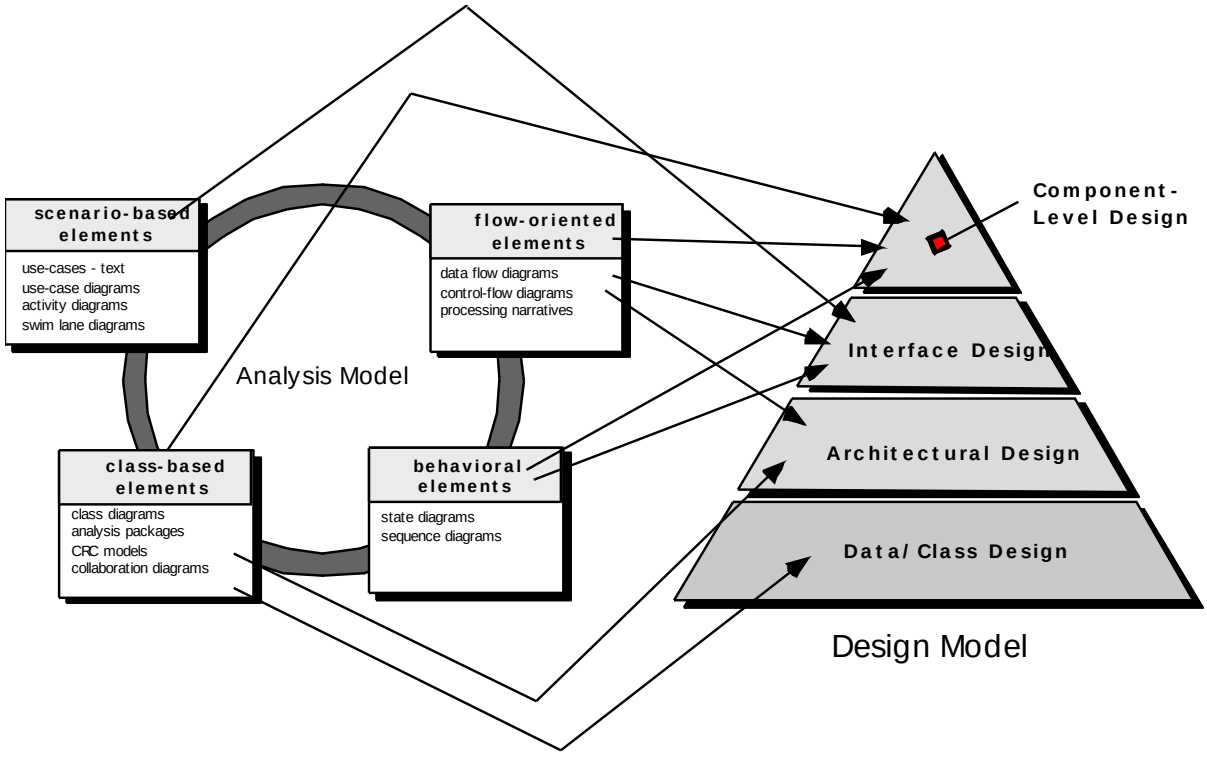
DESIGN ENGINEERING

- The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence.
- Diversification and Convergence - the qualities which demand intuition and judgment are based on experience.
    - Principles and heuristics that guide the way the model is evolved.
    - Set of criteria that enables quality to be judge.
    - Process of iteration that ultimately leads to final design representation.
- Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve. Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines.

From Analysis Model To Design Model:

- Each element of the analysis model provides information that is necessary to create the four design models
    - The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
    - The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
    - The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
    - The component-level design transforms structural elements of the software architecture into a procedural description of software components



DESIGN PROCESS AND DESIGN QUALITY

- *The software design is an iterative process through which requirements are translated into a **blueprint** for constructing the software.*
- Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design. Three characteristics that serve as a guide for the evaluation of a good design.
    - The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
    - The design must be a readable, understandable guide for those how generate code and for those who test and subsequently support the software.
    - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains form an implementation perspective.

■ Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

- A design should exhibit an architecture that
  (a) has been created using recognizable architectural styles or patterns. (b) is composed of components that exhibit good design characteristics. (c) can be implemented in an evolutionary fashion, thereby facilitating
  implementation and testing.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

■Quality Attributes

Hewlett Packard developed set of software quality attributes name 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 measuring by processing speed, response time, resource consumption, throughput and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability-these three attributes represent amore common term, maintainability-in addition, testability, compatibility, and configurability.

DESIGN CONCEPTS

Fundamental software design concepts provide the necessary frame work for "getting it right."

- **Abstraction** - Many levels of abstraction can be posed.
  - The highest level states the solution in broad terms using the language of the problem environment.
  - The lower level gives more detailed description of the solution.
  - *Procedural Abstraction* refers to a sequence of instruction that have specific and limited functions.
  - *Data Abstraction* is a named collection of data that describes a data object.

- **Architecture** – provides overall structure of the software and the ways in which the structure provides conceptual integrity for a system. The goal of software design is to derive and architectural rendering of a system which serves as a framework from which more detailed design activities can be conducted.
  - The Architectural design is represented by the following models
    - *Structural Model* – Organized collection of program components.
    - *Framework Model* – Increases level of design abstraction by identifying repeatable architectural design frameworks that are encountered in similar types of applications.
    - *Dynamic Model* – Addresses the behavioral aspects of the program architecture.
    - *Process Model* – Focuses on design of business or technical process that system must accommodate.
    - *Functional Model* – Used to represent functional hierarchy of the system.

- **Patterns** – A pattern is a named nugget of insight which conveys the essences of proven solutions to a recurring problem with a certain context amidst competing concerns. The design pattern provides a description that enables a designer to determine
  - Whether the pattern is applicable to the current work?
  - Whether the pattern can be reused?
  - Whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern?

- **Modularity** – the software is divided into separately named and addressable components called modules that are integrated to satisfy problem requirements.

- **Information Hiding** – modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Information hiding provides greatest benefits when modifications are required during testing and software maintenance.

- **Functional Independence** – is achieved by developing modules with single minded function and an aversion to excessive interaction with other modules.
  Independence is assessed using two qualitative criteria
  - ***Cohesion*** – is a natural extension of information hiding concept, module performs single task, requiring little interaction with other components in other parts of a program.
  - ***Coupling*** – interconnection among modules and a software structures, depends on interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

- **Refinement** – is a process of elaboration. Refinement causes the designer to elaborate on original statement, providing more and more detailed as each successive refinement occurs.

