

# Chapter 3

3.1	Design quality, Classification of Design Activities, Design Concepts: Modularity and Layering, Introduction to Pattern-Based Software Design,
3.2	Software Architecture, Data Design, Object-Oriented versus Function-Oriented Design, Design of Software Objects, Methods, Cohesion and Coupling between Objects,
3.4	User Interface Design: Rules, User Interface Analysis and Steps in Interface Design, Design Evaluation
3.5	Software Reuse, Component-Based Software Engineering

# Design Quality

- **Software design** is an **iterative process** through which **requirements are translated into a “blueprint”** for constructing the software.
- Developers identify and prioritize the qualities of the system that they should optimize.

# Design and Quality

- **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.
- **Explicit Requirements: The Things You Wrote Down**
- **Implicit Requirements: The Things Your Customers Will Expect**
- **Latent Requirements: Things That Will Delight Your Customers**
- **the design must be a readable, understandable** guide for those who generate code and for those who test and subsequently support the software.
- **the design should provide a complete picture of the software**, addressing the data, functional, and behavioral domains from an implementation perspective.

# Quality Guidelines

- **A design should exhibit an architecture** that:
  1. Created using standard architectural patterns.
  2. composed of components that exhibit good design characteristics.
  3. can be implemented in an evolutionary fashion
- **A design should be modular:** logically partitioned into elements or subsystems

# Quality Guidelines

- 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**.

# Quality Guidelines

- 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**.

# Design Activities

- The implementation phase of software development is the **process of converting a system specification into an executable system** through the **design of system**.



# Design Activities

- The design process activities are the followings:

## 1. Architectural design

- The architectural design for software is the equivalent to the floor plan(overview) of a house.
- Architectural design elements give us an overall view of the software.

# Design Activities

- Architectural model is derived from three sources:
  - (1) Information about the application domain** for the software to be built
  - (2) specific requirements**
  - (3) the availability of architectural Styles and patterns**

# Design Activities

## 2. Interface Design

- analogous to a set of detailed drawings.
- Specifications of software
- The interface design elements for software depict **information flows into and out of the system** and how it is communicated among the components defined as part of the architecture.

# Design Activities

- 3 important elements of interface design:
  - (1) the user interface (UI)
  - (2) external interfaces
  - (3) internal interfaces

## 3. Component-Level Design

- The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.

# Design Activities

- The component-level design for software fully **describes the internal detail of each software component.**
- component-level design **defines data structures** for all local data objects and **algorithmic detail.**
- Within the context of object-oriented software engineering, a **component is represented in UML diagrammatic form.**

# Design Activities

## 4. Data structure design.

- The data structures used in the system implementation are designed in detail and specified.

## 5. Algorithm design.

- In this activity the algorithms used to provide services are designed in detail and specified.

# Design Activities

## 6. Deployment-Level Design

- indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- For example, the elements of the *SafeHome product configured* to operate within three primary computing environments
- home-based PC
- SafeHome control panel
- server housed at CPI Corp. (providing Internet-based access to the system).

# Design Principles

- The **design process** is a sequence of steps that enable the designer to **describe all aspects of the software to be built.**
1. **The design process should not suffer from “tunnel vision.”**- A good designer should consider alternative approaches.
  2. **The design should be traceable to the analysis model.**- means for tracking how requirements have been satisfied by the design model.



# Design Principles

3. **The design should not reinvent the wheel.** - Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world.** - structure of the software design should be replica of structure of the problem domain.

# Design Principles

- 5. The design should exhibit uniformity and integration.**
  - format should be defined for a design team before design work begins.
  - design is integrated if care is taken in defining interfaces between design components.
- 6. The design should be structured to accommodate change.**

# Design Principles

- 7. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.**
- Well designed software should never “bomb.”
  - It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

# Design Principles

## **8. Design is not coding, coding is not design.**

- the level of abstraction of the design model is higher than source code.
- only design decisions made at the coding level address the small implementation details.

## **9. The design should be assessed for quality as it is being created, not after the fact.**

## **10. The design should be reviewed to minimize conceptual (semantic) errors.**

# Design Concepts

- Software Design concepts are **fundamental concepts** which **provides** the software designer with a **foundation from which more sophisticated design methods** can be applied for software design process

# Design Concepts

## 1. Abstraction

- 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.
- lower levels of abstraction, a more procedural orientation is taken.
- Each step in the software process is a refinement in the level of abstraction of the software solution.

# Design Concepts

- A **procedural abstraction** is a named sequence of instructions that has a specific and limited function. (algorithm)
  - Example: word **DOOR** implies a long sequence of procedural steps
- A **data abstraction** is a named collection of data that describes a data object
  - **Example:** word **DOOR** would encompass a set of attributes that describe the door

# Design Concepts

- **control abstraction** implies a program control mechanism for coordinating activities.

## 2. Refinement

- Refinement is process of elaboration
- top-down design strategy
- A program is developed by successively refining levels of procedural detail.



# Design Concepts

## Stepwise Refinement



# Design Concepts

- Refinement causes the designer to **elaborate on the original statement, providing more and more detail** as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts.

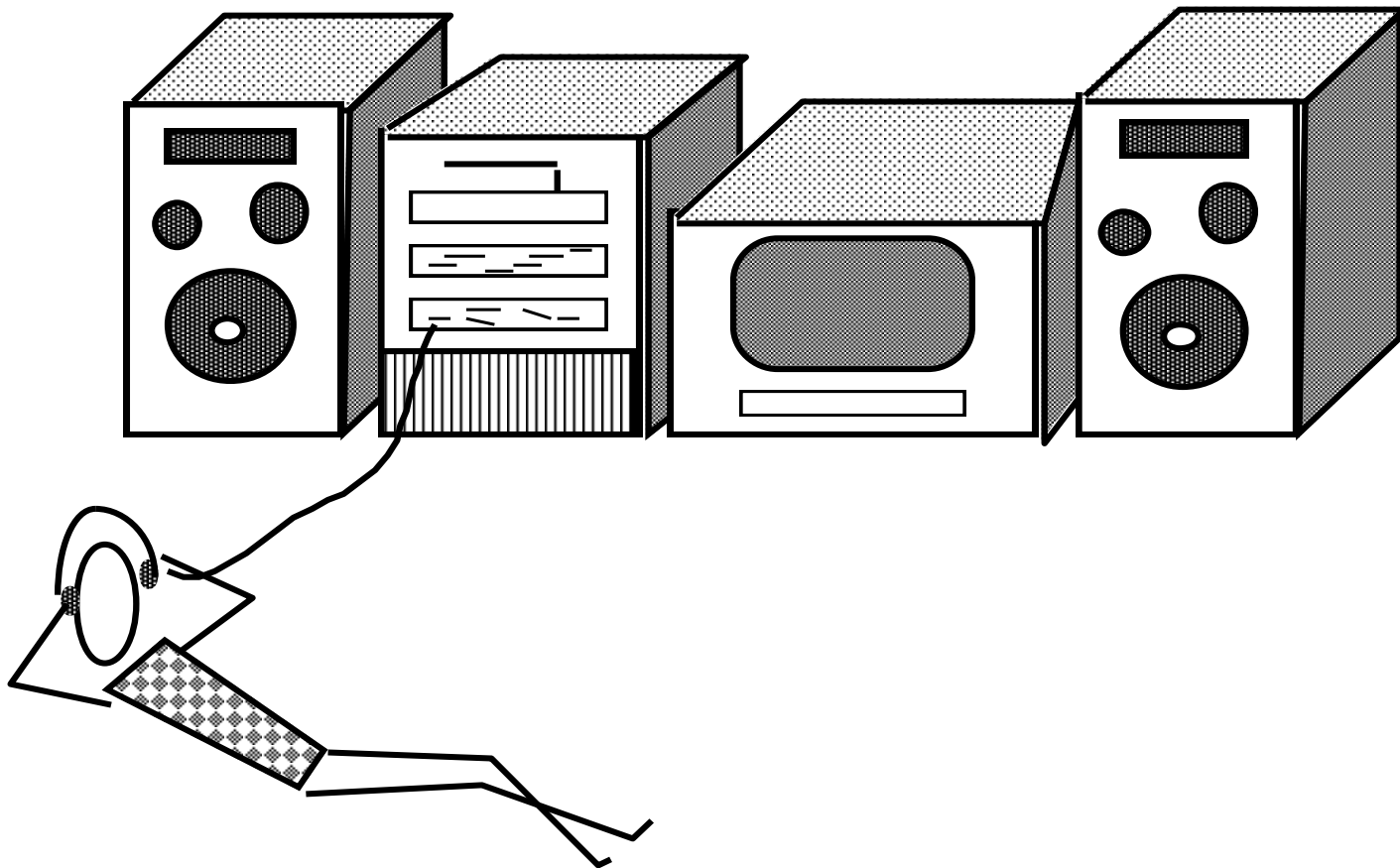
# Design Concepts

## 3. Modularity

- **software is divided** into separately named and addressable **components**, often called modules, that are **integrated to satisfy problem requirements**.
- **modularity** is the single attribute of software that **allows** a program to be **intellectually manageable**.

# Modular Design

*easier to build, easier to change, easier to fix ...*

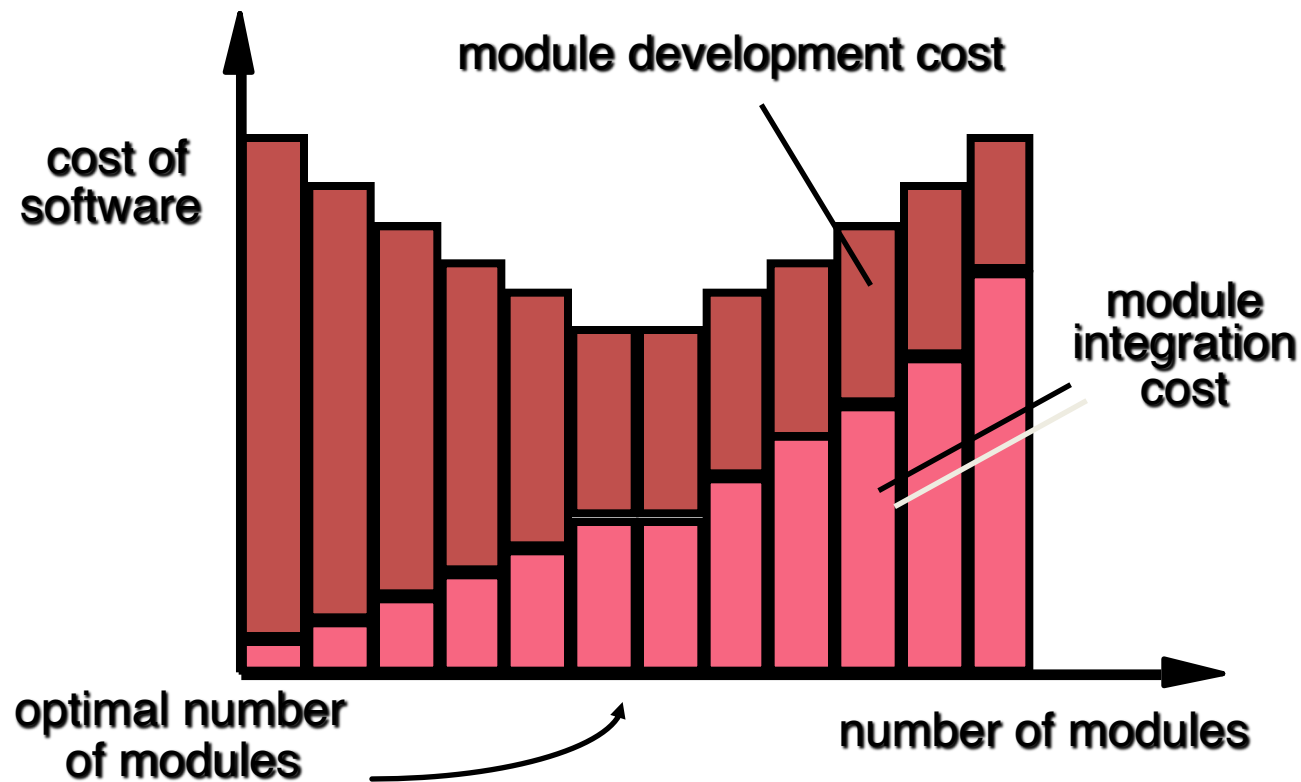


# Design Concepts

- **"divide and conquer"** conclusion—it's easier to solve a complex problem when you break it into manageable pieces - **an argument for modularity.**

