of command interpreter depends upon the number of command supported by the command supported by the commandinterpreter.

- ✓ In the second approach, separate program is created for each & every command supported by the system.
- ✓ Whenever we need to execute any command, OSsearch the program for it, load it into memory & get it executed.

- ✓ The program for commands is stored on certain

2. Graphical User Interface (GUI)

- ✓ Instead of directly entering commands through a command-line interface, a GUI allows a mouse-based, window & menu based system as an interface.
- ✓ GUI gives the desktop environment where mouse & keyboard can be used.
- ✓ Mouse can be moved to certain location & when we click on come location, OS calls the corresponding program.

Characteristics of Good User Interface

1. Clarity
2. Concision
3. Familiarity
4. Responsiveness
5. Consistency
6. Aesthetics
7. Efficiency
8. Attractive
9. Forgiveness

Benefits of Good User Interface

- Higher Revenue.
- Increased user efficiency & satisfaction.
- Reduced development costs.
- Reduced support costs.

The Golden Rules

- ✓ Theo Mandel [MAN97] coins three “golden rules”:

1. Place the user in control.
2. Reduce the user’s memory load.
3. Make the interface consistent.

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

1. Place the User in Control

✓ Number of design principles that allow the user to maintain control:

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

2. Provide for flexible interaction ➔

- Because different users have different interaction preferences, choices should be provided. For example, software might allow auser to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands.
- But every action is not amenable to every interaction mechanism.
- Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complexshape.

3. Allow user interaction to be interruptible and undoable ➔

- Even when involved in a sequence of actions, the user should beable to interrupt the sequence to do something else (without losing the work that had been done).
- The user should also be able to “undo” anyaction.

4. Streamline interaction as skill levels advance and allow the interaction to be customized➔

- Users often find that they perform the same sequence of interactions repeatedly.
- It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

5. Hide technical internals from the casual user ➔

- The user interface should move the user into the virtual world of the application.
- The user should not be aware of the operating system, file management functions, or other arcane computing technology.
- In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

6. Design for direct interaction with objects that appear on the screen ➔

- The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the objectwere a physical thing.
- For example, an application interface that allows a user to “stretch” an object (scale it in size) is an implementation of direct manipulation.

2. Reduce the User's Memory Load

- ✓ The more a user has to remember, the more error-prone will be the interaction with the system.
- ✓ It is for this reason that a well-designed user interface does not tax the user's memory.
- ✓ Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall.
- ✓ Mandel [MAN97] defines design principles that enable an interface to reduce the user's memory load:

1. Reduce demand on short-term memory ➔

- When users are involved in complex tasks, the demand on short-term memory can be significant.
- The interface should be designed to reduce the requirement to remember past actions and results.
- This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

2. Establish meaningful defaults ➔

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

3. Define shortcuts that are intuitive ➔

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

4. The visual layout of the interface should be based on a real world metaphor ➔

- For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process.
- This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

5. Disclose information in a progressive fashion ➔

- Information about a task, an object, or some behavior should be presented first at a high level of abstraction.
- More detail should be presented after the user indicates interest with a mouse pick.
- An example, common to many word-processing applications, is the underlining function.
- The function itself is one of a number of functions under a text style menu. However, every underlining capability is not listed.
- The user must pick underlining, then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

3. Make the Interface Consistent

- ✓ Mandel [MAN97] defines a set of design principles that help make the interface consistent:

1. Allow the user to put the current task into a meaningful context ➔

- Many interfaces implement complex layers of interactions with dozens of screen images.
- It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.
- In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

2. Maintain consistency across a family of applications ➔

- A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

3. If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so ➔

- Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters.
- A change (e.g., using alt-S to invoke scaling) will cause confusion.

Interface Analysis

1. User Analysis

✓ Following information is used to accomplish this user analysis :-

1. **User Interviews :-** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture.

- This can be accomplished with one-on-one meetings or through focus groups.

2. **Sales Input :-** Sales people meet with users on a regular basis & can gather information that will help the software team to categorize users & better understand their requirements.
3. **Marketing Input :-**
4. **Support Input :-** Support staff talks with users on a daily basis. They are the most likely source of information on what works & what doesn't what users like & what they dislike.

2. Task Analysis & Modeling

- ✓ The goal of task analysis is to answer the following questions :-
- ❖ What work will the user perform in specific circumstances ?
- ❖ What tasks & subtasks will be performed as the user does the work ?
- ❖ What specific problem domain objects will the user manipulate as work is performed ?