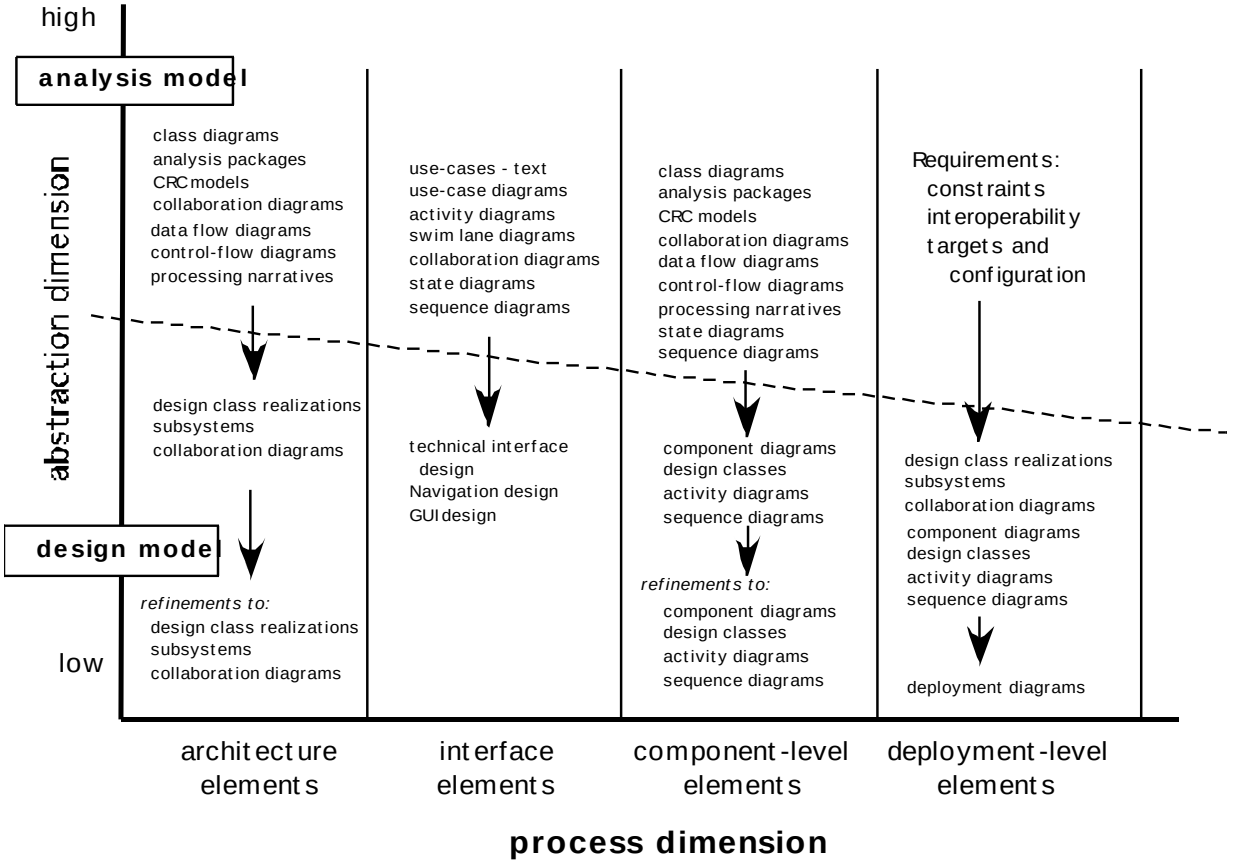
- **Refactoring** – important design activity suggested for agile methods, refactoring is a recognization technique that simplifies design of a component without changing its function or behavior. "Refactoring is a process of changing a software system in such away that it does not alter the external behavior of the code yet improves its internal structure.

- **Design Classes** – describes some element of problem domain, focusing on aspects of the problem that are user or customer visible. The software team must define a set of design classes that
  (1) Refine the analysis classes by providing design detail that will enable the classes to be implemented and
  (2) Create a new set of design classes that implement a software infrastructure to support the business solution. Five different types of design classes each representing a different layer of the design architecture is suggested
    - *User Interface classes* define all abstractions that are necessary for Human Computer Interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine) and the design classes for the interface may be visual representations of the elements of the metaphor.
    - *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
    - *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
    - *Persistent classes* represent data stores (e.g, a database) that will persist beyond the execution of the software.

- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

THE DESIGN MODEL

- The design model can be viewed in two in two different dimensions. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.



■ DATA DESIGN ELEMENTS

- Like other software engineering activities, data design creates a model of data and/or information that is represented at a high level of abstraction.
- At the program component level, the design of data structures and 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 and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

ARCHITECTURAL DESIGN ELEMENTS

- The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms, their size, shape, and relationship to one another, and the doors and windows that allow movement into and 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 three sources:

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

INTERFACE DESIGN ELEMENTS

- The interface design for software is the equivalent to a set of detailed drawing for the doors, windows and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utilities connections (e.g., water, electrical, gas, and telephone) come into the house and are distributed among the rooms depicted in the floor plan.

- There are three 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; and
  (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

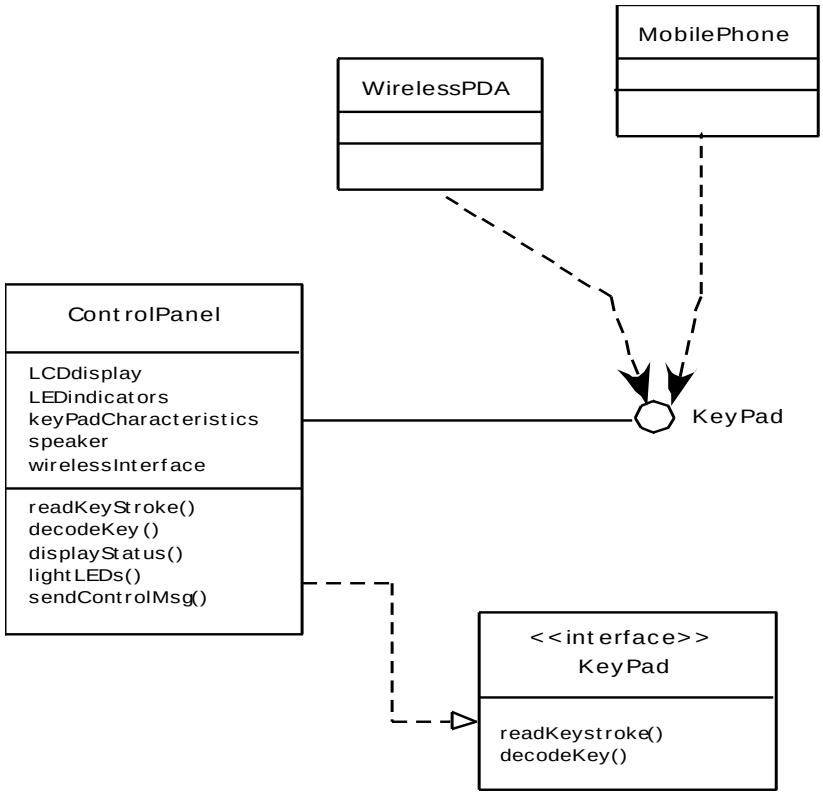
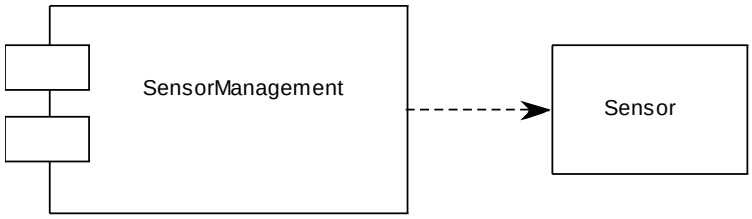