# Modularity: Trade-offs

*What is the "right" number of modules for a specific software design?*



# Design Concepts

- five criteria to define an effective modular system:
  1. **Modular decomposability** – systematic mechanism for **decomposing** the problem into sub problems.
  2. **Modular composability** – **reusing** existing system into new system yield modular solution and not reinvent wheel.
  3. **Modular understandability** - understood as **standalone** unit , it will be easier to build and easier to change.

# Design Concepts

- 4. **Modular continuity** - small changes to the system requirements - changes to **individual modules** rather system – change induced side effects minimized.
  
- 5. **Modular protection** - aberrant condition - error-induced side effects minimized.



# Design Concepts

## 4. Software Architecture

- architecture—the overall structure of the software.
- architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components.

# Design Concepts

- Well designed architecture serves as framework for detailed design activities which can also help in reuse of design level concepts.
- **Properties of an architectural design:**
  1. **Structural properties:** This aspect of the architectural design representation defines the components of a system and their interaction with one another.
  2. **Extra-functional properties:** like performance, capacity, reliability, security, adaptability, and other system characteristics.

# Design Concepts

**3. Families of related systems:** The architectural design **should draw upon repeatable patterns** that are commonly encountered in the design of families of similar systems.

# Design Concepts

- **Different architectural models**
  1. **Structural models** represent architecture as an organized collection of **program components**.
  2. **Framework models** identify **repeatable** architectural design frameworks (**patterns**)
  3. **Dynamic models** address the behavioral aspects of the program architecture, **indicating changes** in configuration of system.

# Design Concepts

- 4. **Process models** focus on the design of the **business or technical process** that the system must accommodate.
- 5. **Functional models** can be used to represent the **functional hierarchy of a system**.

# Design Concepts

## 5. Patterns

- a design pattern describes a design structure that solves a particular design problem within a specific context.
- 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**

# Design Concepts

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

# Design Concepts

## 6. Information Hiding

- The intent of information hiding is to **hide the details of data structures and procedural processing** behind a module interface.
- Knowledge of the **details need not be known by users**
- independent modules that **communicate with one another only that information necessary** to achieve software function



# Design Concepts

- information hiding provides the **greatest benefits when modifications** are required.
- Because most data and procedural **detail are hidden** from other parts of the **software errors less likely to propagate.**

# Design Concepts

## 7. Functional Independence

- Software should be designed such that **each module addresses a specific subset of requirements** and has a **simple interface with other program structure**.
- Software with effective modularity, that is, independent modules, is easier to develop.

# Design Concepts

- **Independent modules**
  - ✓ easier to maintain (and test)
  - ✓ effects caused by design or code modification limited
  - ✓ error propagation is reduced
  - ✓ reusable modules are possible.
- functional independence is a **key to good design**, and **design** is the **key to software quality**.

# Design Concepts

- Independence is assessed using two qualitative criteria:

**COHESION** - the degree to which a module performs one and only one function.

**COUPLING** - the degree to which a module is "connected" to other modules in the system.

# Design Concepts

## 1. Cohesion

- A cohesive module performs a **single task**, requiring **little interaction** with **other components** in other parts of a program.
- **High degree of cohesion is advisable but it is also necessary and advisable for components to perform multiple functions.**

# Design Concepts

## 2. Coupling

- Coupling is an indication of **interconnection among modules** in a software structure.
- Coupling depends on the **interface complexity** between modules – entry point and type of data that passes through interface.
- In software design, one should strive for the lowest possible coupling.

# Design Concepts

- Simple connectivity among modules results in software that is **easier to understand** and **less prone to a “ripple effect”**

# Design Concepts

## 8. Refactoring

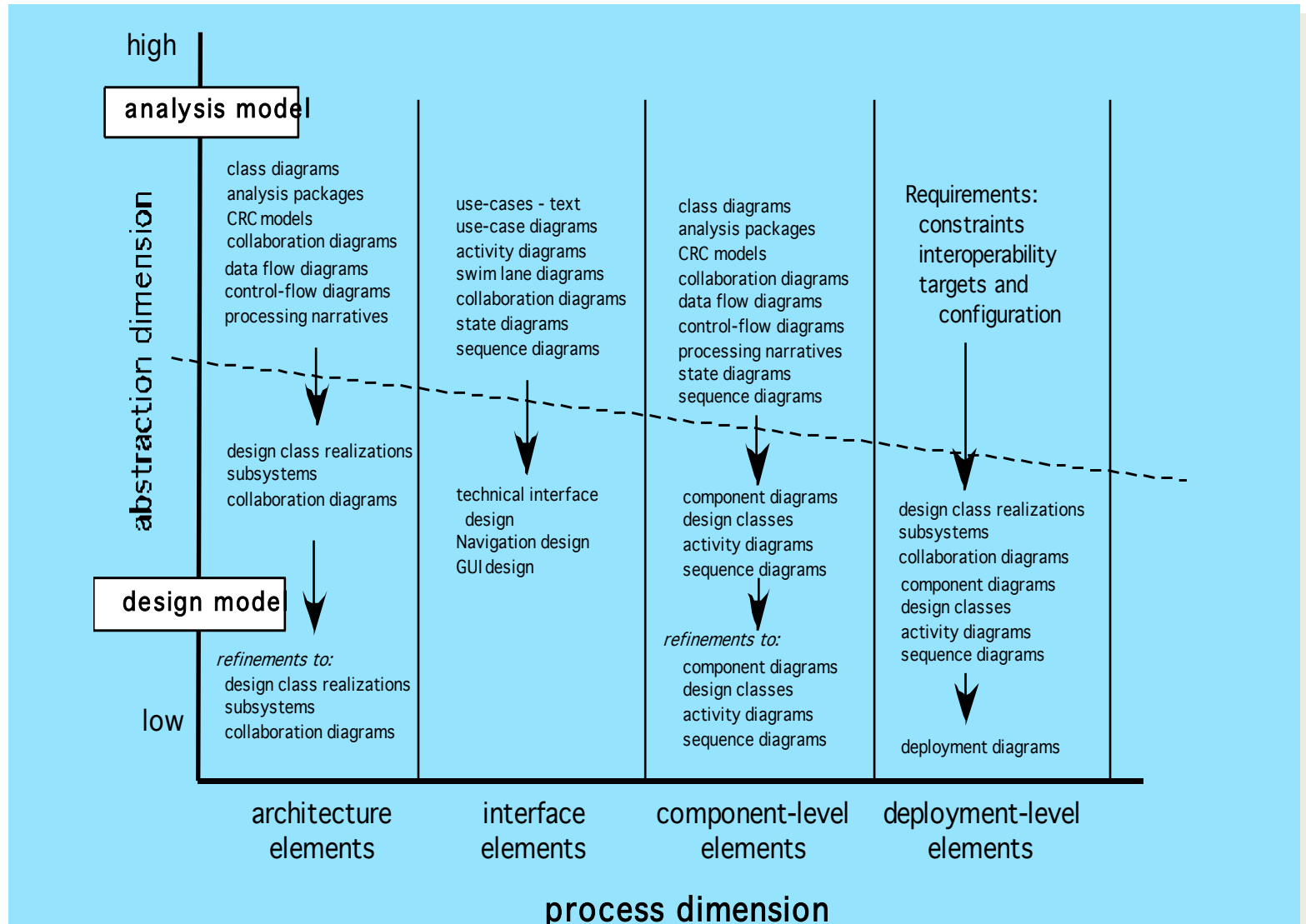
- Refactoring is the process of **changing a software system** in such a way that it **does not alter the external behavior** of the code [design] yet **improves its internal structure**.
- When software is refactored:
  - ✓ the existing design is examined for redundancy
  - ✓ unused design elements
  - ✓ inefficient or unnecessary algorithms
  - ✓ poorly constructed or inappropriate data structures,
  - ✓ any other design failure that can be corrected to yield a better design.