- ✓ To answer these questions, you must applied certain techniques to the user interfaces.

1. Use Cases :-

- ✓ The use case is developed to show how an end user performs some specific work-related task.
- ✓ Use case is written in an informal style (a simple paragraph) in the First-person.

- ✓ For example, Assume that a small software company wants to build a computer aided design system explicitly for interior designers.
- ✓ To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function.
- ✓ When asked: “How do you decide where to put furniture in a room ?” an interior designer writes the following informal use case:

- ❑ accents –lamps, rugs, paintings, plants, smaller pieces, & my notes on any desires my customers has for placement.
- ❑ I then draw each item from my lists using a template that is scaled to the floor plan.
- ❑ I label each item I draw & use pencil because I always move things.
- ❑ I consider a number of alternatives placements & decide on the one I like best.
- ❑ Then, I draw a rendering (a 3-D picture) of the room to give my customer a feel for what it will look like.

- ✓ This use case provides a basic description of one important work task for computer-aided design system.
- ✓ From it, you can extract tasks, objects, & the overall flow of the interaction.

2. Task Elaboration :-

- ✓ Task analysis can be applied in 2 ways.
- ✓ An interactive, computer based system is often used to replace a manual or semi manual activity.
- ✓ To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform.
- ✓ Alternatively, you can study an existing specification for a computer based solution & derive a set of user tasks that will accommodate the user model, the design model & system perception.

- ✓ For example, using information contained in use case, furniture layout can be refined into following tasks:-

1. Draw a floor plan based on room dimensions.
2. Place windows & doors at appropriate locations.
3. (a) Use furniture templates to draw scaled furniture outlines on the floor plan. (b) Use accents template to draw scaled accents on the floor plan.
4. Move furniture outlines & accent outlines to get the best placement.
5. Label all furniture & accentoutlines.
6. Draw dimensions to show location, &
7. Draw perspective-rendering view for the customer.

- ✓ A similar approach could be used for each of the other major tasks.

Interface Design Steps

✓ Following are the interface design steps:-

1. Using information developed during interface analysis, define interface objects & actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

✓ Regardless of sequence of tasks, you should

(1) Always follow the Golden Rules.

(2) Model how the interface will be implemented.

(3) Consider the environment (ex. Display technology, OS, development tools that will be used).

Applying Interface Design Steps

- ✓ The definition of interface objects and the actions that are applied to them.
- ✓ Description of a user scenario is written.
- ✓ Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.
- ✓ Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified.
- ✓ A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon).

- ✓ The implication of this action is to create a hard-copy report.
- ✓ An application object represents application-specific data that is not directly manipulated as part of screen interaction.
- ✓ For example, a mailing list is used to store names for a mailing.
- ✓ The list itself might be sorted, merged, or purged (menu-based actions) but it is not dragged and dropped via user interaction.
- ✓ When the designer is satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed.

- ✓ Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items is conducted.
- ✓ If a real world metaphor is appropriate for the application, it is specified at this time and the layout is organized in a manner that complements the metaphor.

❖ Scenario :-

- ✓ The homeowner wishes to gain access to the SafeHome system installed in his house. Using software operating on a remote PC(e.g., a notebook computer carried by the homeowner while at work or traveling), the homeowner determines the status of the alarm system, arms or disarms the system, reconfigures security zones, and views different rooms within the house via preinstalled video cameras.

- ✓ To access SafeHome from a remote location, the homeowner provides an identifier and a password.
- ✓ These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security.
- ✓ Once validated, the user (with full access privileges) checks the status of the system and changes status by arming or disarming SafeHome.
- ✓ The user reconfigures the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required.
- ✓ The user views the interior of the house via strategically placed video cameras.
- ✓ The user can pan and zoom each camera to provide different views of the interior.

❖ Homeowner tasks :-

- *accesses* the *SafeHome* system
- *enters* an **ID** and **password** to allow remote access
- *checks* **system status**
- *arms* or *disarms SafeHome* system
- *displays* **floor plan** and **sensor locations**
- *displays* **zones** on floor plan
- *changes* **zones** on floor plan
- *displays* **video camera locations** on floor plan
- *selects* **video camera** for viewing
- *views* **video images** (4 frames per second)
- *pans* or *zooms* the **video camera**

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

Design Issues

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

- ✓ **System Response Time :-**

- *System response time* is the primary complaint for many interactive applications.
- In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

- System response time has **two important characteristics: length and variability.**
- If the length of system response is too long, user frustration and stress is the inevitable result.