Figure 9.6 UML interface representation for **ControlPanel**

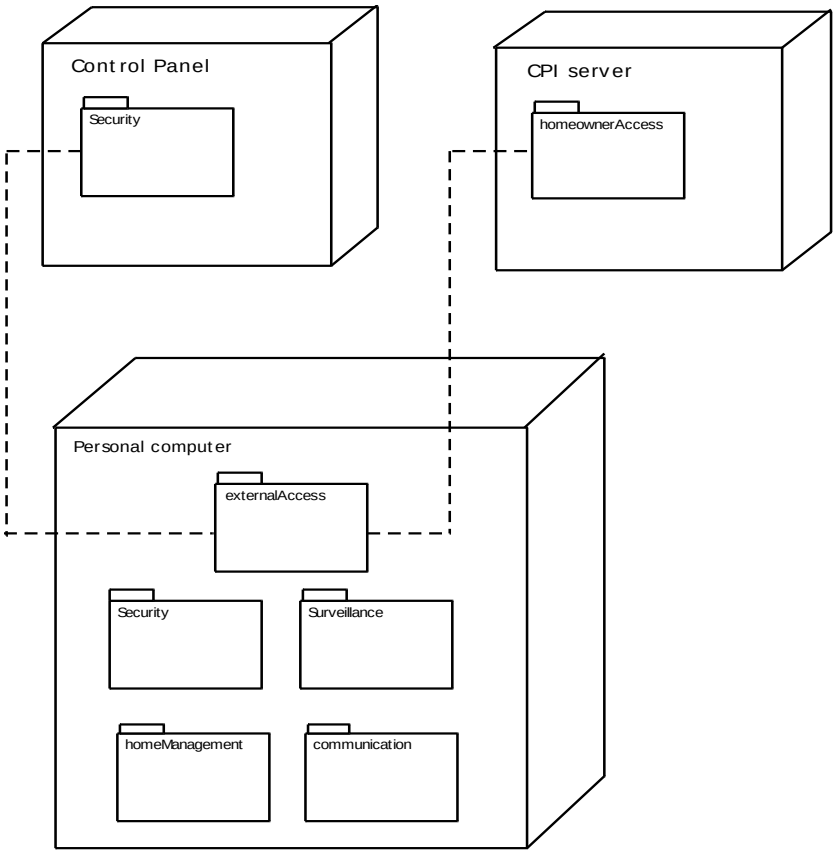
- COMPONENT-LEVEL DESIGN ELEMENTS

  - The component-level design for software is equivalent to a set of detailed drawings for each room in a house. These drawing depict wiring and plumbing within each room, the location of electrical receptacles and switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component.



- DEPLOYMENT-LEVEL DESIGN ELEMENTS

  - Deployment level design elements indicates how software functionality and subsystems will be allocates within the physical computing environment that will support the software.



Pattern Based Software Design:

- Mature engineering disciplines make use of thousands of design patterns for such things as buildings, highways, electrical circuits, factories, weapon systems, vehicles, and computers
- Design patterns also serve a purpose in software engineering
- Architectural patterns
  - Define the overall structure of software
  - Indicate the relationships among subsystems and software components
  - Define the rules for specifying relationships among software elements
- Design patterns
  - Address a specific element of the design such as an aggregation of components or solve some design problem, relationships among components, or the mechanisms for effecting inter-component communication
  - Consist of creational, structural, and behavioral patterns
- Coding patterns
  - Describe language-specific patterns that implement an algorithmic or data structure element of a component, a specific interface protocol, or a mechanism for communication among components

CREATING AN ARCHITECTURAL DESIGN

- Design is an activity concerned with making major decisions, often of s structural nature.
- It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes.
- Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.
    - What is Architectural design?
    - Who does it?
    - Why is it important?
    - What are the steps?
    - What is the work product?
    - How do I ensure that I have done it right?

SOFTWARE ARCHITECTURE

- Effective software architecture and its explicit representation and design have become dominant themes in software engineering.

■ **Why Architecture?**

- The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:
  (1) analyze the effectiveness of the design in meeting its stated requirements,
  (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and
  (3) reduce the risks associated with the construction of the software.

■ **Why is Architecture Important?**

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

DATA DESIGN

The data design action translates data objects defined as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

■ **At the architectural level …**
- Design of one or more databases to support the application architecture
- Design of methods for 'mining' the content of multiple databases
  - navigate through existing databases in an attempt to extract appropriate business-level information
  - Design of a data warehouse—a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business

■ **At the component level …**
- refine data objects and develop a set of data abstractions
- implement data object attributes as one or more data structures
- review data structures to ensure that appropriate relationships have been established
- simplify data structures as required
  - The systematic analysis principles applied to function and behavior should also be applied to data.

- All data structures and the operations to be performed on each should be identified.
- A data dictionary should be established and used to define both data and program design.
- Low level data design decisions should be deferred until late in the design process.
- The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
- A library of useful data structures and the operations that may be applied to them should be developed.
- A software design and programming language should support the specification and realization of abstract data types.

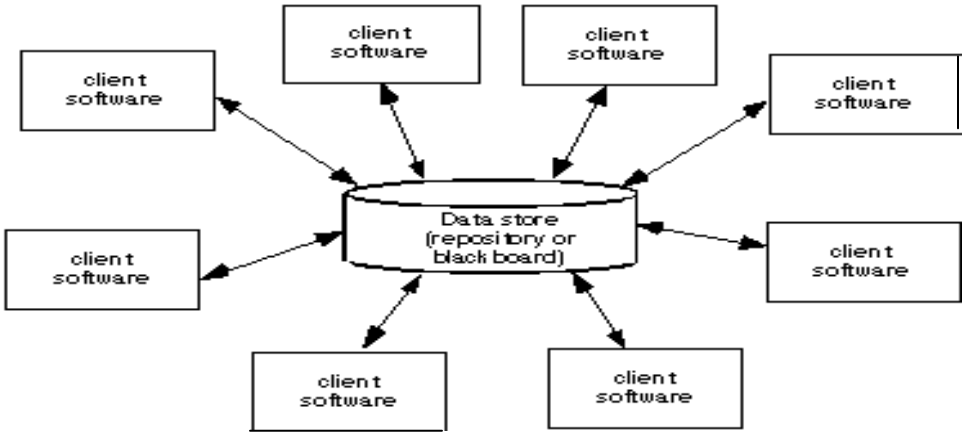
ARCHITECTURAL STYLES

- Each style describes a system category that encompasses:
  (1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
  (2) a set of connectors that enable "communication, coordination and cooperation" among components,
  (3) constraints that define how components can be integrated to form the system, and
  (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

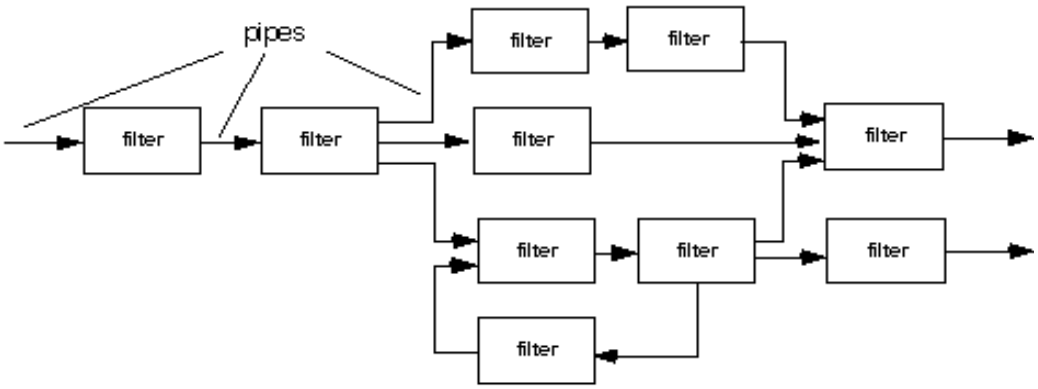
■ **Data-Centered Architecture**

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. The following figure illustrates a typical data-centered style.

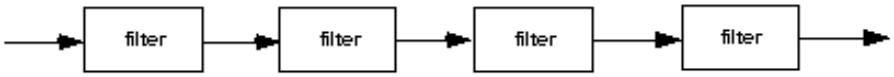


■ Data Flow Architecture

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter structure has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential.