# The Design Model

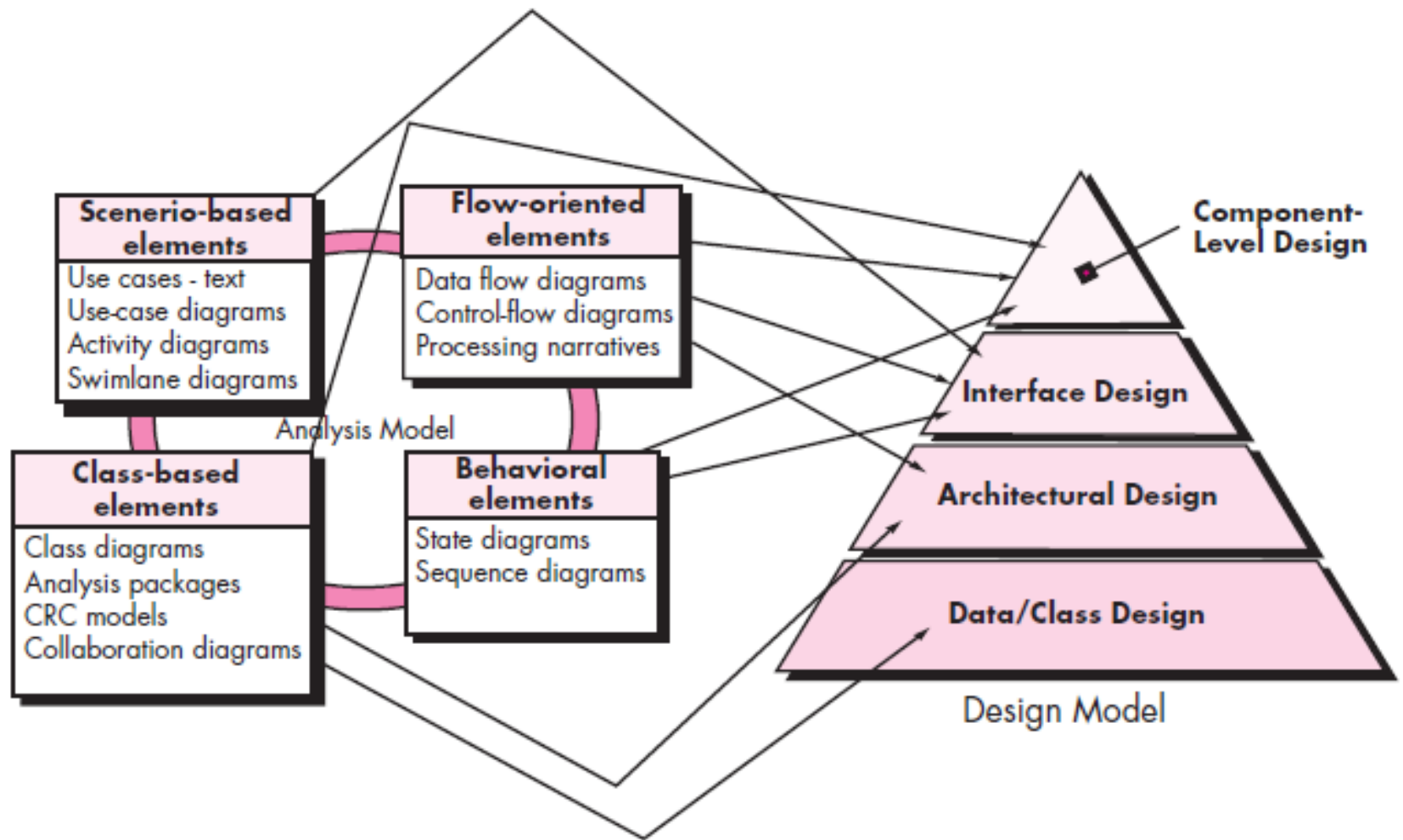
- The design model can be viewed in two different dimensions:
- Each element of analysis model is transformed into design model and refined iteratively.
- ✓ The **process dimension** as design tasks are executed as part of software process.
- ✓ The **abstraction dimension** represents the level of detail in each element

# The Design Model



**FIGURE 8.1**

Translating the requirements model into the design model



# The Design Model

- Elements of Design Model

- ✓ **Data Design Elements**

- data structures
- database architecture

- ✓ **Architectural Design Elements**

- The architectural design for software is the equivalent to the **floor plan** of a house.
- **Three sources for development of architecture model**
  - (1) information** about the **application domain** for the software to be built.
  - (2) specific requirements** model elements (data flow diagrams , collaboration etc.)

# The Design Model

(3) architectural **styles and patterns**

## ✓ **Interface Design Elements**

- The interface design for software is analogous to a set of **detailed drawings**
- 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
  - (3) internal interfaces between various design components.

# The Design Model

- Interface design elements **allow external/ internal communication** and **collaboration** among various components.
- ✓ **Component-Level Design Elements**
  - The component-level design for software is the equivalent to a **set of detailed drawings** (and specifications) E.g. Each room of house.
  - Relationship between the components.
- ✓ **Deployment-Level Design Elements**
  - Deployment-level design elements indicate how **software functionality and subsystems** will be allocated within the software environment.

# Pattern-based Software Design

- The best designers in any field have an uncanny **ability to see patterns that characterize a problem** and corresponding patterns that can be combined to create a solution.
- A description of a design pattern may also consider a set of design forces.
  - ***Design forces* describe non-functional requirements** (e.g., ease of maintainability, portability) **associated with the software** for which the pattern is to be applied.

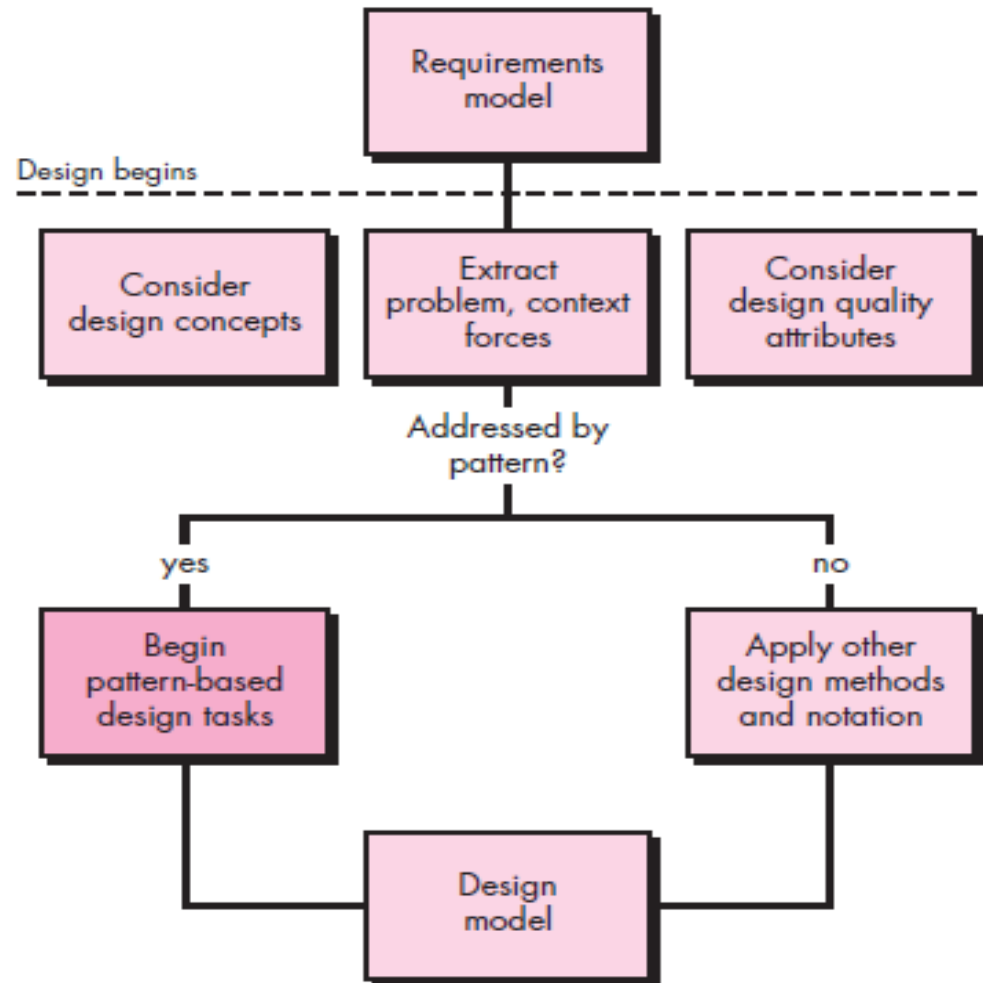
# Pattern-based Software Design

- The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a **variety of problems (problem solving)**.
- Throughout the design process, one should look for every opportunity to **apply existing design patterns** rather than creating new ones.



# Pattern-based Software Design

## 1. Pattern-Based Design in Context



Use methods and modeling tools available for architectural, component level, and interface design.

# Pattern-based Software Design

## 2. Thinking in Patterns

- **“new way of thinking”** when one **uses patterns** as part of the design activity.
- Good design begins by considering context—the big picture.

# Pattern-based Software Design

- **approach that enables a designer to think in patterns:**
  1. Context of **software build and requirements** to be understood.
  2. extract the **patterns** that are **present at level of abstraction**.
  3. Begin design with “**big picture**” patterns that establish a **context or skeleton** for further design work.

# Pattern-based Software Design

4. **“Work inward from the context”** lower levels of abstraction that contribute to the design solution.
5. **Repeat steps 1 to 4** until the complete design is fleshed out.
6. **Refine the design** by adapting each pattern to the specifics of the software

# Example of Thinking in Pattern

- **SafeHomeAssured.com** WebApp.
- How to **provide information** about *SafeHome* products and services?
- How to **sell *SafeHome*** products and services to customers?
- How to establish **Internet-based monitoring and control** of an installed security system?

# Pattern-based Software Design

## 3. Design Tasks

- The following design tasks are applied when a pattern-based design philosophy is used:
  1. **Examine the requirements model and develop** a problem hierarchy.
  2. Determine if a **reliable pattern language** has been **developed** for the problem domain.

# Pattern-based Software Design

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

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

**Repeat steps until all broad problems have been addressed**

# Pattern-based Software Design

5. **user interface design problems** have been - **search** the many user interface **design pattern** repositories for appropriate patterns.
6. **Compare various pattern** against the existing ones for better designing.
7. Be certain to refine the design as it is derived from patterns using **design quality criteria** as a guide.



# Pattern-based Software Design

## 4. Building a Pattern-Organizing Table

- As pattern based design proceeds organizing and categorizing problem may occur.
- Solution: A pattern-organizing table can be implemented as a spreadsheet model

# Pattern-based Software Design

- Pattern Organizing Table

	Database	Application	Implementation	Infrastructure
<b>Data/Content</b>				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
<b>Architecture</b>				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
<b>Component-level</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
<b>User interface</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

# Pattern-based Software Design

## 5. Common Design Mistakes

- A number of common mistakes occur when pattern-based design is used.
  1. **not enough time** has been spent to understand the underlying problem.
  2. **Wrong pattern selected.**
  3. **Quality** standards ignored.

# Software Architecture

- **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.

# Software Architecture

- **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”.