- However, a very brief response time can also be detrimental if the user is being paced by the interface.
- A rapid response may force the user to rush and therefore make mistakes.

- *Variability* refers to the deviation from average response time, and in many ways, it is the most important response time characteristic.
- Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command is preferable to a response that varies from 0.1 to 2.5 seconds.
- The user is always off balance, always wondering whether something "different" has occurred behind the scenes.

✓ Help Facilities:-

- Two different types of help facilities are encountered: integrated and add-on [RUB88].
- An integrated help facility is designed into the software from the beginning.
- It is often context sensitive, enabling the user to select from those topics that are relevant to the actions currently being performed.
- Obviously, this reduces the time required for the user to obtain help and increases the "friendliness" of the interface.

- An add-on help facility is added to the software after the system has been built.
- In many ways, it is really an on-line user's manual with limited query capability.
- The user may have to search through a list of hundreds of topics to find appropriate guidance, often making many false starts and receiving much irrelevant information.

✓ Error Handling:-

- Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry.
- At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.
- There are few computer users who have not encountered an error of the form:

SEVERE SYSTEM FAILURE -- 14A

Somewhere, an explanation for error 14A must exist.

❖ **In general, every error message or warning produced by an interactive system should have the following characteristics :-**

- The message should describe the problem in jargon that the user can understand.
- The message should provide constructive advice for recovering from the error.

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

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

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

✓ Menu & Command Labeling :-

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

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?

Design Evaluation

- ✓ Once an operational user interface prototype has been created, it must be evaluated to determine whether it meets the needs of the user.

- ✓ Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.

- ✓ The user interface evaluation cycle takes the form shown in Figure 15.3. **After the design model has been completed, a first-level prototype is created.**

- ✓ The prototype is evaluated by the user, who provides the designer with direct comments about the efficacy (effect) of the interface.