(a) pipes and filters



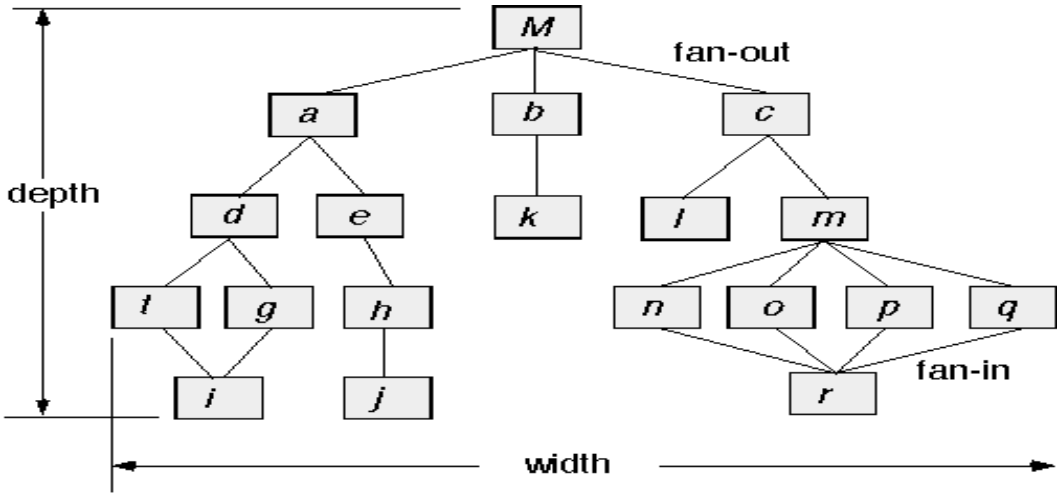
(b) batch sequential

■ Call and Return Architecture

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. Two substyles exist within this category:

    - *Main program/subprogram architecture.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. The following figure illustrates architecture of this type.

    - *Remote procedure call architecture.* The components of main program /subprogram architecture are distributed across multiple computers on a network.

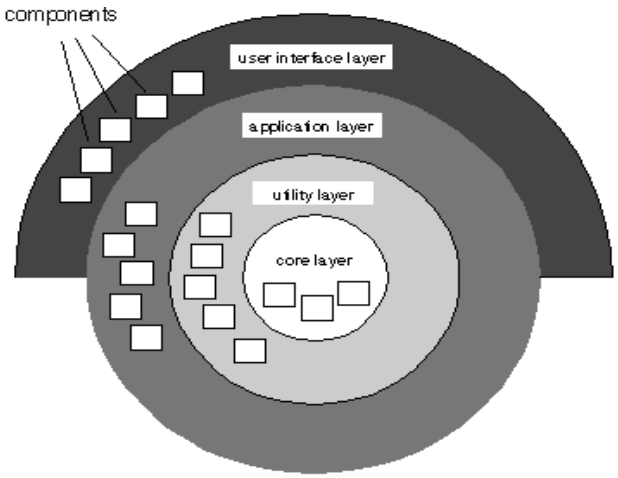


■ **Object-oriented architecture**

- The component of a system encapsulates data and the operations that must be applied to manipulate the data communication and coordination between components is accomplished via message passing.

■ **Layered Architecture**

- The basic structure of a layered architecture is illustrated in the following figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



ARCHITECTURAL PATTERNS

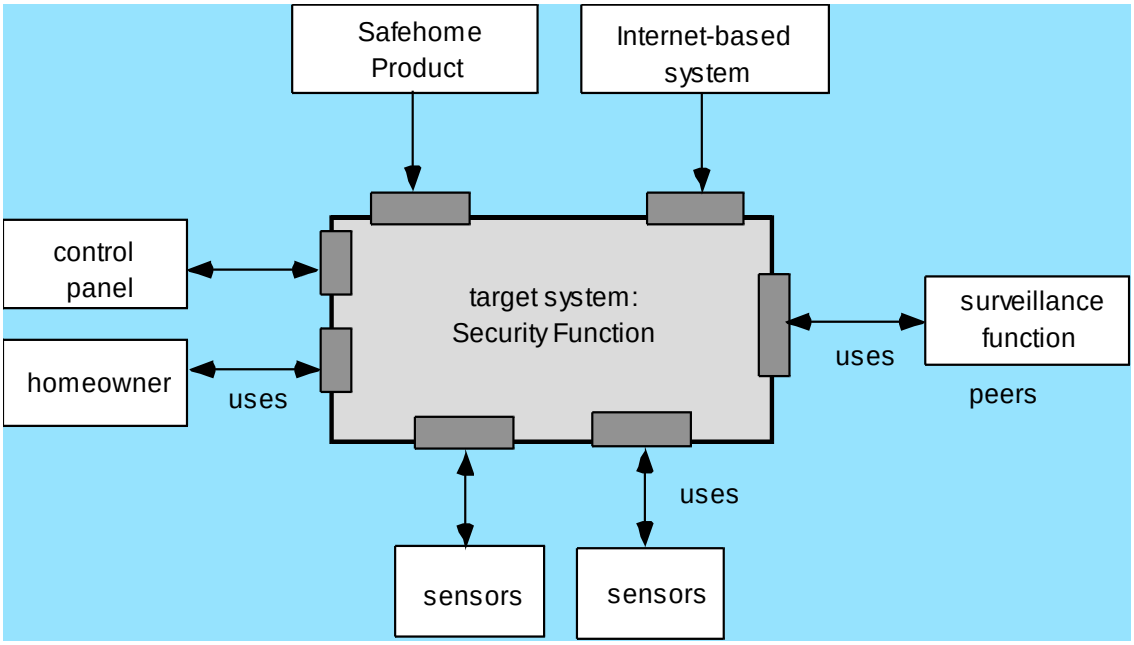
- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
    - *operating system process management* pattern
    - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
    - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
    - an *application level persistence* pattern that builds persistence features into the application architecture
- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
    - A *broker* acts as a 'middle-man' between the client component and a server component.

ARCHITECTURAL DESIGN

- The software must be placed into context
    - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
    - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

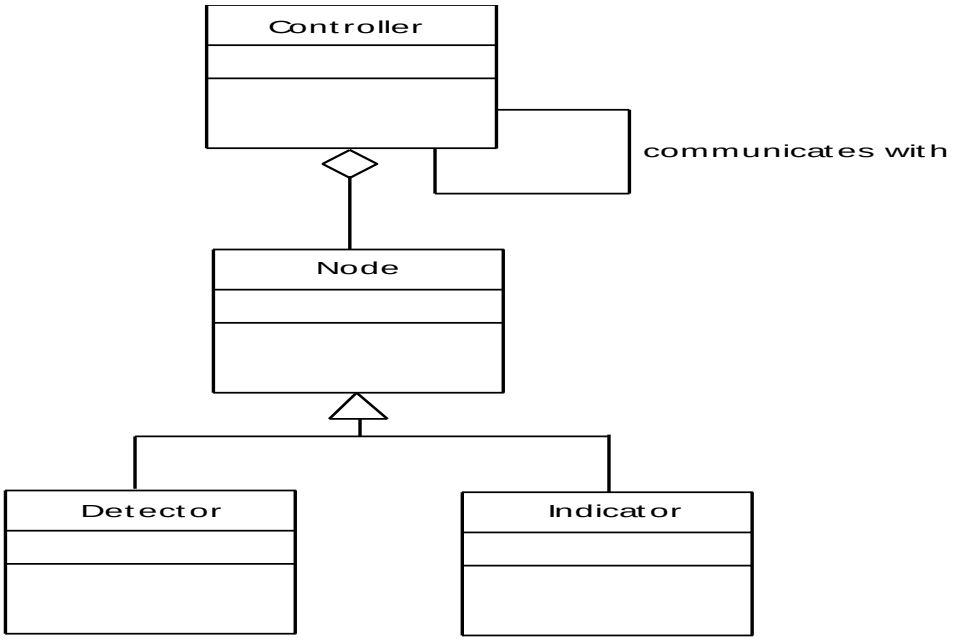
■ Architectural Context

- A system engineer must model context. A system context diagram accomplishes this requirement by representing the flow of information into and out of the system, the user interface and relevant support processing. At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure.