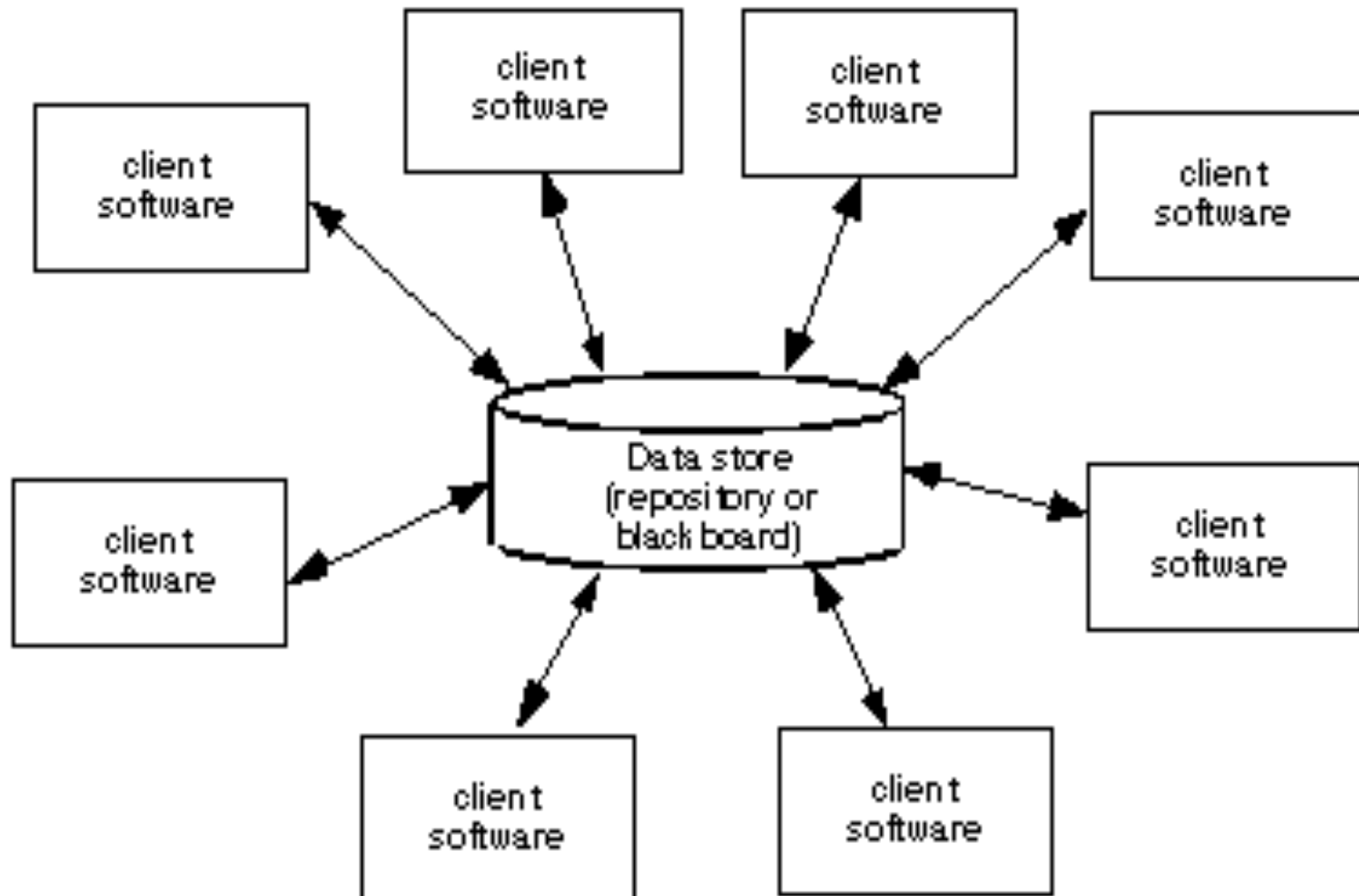
# Architectural Styles

- **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
  - (4) **semantic models** that enable a designer to **understand the overall properties of a system** by analyzing the known properties of its constituent parts.

# Architectural Styles

- **Data-centered architectures.**
- A **data store** (e.g., a file or database) **resides at the center** of this architecture and is accessed frequently by other components.
- **Client software** accesses a **central repository**.
- Data-centered architectures promote **integrability**.
- New client components can be added easily.
- Client components independently execute processes.

# Data-Centered Architecture

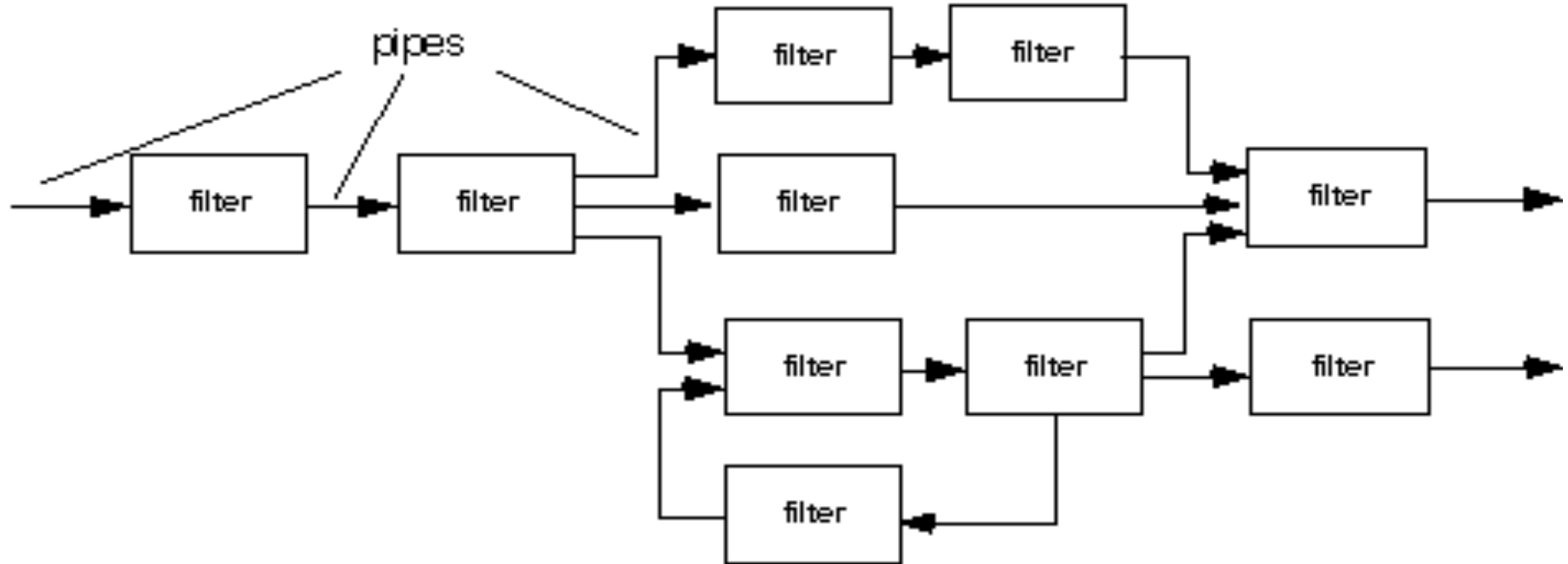




# Architectural Styles

- **Data-flow architectures**
- This architecture is applied when **input data** are to be **transformed through a series of computational or manipulative components into output data**.
- Pipe and filter pattern.
- Each filter works independently.
- If the data **flow degenerates into a single line of transforms**, it is termed **batch sequential**.

# Data Flow Architecture



(a) pipes and filters

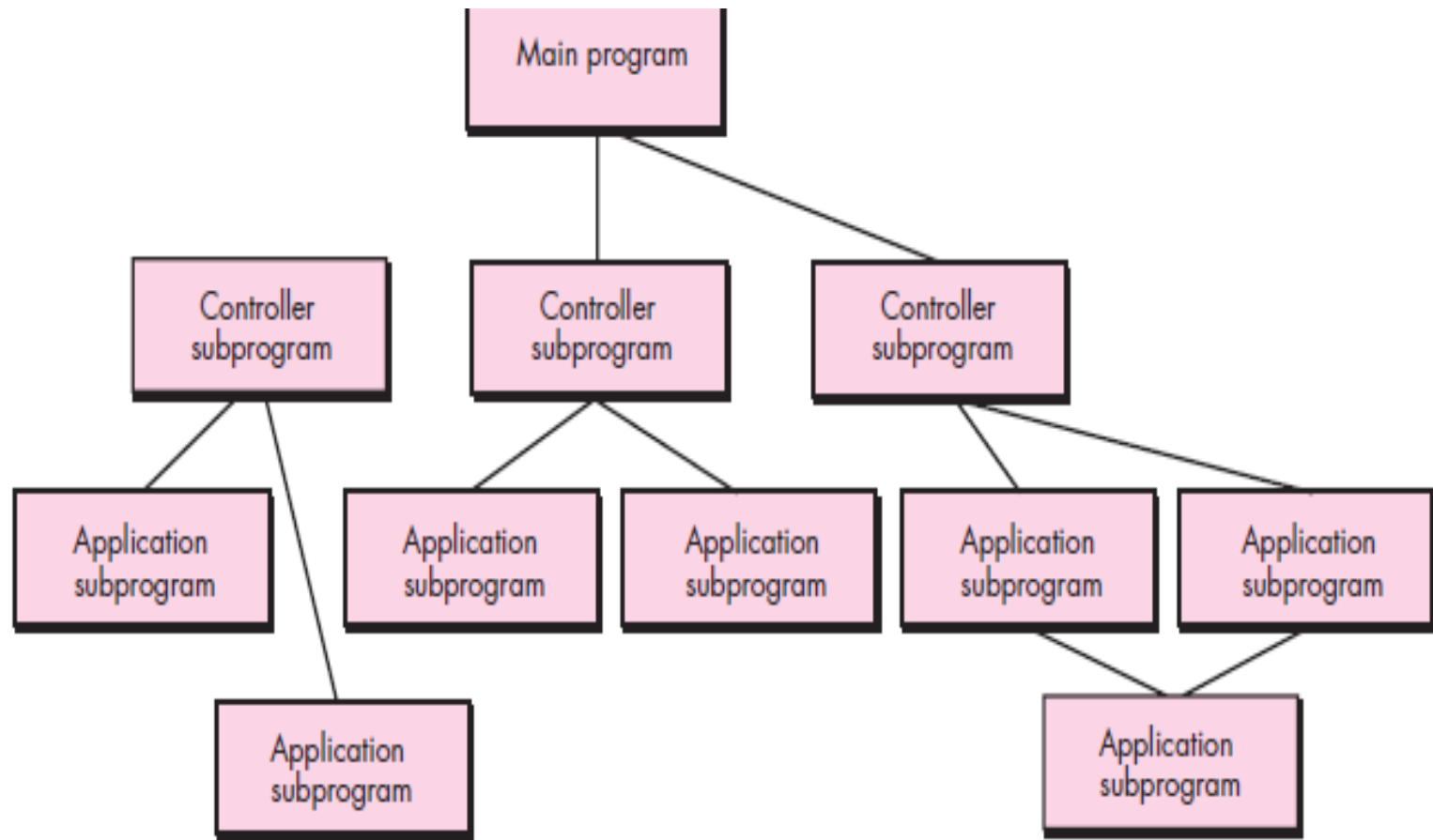


(b) batch sequential

# Architectural Styles

- **Call and return architectures.**
- This architectural style enables you to achieve a **program structure that is relatively easy to modify and scale.**
- **Sub styles of call and return architecture:**
  - 1. Main program/subprogram architectures: main program invokes a number of program components that in turn may invoke still other components.**
  - 2. Remote procedure call architectures: The components of a main program/subprogram architecture are distributed across multiple computers on a network.**