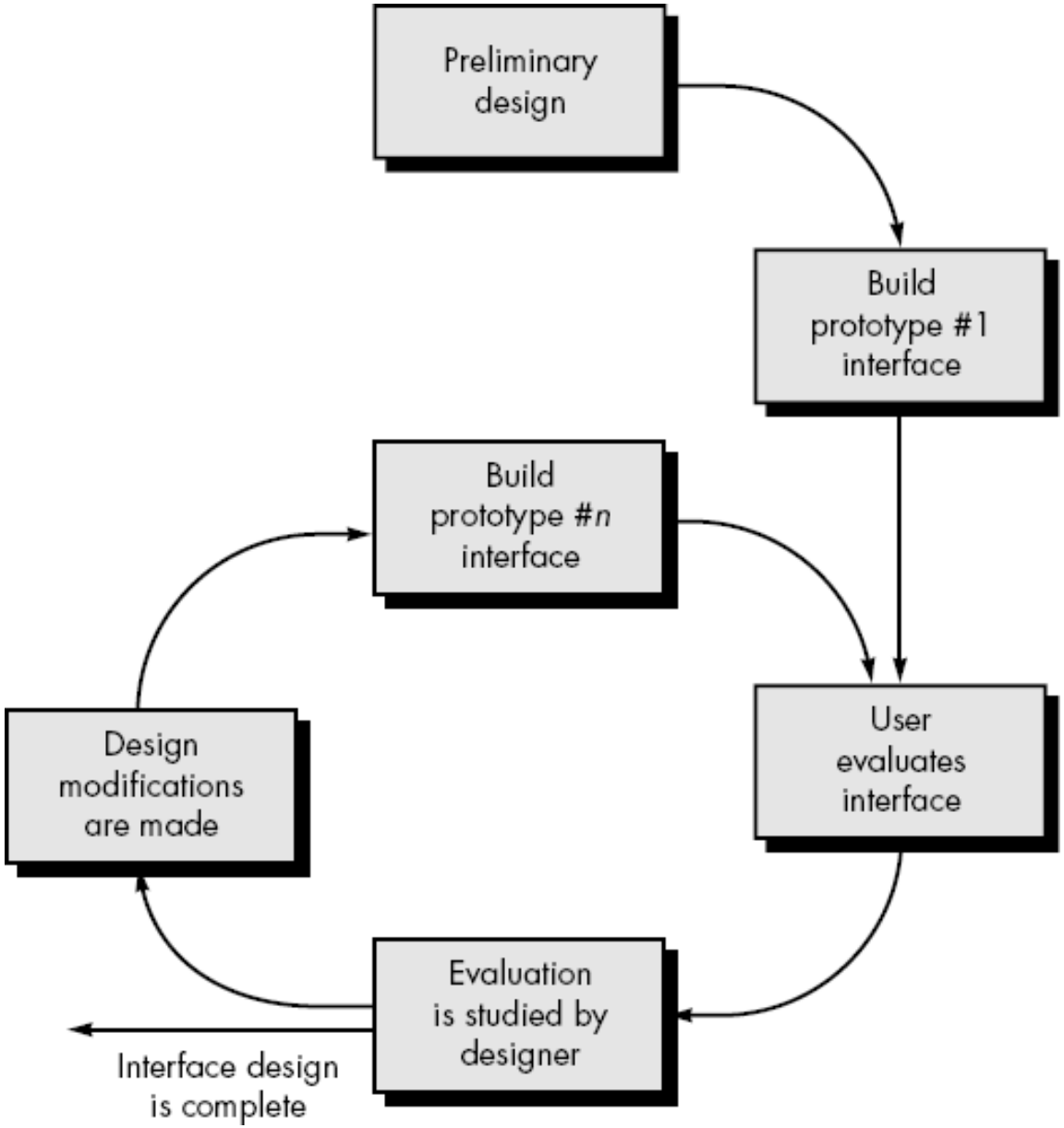

Figure 15.3 : The interface design evaluation cycle

- ✓ In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data(e.g., 80 percent of all users did not like the mechanism for saving data files).

- ✓ Design modifications are made based on user input and the next level prototype is created.

- ✓ The evaluation cycle continues until no further modifications to the interface design are necessary.

- ✓ If a design model of the interface has been created, a number of evaluation criteria [MOR81] can be applied during early design reviews ➔

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

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

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

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

- ✓ Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface.

- ✓ Tocollect qualitative data, questionnaires can be distributed to users of the prototype.

- ✓ Questions can be all (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, or (4) percentage (subjective) response.

Case Study :-
WebApp Interface Design

✓ Interface design principles & guidelines :-

- ✓ The user interface of a WebApp is its “first impression”.
- ✓ Regardless of the value of its content, the sophistication of its processing capabilities & services, & the overall benefit of the WebApp itself, a poorly designed interface will disappoint the potential user & may, in fact, cause the user to go elsewhere.

- ✓ There are some design principles :-

1. Anticipation

- ➢ ***“A WebApp should be design so that it anticipates the user’s next move.”***
- ➢ Ex. Consider a customer support WebApp developed by a manufacturer of computer printers.
- ➢ A user has requested a content object that presents information about a printer driver for a newly released OS.

- The designer of the WebApp should anticipate that the user might request a download of the driver & should provide navigation facilities that allow this to happen without requiring the user to search for this capability.

2. Communication

- ***"The interface should communicate the status of any activity initiated by the user."***
- The interface should also communicate user status (e.g. the user's identification) & her location within the WebApp content hierarchy.

3. Consistency

- ***"The use of navigation controls, menus, icons, & aesthetics (e.g. color, shape, layout) should be consistent throughout the WebApp."***
- Ex. If underlined blue text implies a navigation link, content should never incorporate blue underlined text that does not imply a link.

4. Controlled Autonomy

- ***"The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application."***

- Ex. Navigation to secure portions of the WebApp should be controlled by UserID & Password, & there should be no navigation mechanism that enables a user to circumvent (dissipate) these controls.

5. Efficiency

- ***"The design of the WebApp & its interface should optimize the user's work efficiency, not the efficiency of the developer who designs & builds it or the client-server environment that executes it."***

6. Flexibility

- ***"The interface should be flexible enough to enable some users to accomplish tasks directly & others to explore the WebApp in a somewhat random fashion."***

7. Focus

- ***"The WebApp interface (& the content it presents) should stay focused on the user tasks at hand."***
- The problem is that the user can rapidly become lost in many layers of supporting information & lose sight of the original content that he wanted in the first place.

8. Human Interface Objects

- ***"Avast library of reusable human interface objects has been developed for WebApps. Use them."***
- Any interface object that can be seen, heard, touched, or otherwise perceived by an end user can be acquired from any one of anumber of object libraries.

9. Learnability

- ***"A WebApp interface should be designed to minimize learning time, & once learned, to minimize relearning required when the WebApp is revisited."***
- Interface should be simple.

10. Readability

- ***"All information presented through the interface should readable by young & old."***
- The interface designer should emphasize readable type styles, style font, sizes, & color background choices that enhance contrast.