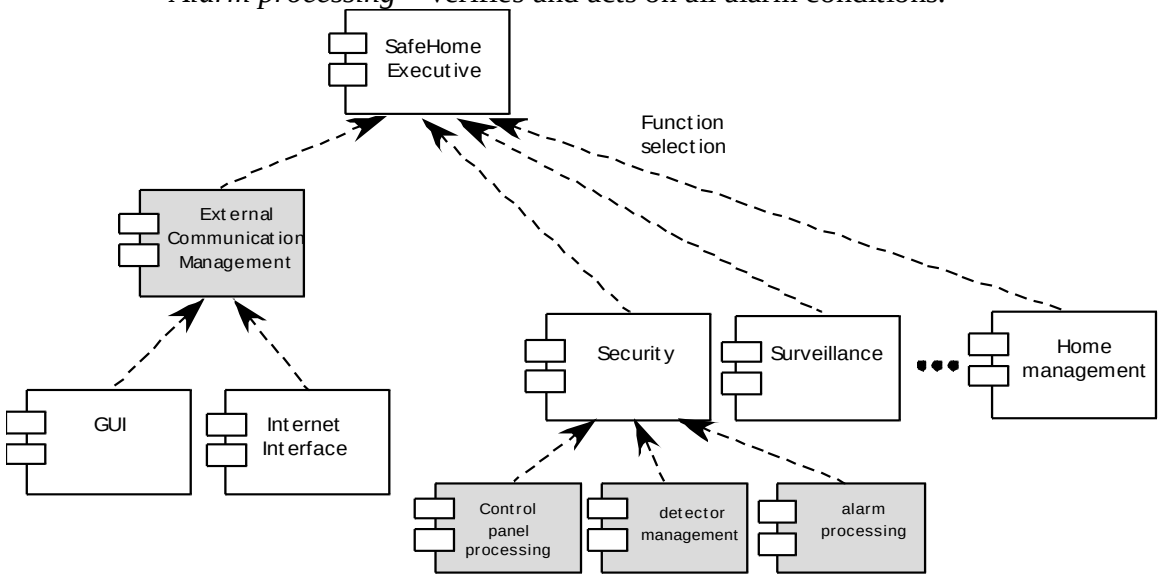
■ Archetypes

- the *SafeHome* home security function, we might define the following archetypes:
    - Node – Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of
      (1) various sensors and
      (2) a variety of alarm (output) indicators.
    - Dectector – An abstraction that encompasses all sensing equipment that feeds information into the target system.
    - Indicator – An abstraction that represents all mechanism (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
    - Controller – An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.



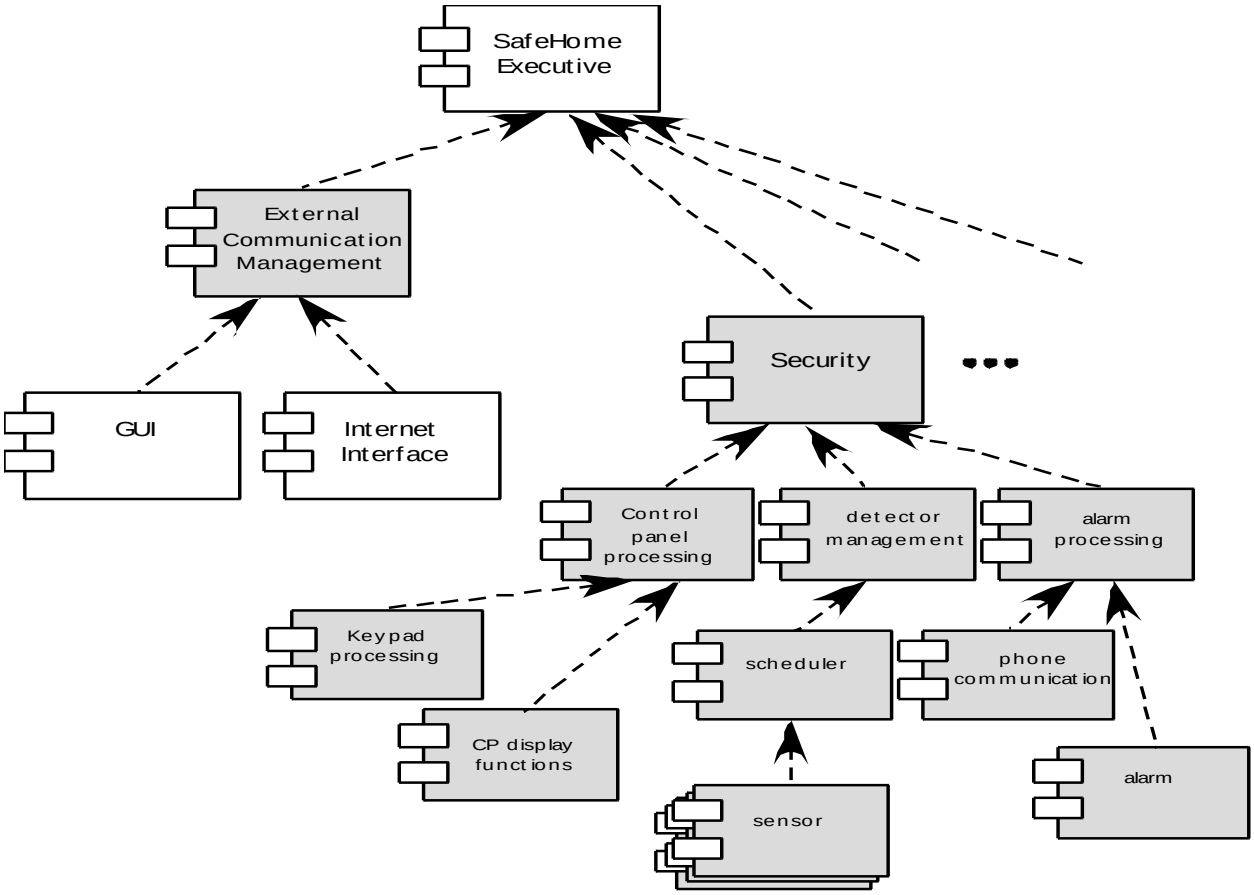
■ Refining the Architecture into Components

- Continuing the *SafeHome* home security function example, we might define the set of top-level components that address the following functionality.
  - *External communication management* – coordinates communication of the security function with external entities, for example, internet-based systems. External alarm notification.
  - *Control panel processing* – manages all control panel functionality.
  - *Detector management* – coordinates access to all detectors attached to the system.
  - *Alarm processing* – verifies and acts on all alarm conditions.



■ Describing Instantiations of the System

- The instantiation of the architecture means that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
    - Fig Illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in figure are refined further to show additional detail.