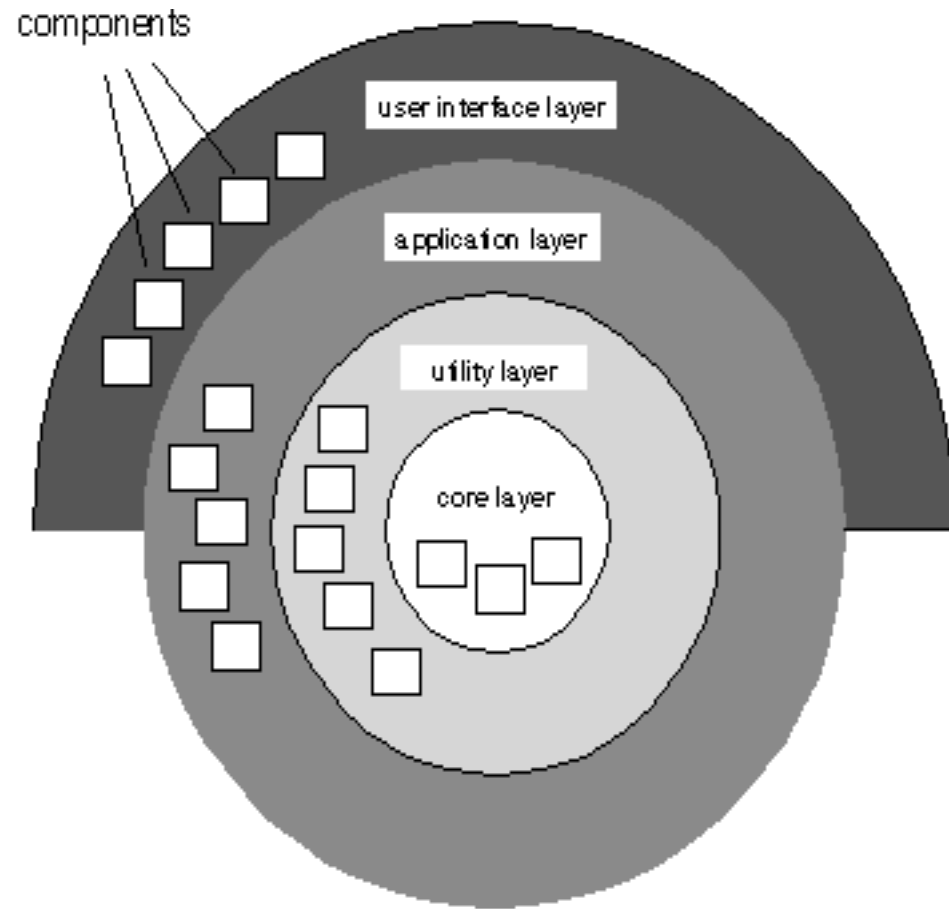
# Call and return architectures.



# Architectural Styles

- **Layered architectures.**
- Layered architecture focuses on the **grouping of related functionality within an application** into distinct layers that are stacked vertically on top of each other.

# Layered Architecture



# Architectural Patterns Issues

- A software architecture may have a number of **architectural patterns** that **address** issues such as **concurrency, persistence, and distribution**.
- **(1) Concurrency**—applications must **handle multiple tasks** in a manner that simulates parallelism.
- Example 1: **Operating System Process Management pattern** that provides built-in OS features that allow components to execute concurrently.
- Example 2: **Task Scheduler pattern** invokes the next concurrent object

# Architectural Patterns

**(2) Distribution** - The distribution problem addresses the manner in which systems or components within systems communicate with one another in a **distributed environment**.

- Example 1: **Broker pattern**: broker acts as a middle man to constitute communication.



# Architectural Patterns

**(3) Persistence - Data persists if it survives past the execution of the process that created it.**

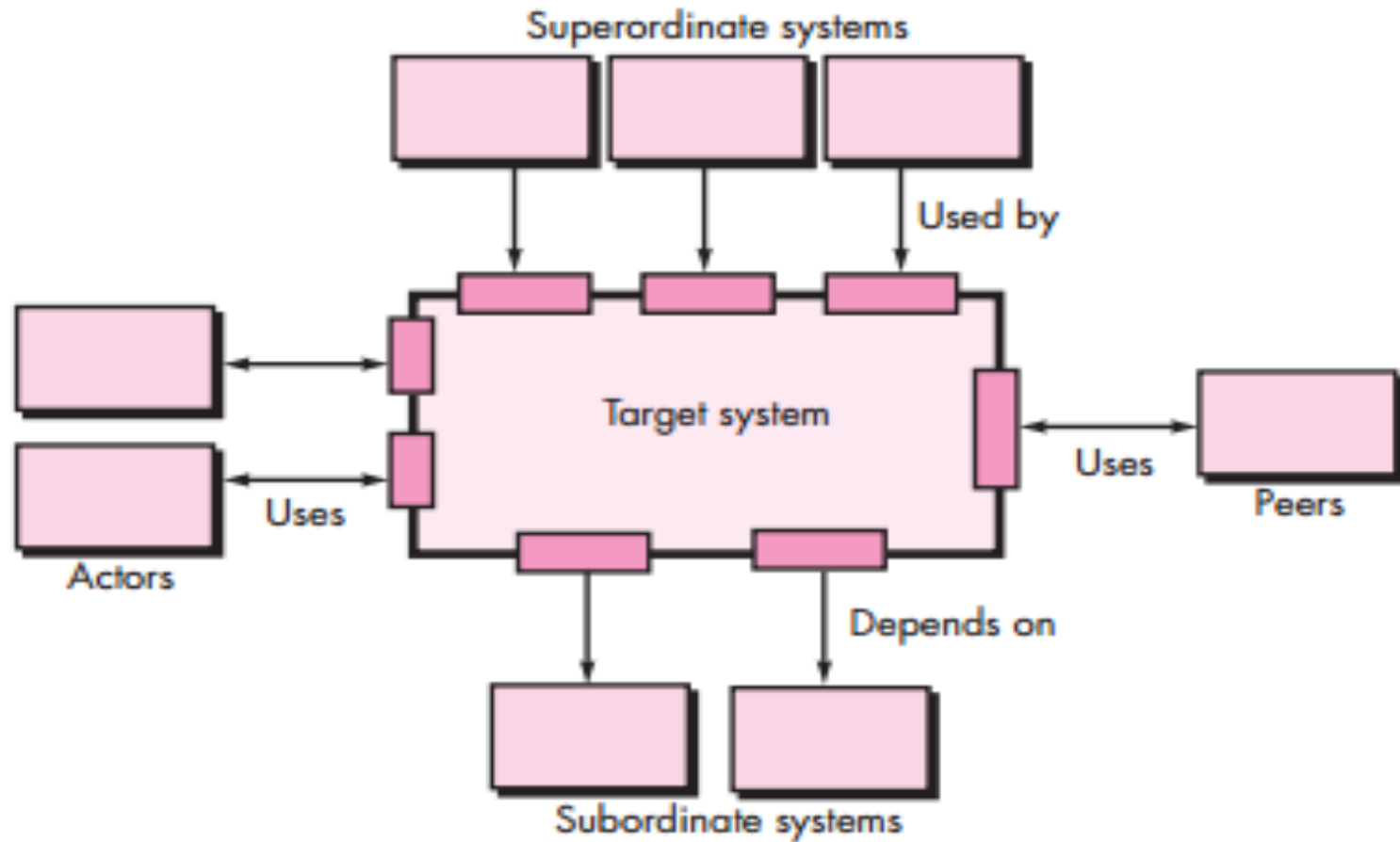
- **Persistent data are stored in a database or file and modified later.**

**Example 1: Database management system pattern** which stores and retrieves data.

# Architectural Design

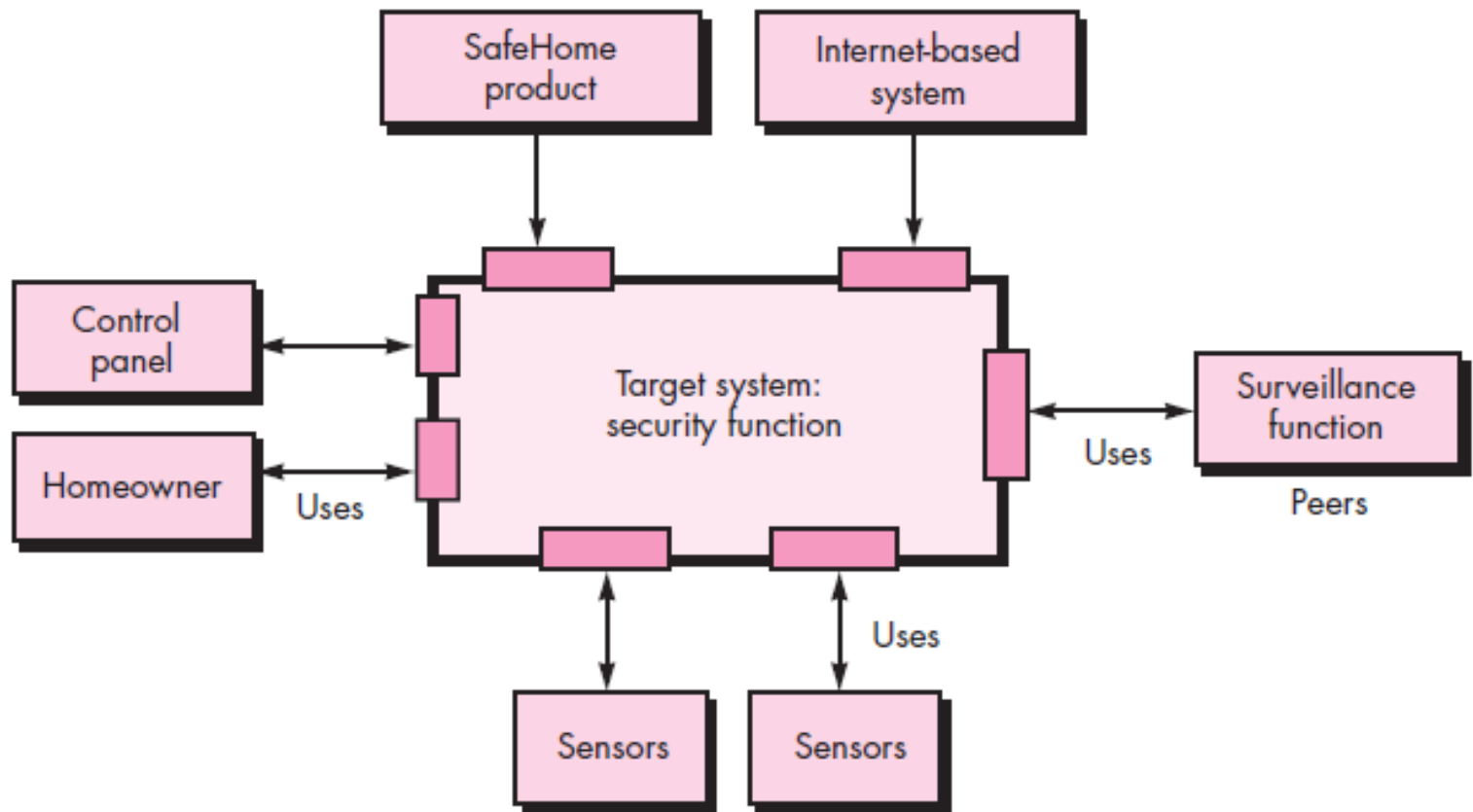
- The software must be placed into context
  - the **design** should **define** the **external entities** (other systems, devices, people) that the **software interacts** with and the **nature of the interaction**
- A set of architectural archetypes should be identified
  - An **archetype** is an abstraction (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 Diagram** - model the manner in which software interacts with entities external to its boundaries.