Modeling Component-Level Design

Introduction:

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
  - The translation can open the door to subtle errors that are difficult to find and correct later
  - "Effective programmers should not waste their time debugging – they should not introduce bugs to start with." Edsgar Dijkstra
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

Definition:

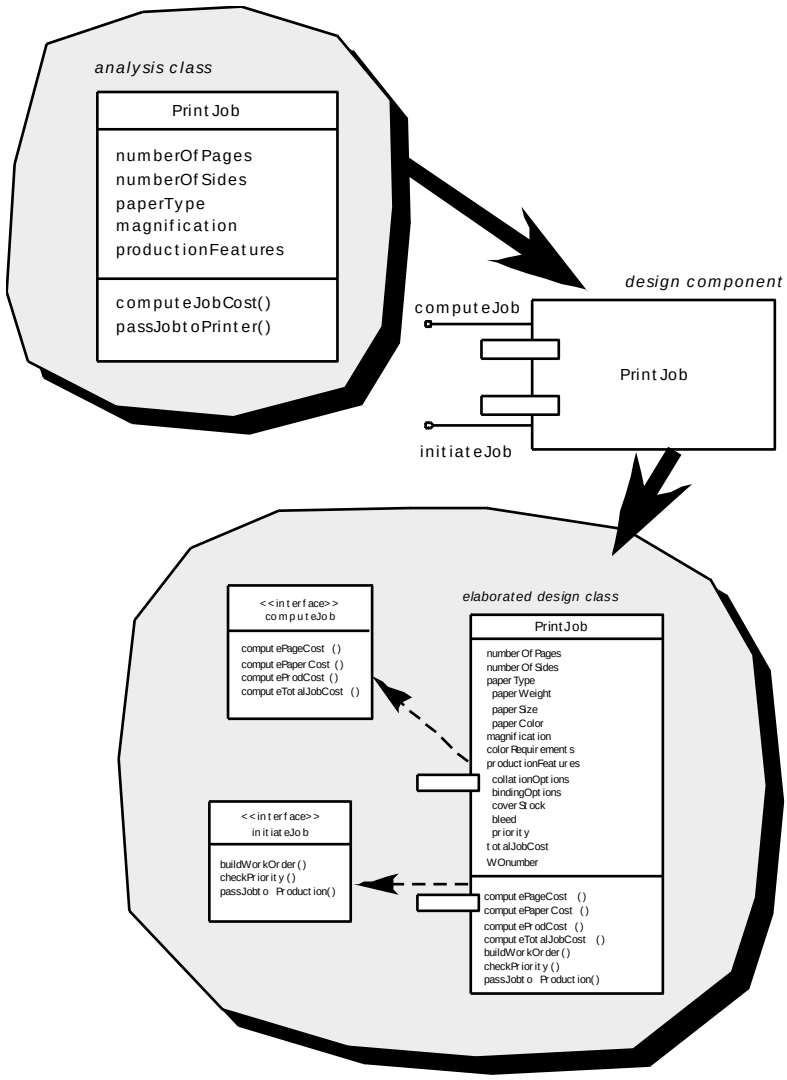
- A software component is a modular building block for computer software
  - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
  - Other components
  - Entities outside the boundaries of the system
- Three different views of a component
  - An object-oriented view
  - A conventional view
  - A process-related view

Object-oriented view

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
  - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
  - Provide further elaboration of each attribute, operation, and interface
  - Specify the data structure appropriate for each attribute
  - Design the algorithmic detail required to implement the processing logic associated with each operation

Design the mechanisms required to implement the interface to include the messaging that occurs between objects.

Component in Object-Oriented View:

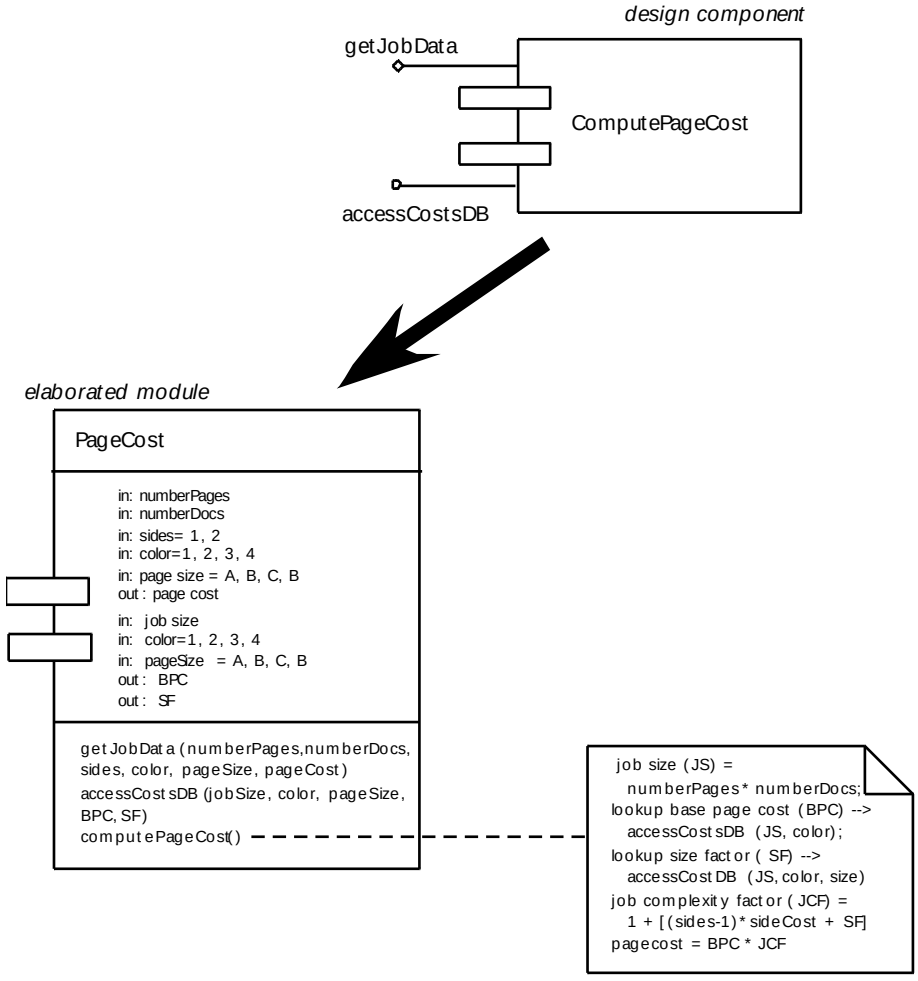


Conventional View:

- A component is viewed as a <u>functional</u> element (i.e., a module) of a program that incorporates
  - The <u>processing logic</u>
  - The internal <u>data structures</u> that are required to implement the processing logic
  - An <u>interface</u> that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
  - A <u>control</u> component that coordinates the invocation of all other problem domain components
  - A <u>problem domain</u> component that implements a complete or partial function that is required by the customer
  - An <u>infrastructure</u> component that is responsible for functions that support the processing required in the problem domain

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
  - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
  - Control components reside near the top
  - Problem domain components and infrastructure components migrate toward the bottom
  - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
  1. Define the interface for the transform (the order, number and types of the parameters)
  2. Define the data structures used internally by the transform
  3. Design the algorithm used by the transform (using a stepwise refinement approach)
-

Conventional Component:



Process-related View:

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
  - A complete description of their interface
  - The functions they perform
  - The communication and collaboration they require

Designing Class-Based Components

Component-level Design Principles:

- The Open-Closed Principle (OCP).
  - "A module [component] should be open for extension but closed for modification.
  - Extensible within the functional domain it addresses without the need to make internal (code or logic-level) modifications.
  - There shall be an abstraction between the design class and the functionality that is likely to be extended.
- Example:
  - A Detector class that must check the status of each type of security sensor.
  - Instead of an if-then-else, we put a "sensor interface" as a consistent view of sensors to Detector
  - A new type of sensor can be added with no change to Detector class

- The Liskov Substitution Principle (LSP).