**FIGURE 9.6**

Architectural context diagram for the *SafeHome* security function



**Archetypes** - An archetype is a class or **pattern that represents a core abstraction that is critical to the design** of an architecture for the target system.

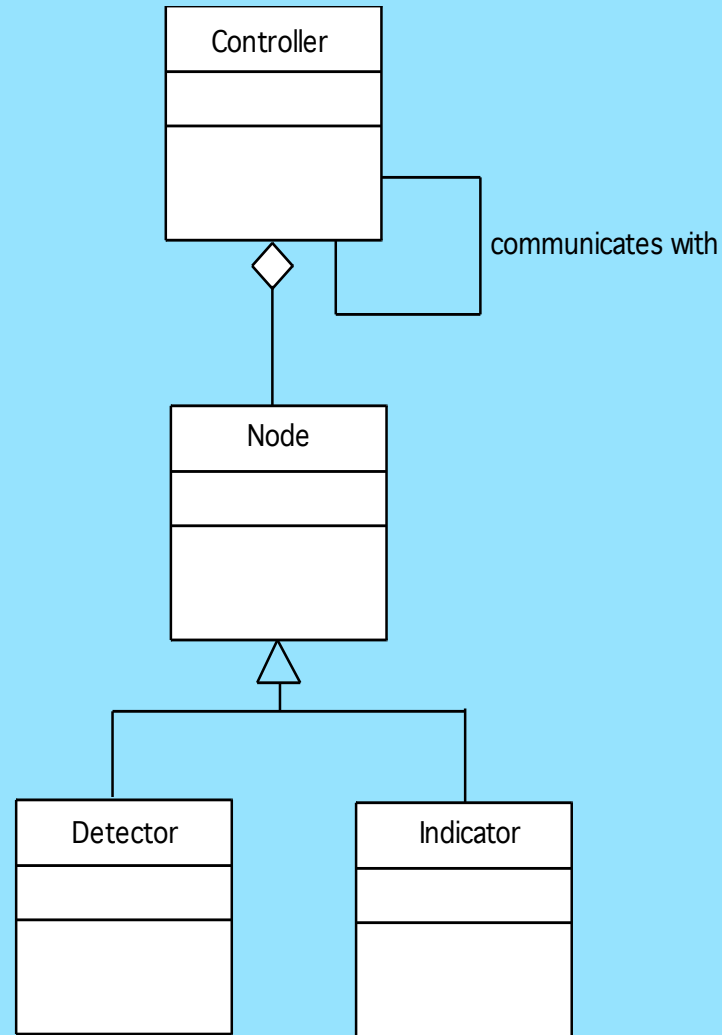
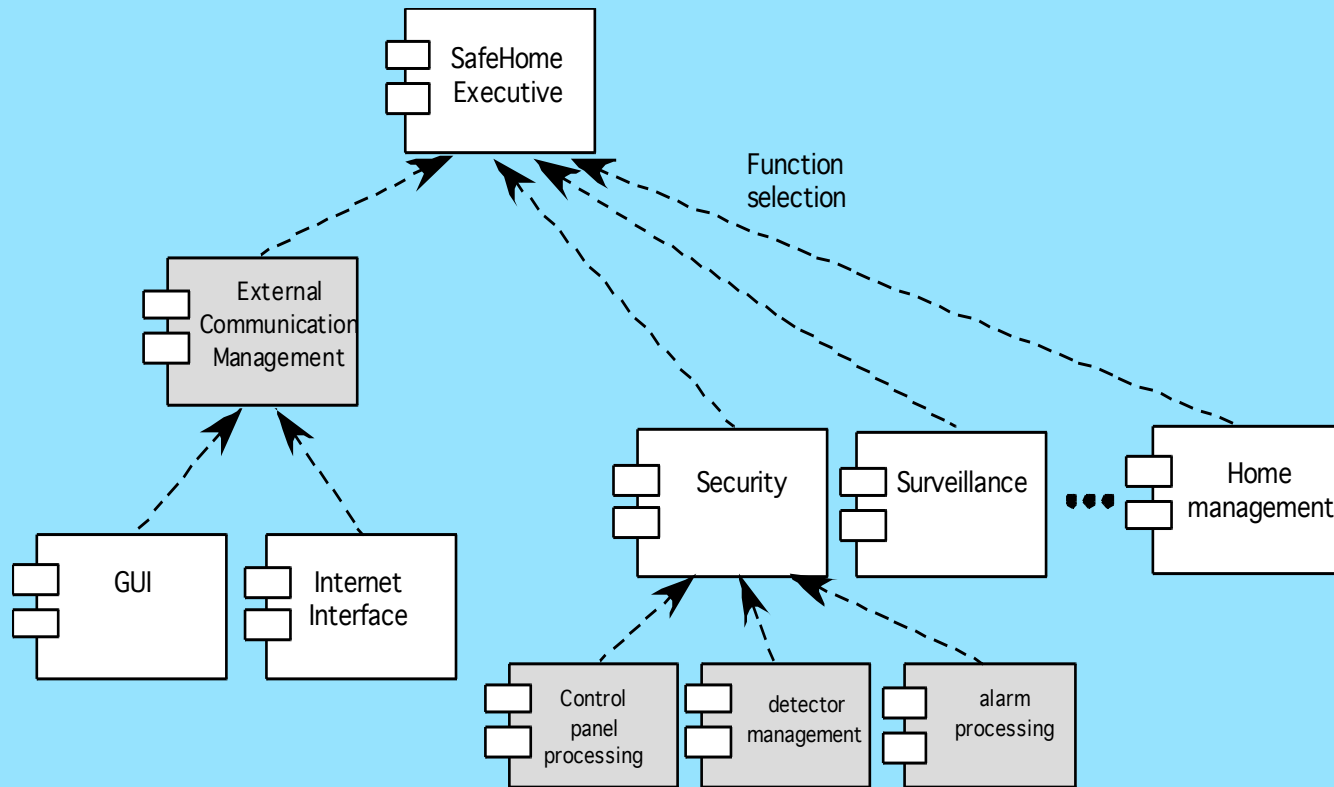
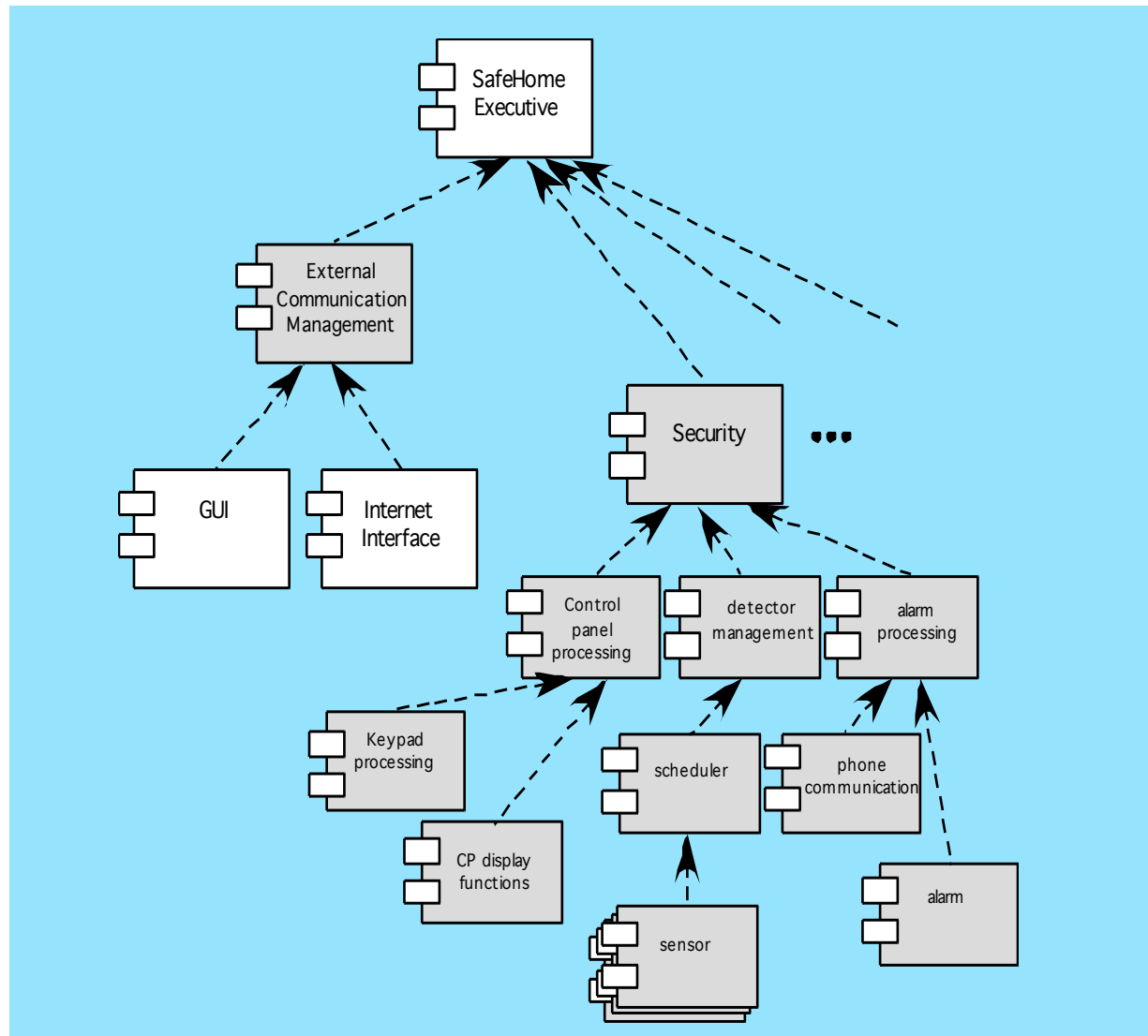


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

# Component Structure



Refined Component Structure - A well-designed habitat allows for the **successful evolution of all the components needed in a software system.**



# Cohesion

- Cohesion 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.
- **Different types of cohesion**
  - (1) **Functional** - Based on **operations** , this level of cohesion occurs when a component **performs a targeted computation** and then returns a result.

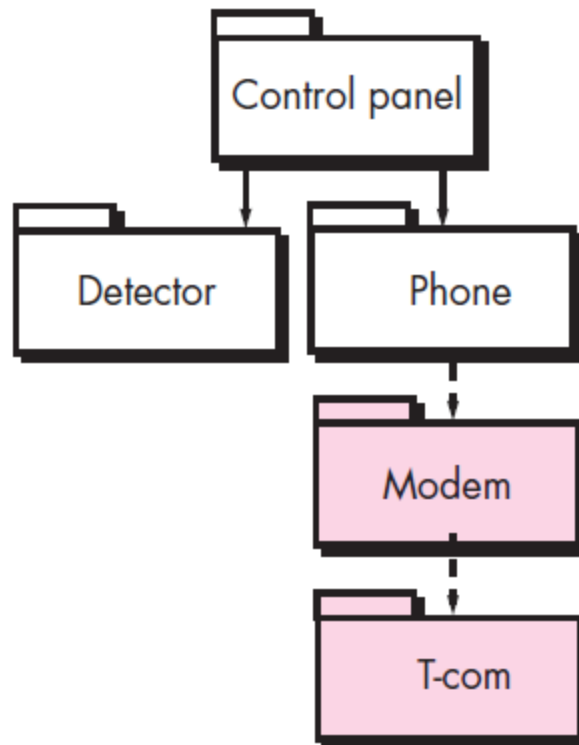


# Cohesion

**(2) Layer** - Exhibited by packages, components, and 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**.

**(3) Communicational.** All operations that access the same data are defined within one class. Focus is only on data in question.

# Layered Cohesion



# Coupling

- **Coupling** is a qualitative measure of the **degree to which classes are connected to one another.**
- **Different types of coupling**
  - (1) Content coupling** Occurs when one component **secretly modifies data** that is internal to another component
  - (2) Common coupling** Occurs when a number of components **all make use of a global variable.** Common coupling can **lead to error propagation.**

# Coupling

**(3) Control coupling** – Occurs when **one operation controls flow of another operation** and control signals are sent. Changes done to an operation are to be monitored for errors.

**(4) Stamp coupling** – Occurs when systems **operation are in nested state** , modifying the system becomes complex.

Similar to common coupling but here global data is accessible to selected routines.

# Coupling

**(5) Data coupling** - Occurs when operations **pass long strings of data arguments**.

B/W of communication increases complexity increases maintenance becomes difficult.

**(6) Routine call coupling** - Occurs when one operation invokes another

**(7) Type use coupling.** Occurs when component **A** uses a data type defined in component **B**.

If the type definition changes, every component that uses the definition must also change.

# Coupling

**(8) Inclusion or import coupling.** Occurs when component **A** imports or includes a package or the content of component **B**.

**(9) External coupling** - Occurs when a **component communicates or collaborates with infrastructure components** (e.g., operating system functions, database capability, telecommunication functions).

This type of coupling is necessary but it should also be limited.

# User Interface Design Rules

## **1. Place the user in control.**

- A system or user interface **should be designed keeping user in mind.**
- Interface designed should be **easy to use and build.**
- 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.**

Example: Spell Checking mode

# User Interface Design Rules

**(2) Provide for flexible interaction.** Because different users have different interaction preferences, choices should be provided.

**Example:** software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen etc.

**(3) Allow user interaction to be interruptible and undoable.**

**(4) Hide technical internals from the casual user.**



# User Interface Design Rules

**(5) Design for direct interaction with objects that appear on the screen.**

Example: Zoom option

## **2. Reduce the User's Memory Load**

- **The more a user has to remember, the more error-prone the interaction with the system will be.**
- well-designed user interface does not tax the user's memory

# User Interface Design Rules

- defines design principles that enable an interface to reduce the user's memory load:

**(1) Reduce demand on short-term memory** – complex tasks demand on short term memory is high - interface designed to reduce dependency.

**(2) Establish meaningful defaults** - “reset” option should be available for enabling original default values.

# User Interface Design Rules

**(3) Define shortcuts that are intuitive (easy to use).**

**Ctrl S , alt P..**

**(4) The visual layout of the interface should be based on a real-world metaphor.**

**(5) Disclose information in a progressive fashion -**  
The interface should be organized hierarchically.

- An example, word-processing applications

# User Interface Design Rules

## 3. Make the Interface Consistent

“Things that look different should act different.  
Things that look the same should act the same.”

**(1) Allow the user to put the current task into a meaningful context.**

- Complex layers to be implemented with clear indication to the user about the context of the work at hand.

# User Interface Design Rules

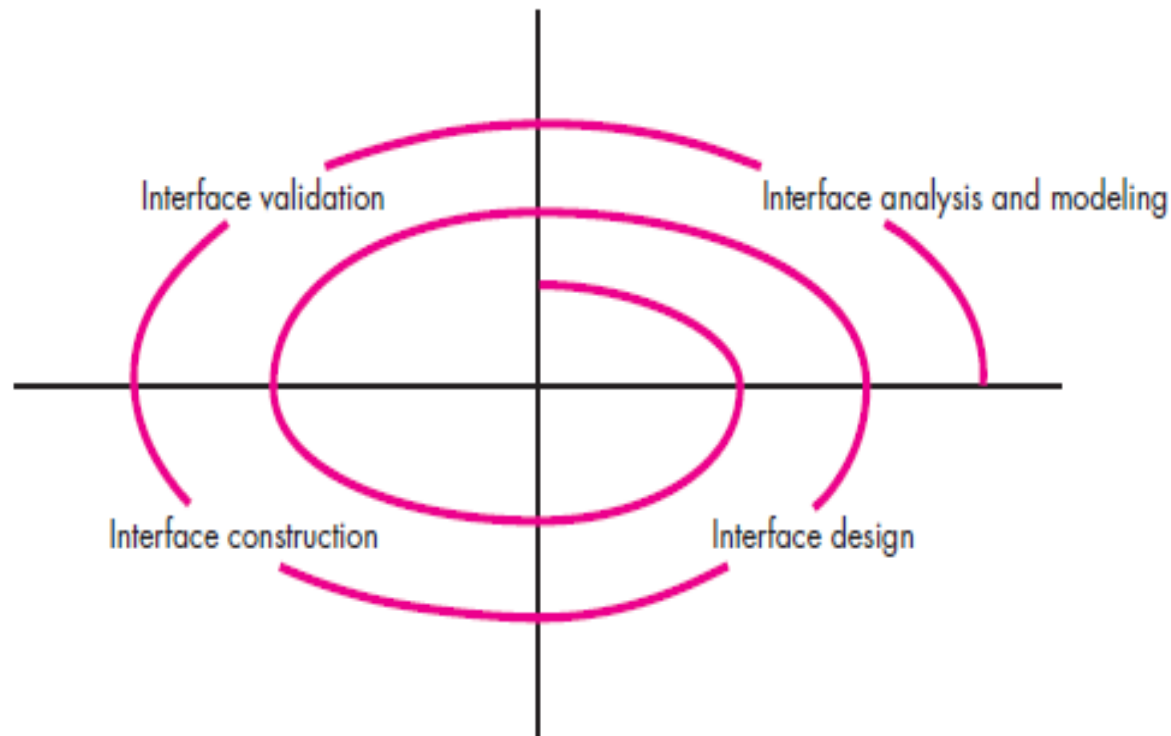
## **(2) Maintain consistency across a family of applications.**

- A set of applications (or products) should all implement the same design rules - 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.**

# The 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 design process

**(1) *Interface analysis*** – focuses on the **profile of the users** who will interact with the system. Different **user categories** are **defined**.

- Once **general requirements** have been **defined**, a more **detailed *task analysis*** is conducted.

**(2) *Interface design*** – is to **define a set of interface objects and actions** that enable a user to perform all defined tasks to **meet goals** defined.

# The user interface design process

**(3) Interface construction** - normally begins with the **creation of a prototype** that enables usage scenarios to be evaluated.

**(4) Interface validation** - focuses on

- (1) the ability of the interface to implement **every user task correctly, 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**
- (3) the **users' acceptance** of the **interface** as a useful tool in their work



# Interface Analysis

- **understand the problem before you attempt to design a solution.**
- (1) the people (**end users**) who will interact with the system through the interface.
  - (2) the **tasks that end users must perform** to do their work,
  - (3) the **content that is presented** as part of the interface
  - (4) the **environment** in which these **tasks will be conducted**

# Interface Analysis Steps

## (1) User Analysis

- **User Interviews.**
- **Sales input** - Sales people meet with users on a regular basis and can gather information.
- **Marketing input.**
- **Support input** - Support staff talks with users on a daily basis.

# Interface Analysis Steps

## (2) Task Analysis and Modeling

- In task analysis and modeling information about the **tasks and subtasks** performed by the user , their **workflow and hierarchy** is to be gathered.
1. **Use cases.**
  2. **Task elaboration** - stepwise elaboration (also called functional decomposition or stepwise refinement) for software to accomplish some desired function.

# Interface Analysis Steps

**3. Object elaboration:** Examine use case and other information obtained from the user.

**4. Workflow analysis** - When a number of different users, makes use of a user interface workflow analysis becomes important.

Eg: Pharmaceutical Company

**5. Hierarchical representation** - Once workflow has been established, a task hierarchy can be defined for each user type.

# Interface Analysis Steps

- **User task: *Requests that a prescription be refilled***
- *Provide identifying information.*
  - ✓ *Specify name.*
  - ✓ *Specify userid.*
  - ✓ *Specify PIN and password.*
- *Specify prescription number.*
- *Specify date refill is required.*

# Interface Analysis Steps

**(3) Analysis of Display Content** - the presentation of a variety of different types of content.

- For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).