"Subclasses should be substitutable for their base classes.

- Any derived class must honor any contract between the base class and the components that use it

  Contract:
- Precondition that must be true before a component uses a base class
- Post-condition that should be true after the component uses a base class

- 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
  - Depend on abstractions (like interfaces) rather than concrete components

- The Interface Segregation (isolation) Principle (ISP). "Many client-specific interfaces are better than one general purpose interface.
    - Specialized interfaces for each major category of clients
    - Interface for a client: only operations useful for that client
- Operations used by many, shall be specified in each interface

Component Packaging Principles

- Release reuse equivalency principle
    - The granularity of reuse is the granularity of release
    - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle
    - Classes that change together belong together
    - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle
    - Classes that aren't reused together should not be grouped together
    - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Component-Level Design Guidelines

- Components
    - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
    - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
    - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
    - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
    - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion:

- Cohesion is the "single-mindedness' of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible

- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - Functional
    - A module performs one and only one computation and then returns a result
  - Layer
    - A higher layer component accesses the services of a lower layer component
  - Communicational
    - All operations that access the same data are defined within one class
  - Sequential
    - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
  - Procedural
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
  - Temporal
    - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
  - Utility
    - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling:

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
  - Data coupling
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
  - Stamp coupling
    - A whole data structure or class instantiation is passed as a parameter to an operation
  - Control coupling
    - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
    - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result

- Common coupling
  - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
- Content coupling
  - One component secretly modifies data that is stored internally in another component

➢ **Other kinds of coupling (unranked)**

- Subroutine call coupling
  - When one operation is invoked it invokes another operation within side of it
- Type use coupling
  - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
  - If/when the type definition changes, every component that declares a variable of that data type must also change
- Inclusion or import coupling
  - Component A imports or includes the contents of component B
- External coupling
  - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component level Design

1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
   - These classes are usually not present in the analysis or architectural models
   - These classes include GUI components, operating system components, data management components, networking components, etc.
3) Elaborate all design classes that are not acquired as reusable components
   - Specify message details (i.e., structure) when classes or components collaborate
   - Identify appropriate interfaces (e.g., abstract classes) for each component
   - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
   - Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams
4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
   - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
6) Elaborate deployment diagrams to provide additional implementation detail

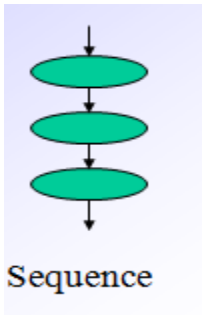
- Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments

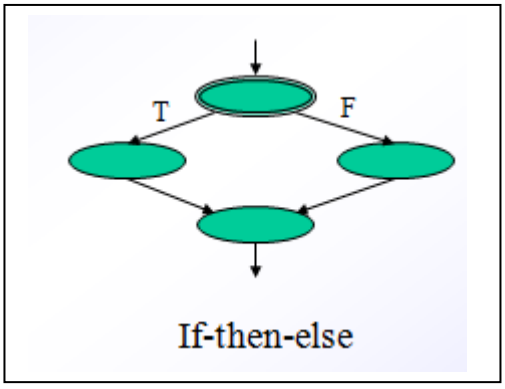
7) Factor every component-level design representation and always consider alternatives
   - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
   - The final decision can be made by using established design principles and guidelines

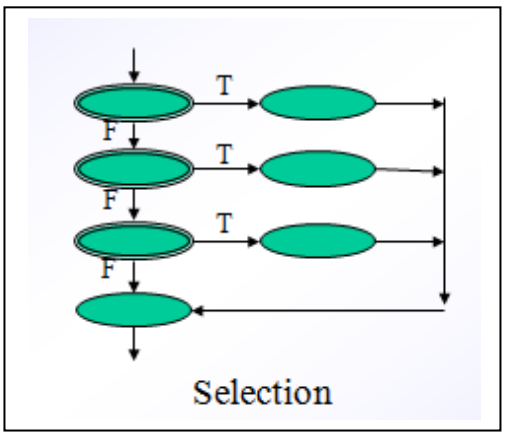
Designing Conventional Components

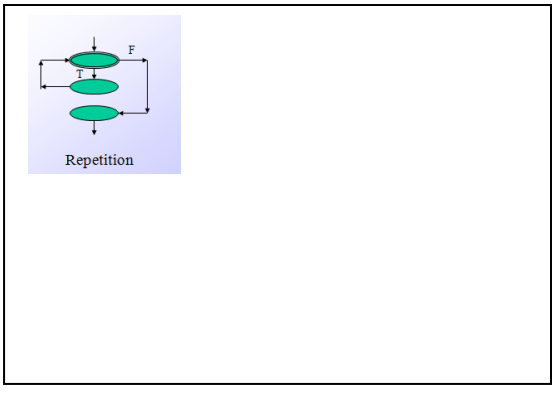
- Conventional design constructs emphasize the maintainability of a functional/procedural program
  - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs
  - **Graphical design notation**
    - Sequence, if-then-else, selection, repetition
  - **Tabular design notation**
  - **Program design language**
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

Graphical design notation:









Tabular Design Notation:

1) List all actions that can be associated with a specific procedure (or module)
2) List all conditions (or decisions made) during execution of the procedure
3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
4) Define rules by indicating what action(s) occurs for a set of conditions

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
regular customer	T	T				
silver customer			T	T		
gold customer					T	T
special discount	F	T	F	T	F	T
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

PERFORMING USER INTERFACE DESIGN

- User Interface design create an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

- **Interface Design**

Easy to learn?

Easy to use?

Easy to understand?

- ***Typical Design Errors***

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
Arcane/unfriendly

GOLDEN RULES

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

- **Place the User in Control**
  - Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
  - Provide for flexible interaction.
  - Allow user interaction to be interruptible and undoable.
  - Streamline interaction as skill levels advance and allow the interaction to be customized.
  - Hide technical internals from the casual user.
  - Design for direct interaction with objects that appear on the screen.

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

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

USER INTERFACE ANALYSIS AND DESIGN

- The overall process for analyzing and designing a user interface begins with the creation of different models of system function.

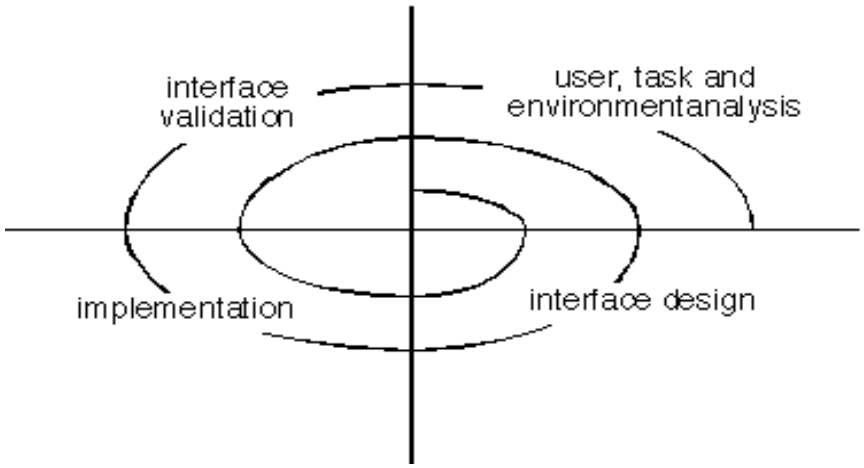
■ Interface Analysis and Design Models

- User model — a profile of all end users of the system
- Design model — a design realization of the user model
- Mental model (system perception) — the user's mental image of what the interface is
- Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics.

■ User Interface Design Process

- The analysis and design process for user interfaces is iterative and can be represented using a spiral model.
- The user interface analysis and design process encompasses four distinct framework activities.
    - User, task and environment analysis and modeling.
    - Interface design.
    - Interface construction (implementation).
    - Interface validation.

- The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are
    - Where will the interface be located physically?
    - Will the user be sitting, standing or performing other tasks unrelated to the interface?
    - Does the interface hardware accommodate space, light or noise constraints?
    - Are there special human factors consideration driven by environmental factors?
- The information gathered as part of the analysis activity is used to create an analysis model for the interface.
- 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.
- 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 interfaced is easy to use and easy to learn
    (3) the users acceptance of the interface as a useful tool in their work.



INTERFACE ANALYSIS

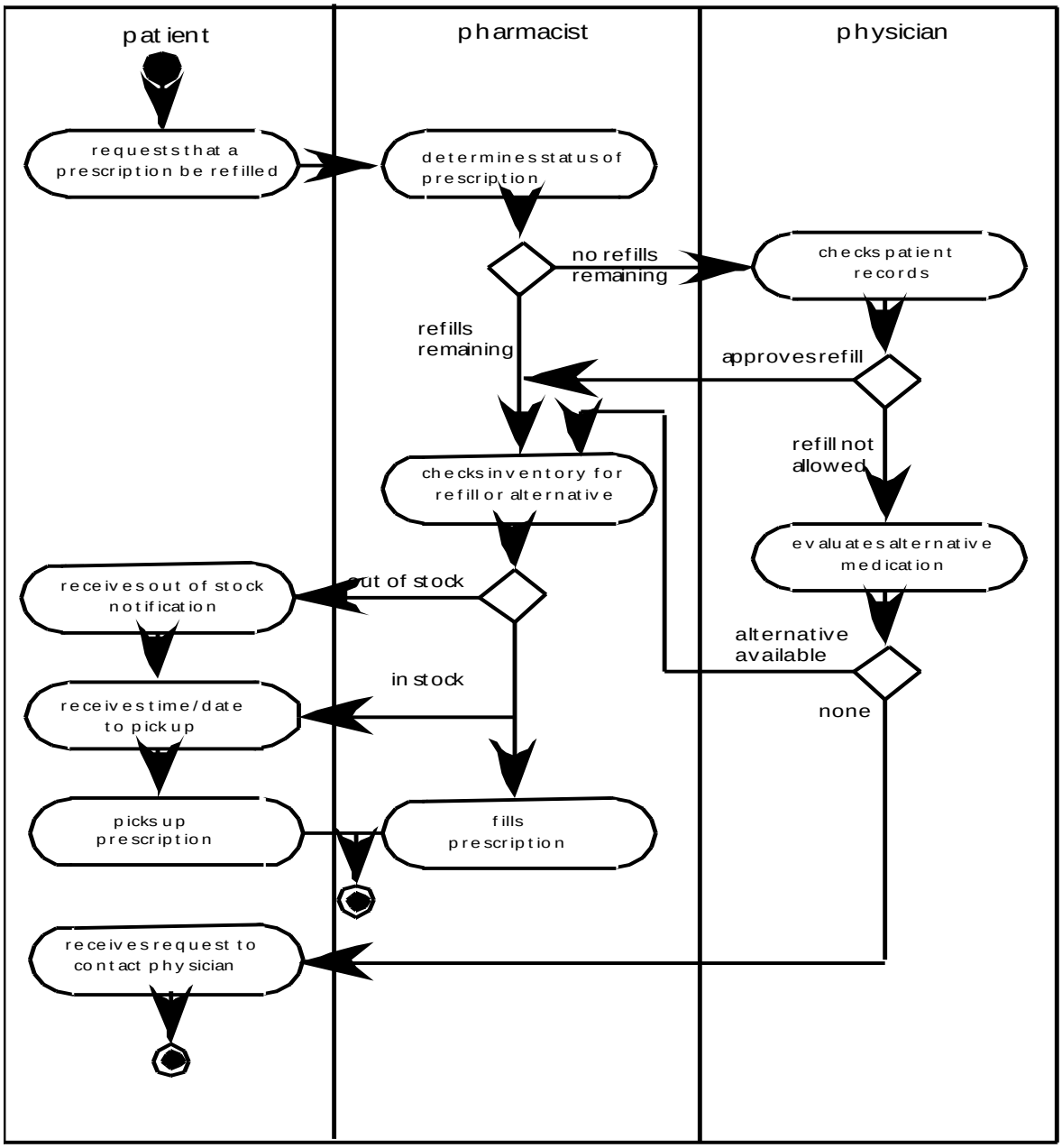
- Interface analysis means understanding
  - the people (end-users) who will interact with the system through the interface;
  - the tasks that end-users must perform to do their work,
  - the content that is presented as part of the interface
  - the environment in which these tasks will be conducted**.**

- **User Analysis**
  - Are users trained professionals, technician, clerical, or manufacturing workers?
  - What level of formal education does the average user have?
  - Are the users capable of learning from written materials or have they expressed a desire for classroom training?
  - Are users expert typists or keyboard phobic?
  - What is the age range of the user community?
  - Will the users be represented predominately by one gender?
  - How are users compensated for the work they perform?
  - Do users work normal office hours or do they work until the job is done?
  - Is the software to be an integral part of the work users do or will it be used only occasionally?
  - What is the primary spoken language among users?
  - What are the consequences if a user makes a mistake using the system?
  - Are users experts in the subject matter that is addressed by the system?
  - Do users want to know about the technology the sits behind the interface?

- **Task Analysis and Modeling**
  - Answers the following questions …
    - What work will the user perform in specific circumstances?
    - What tasks and subtasks will be performed as the user does the work?
    - What specific problem domain objects will the user manipulate as work is performed?
    - What is the sequence of work tasks—the workflow?
    - What is the hierarchy of tasks?

- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

■ **Swimlane Diagram**



■ Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

INTERFACE DESIGN STEPS

- Using information developed during interface analysis; define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

■ Interface Design Patterns

The Design pattern is an abstraction that prescribes a design solution to a specific, well bounded design problem

- Patterns are available for
    - The complete UI
    - Page layout
    - Forms and input
    - Tables
    - Direct data manipulation
    - Navigation
    - Searching
    - Page elements
    - e-Commerce

■ Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

DESIGN EVALUATION

- Design evaluation determines 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 provide impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.
- After the design model has been completed, a first level prototype is created.
- The design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews.
  - 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.
  - 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.
  - The number of actions, tasks and system states indicated by the design model imply the memory load on users of the system
  - Interface style, help facilities and error handling protocol provide indication of the complexity of the interface and the degree to which it will be accepted by the user.
- To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be
  (1) simple yes/no response,
  (2) numeric response,
  (3) scaled (subjective) response,
  (4) likert scales (e.g., strongly agree, somewhat agree),
  (5) percentage (subjective) response or
  (6) open-ended.
- If quantitative data are desired, a form of time study analysis can be conducted.