**(4) Analysis of the Work Environment**

# Interface Design Steps

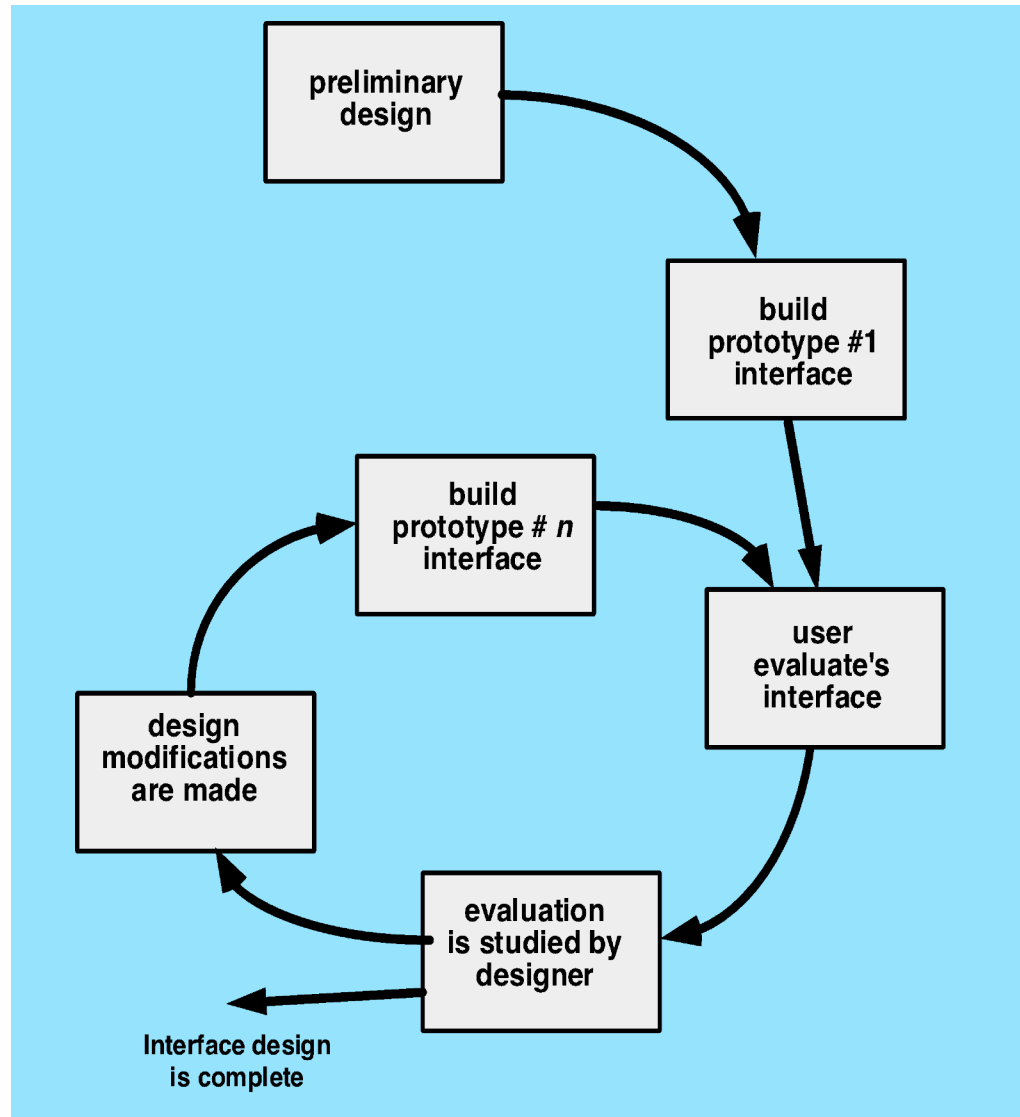
- Using **information developed during interface analysis**, define interface objects and actions (operations).
- **Define events (user actions) that will cause the state of the user interface to change.** Model this behavior.
- Depict each **interface state as it will actually look to the end-user.**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

# Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility – for physically challenged users
- Internationalization



# Design Evaluation Cycle



# Design with Reuse

- Building software from reusable components.

# Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems
- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on *systematic reuse*

# Reuse-based software engineering

- **Application system reuse**

- The whole of an application system may be reused either by incorporating it without change into other systems or by configuring the application for different customers.
- application families that have a common architecture, but which are tailored for specific customers, may be developed.

# Reuse-based software engineering

- **Component reuse**

- Components of an application from sub-systems to single objects may be reused

- **Object and Function reuse**

- Software components that implement a single function, such as a mathematical function, or an object class may be reused

# Benefits of reuse

- Increased reliability
  - Components exercised in working systems
- Reduced process risk
  - Less uncertainty in development costs
- Effective use of specialists
  - Reuse components instead of people
- Standards compliance
  - Embed standards in reusable components
- Accelerated development
  - Avoid original development and hence speed-up production

# Component-based software engineering

- Component-based software engineering (CBSE) is an **approach to software development that relies on reuse**
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- **Components are more abstract than object classes** and can be considered to be stand-alone service providers

# Component-based software engineering

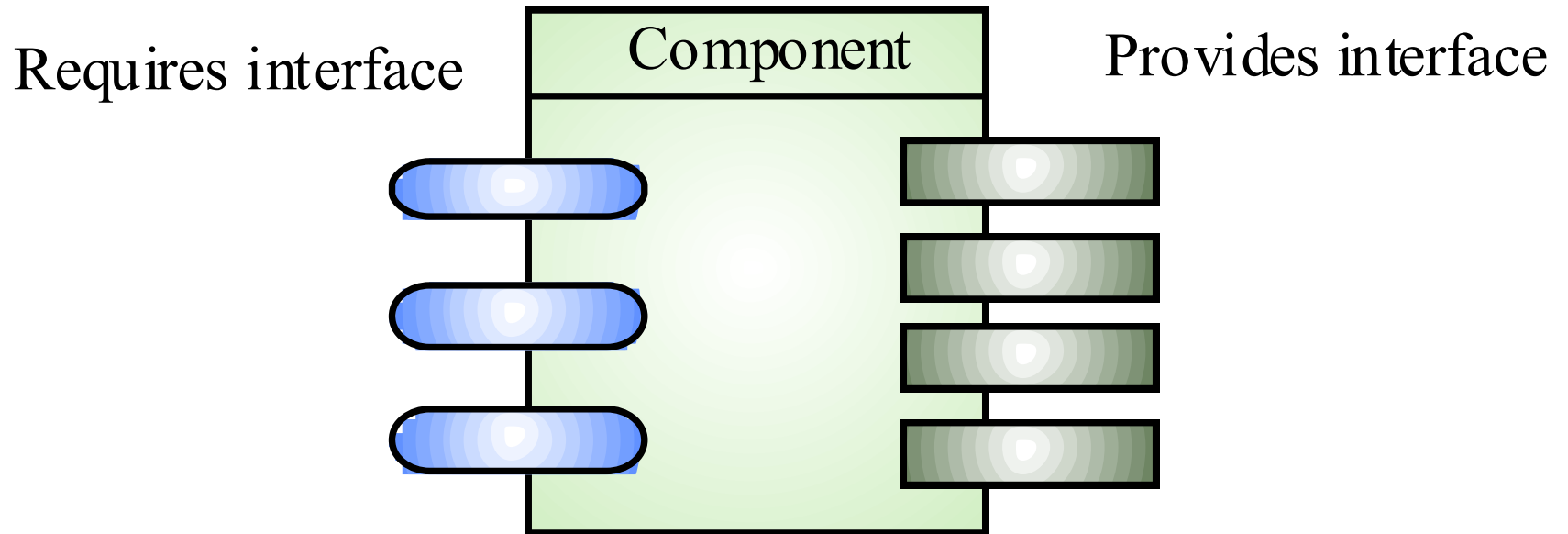
- Components can range in size from simple functions to entire application systems



# Key Goals of CBSE

- Save time and money when building large and complex systems
- Enhance the software quality
- Detect defects within the systems
- Improved efficiency
- Minimized delivery

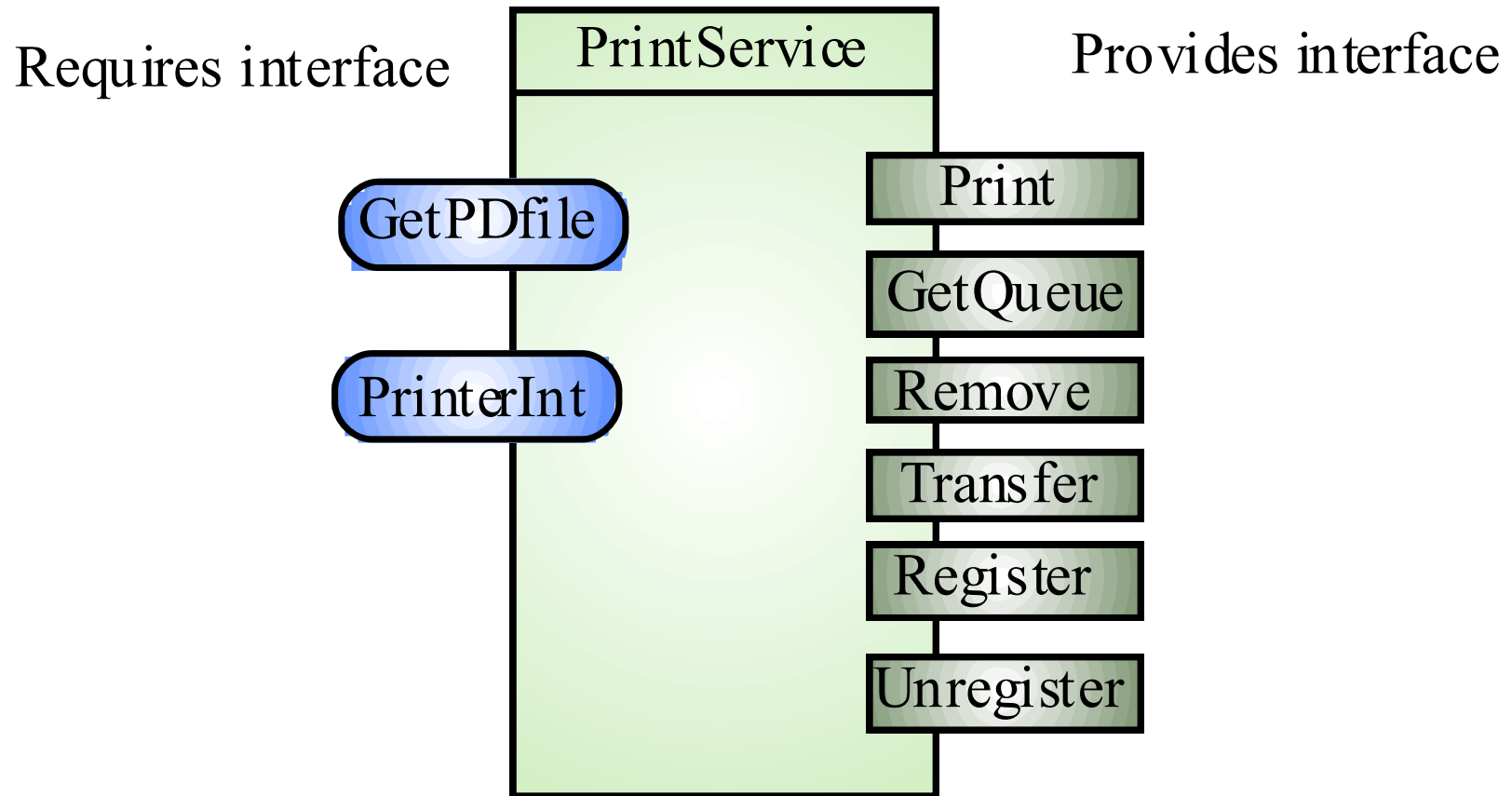
# Component interfaces



# Component Interfaces

- These interfaces reflect the services that the component provides and the services that the component requires to operate correctly
- **Provides interface**
  - Defines the services that are provided by the component to other components
- **Requires interface**
  - Defines the services that specifies what services must be made available for the component to execute as specified

# Example: Printing services component



# CBSE processes

- CBSE processes are software processes that support component-based software engineering.
- They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.

# CBSE processes

- There are two types:

## **1. Development for reuse**

- This process is concerned with **developing components or services that will be reused** in other applications.
- It usually involves generalizing existing components.

## **2. Development with reuse**

- This is the process of **developing new applications using existing components and services.**

# CBSE processes - **Development for reuse**

- CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system.

# CBSE processes - Development with reuse

