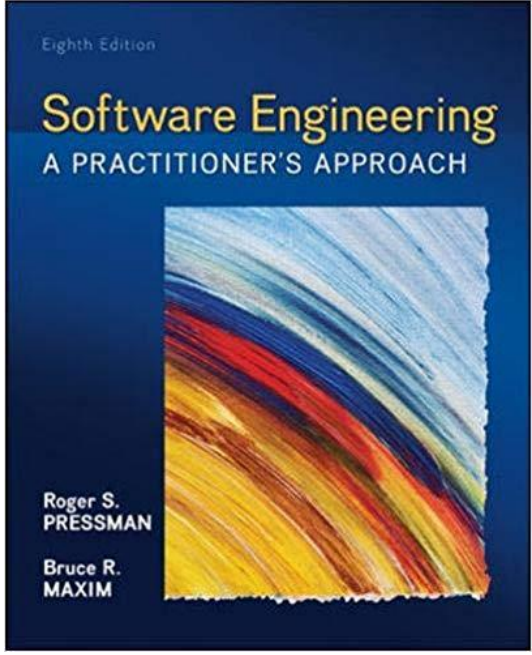



# Software Design

CS46 Software Engineering

*Dr. Shilpa chaudhari*

*Department of Computer Science and Engineering*
*RIT, Bangalore*

# Outline of Session-3

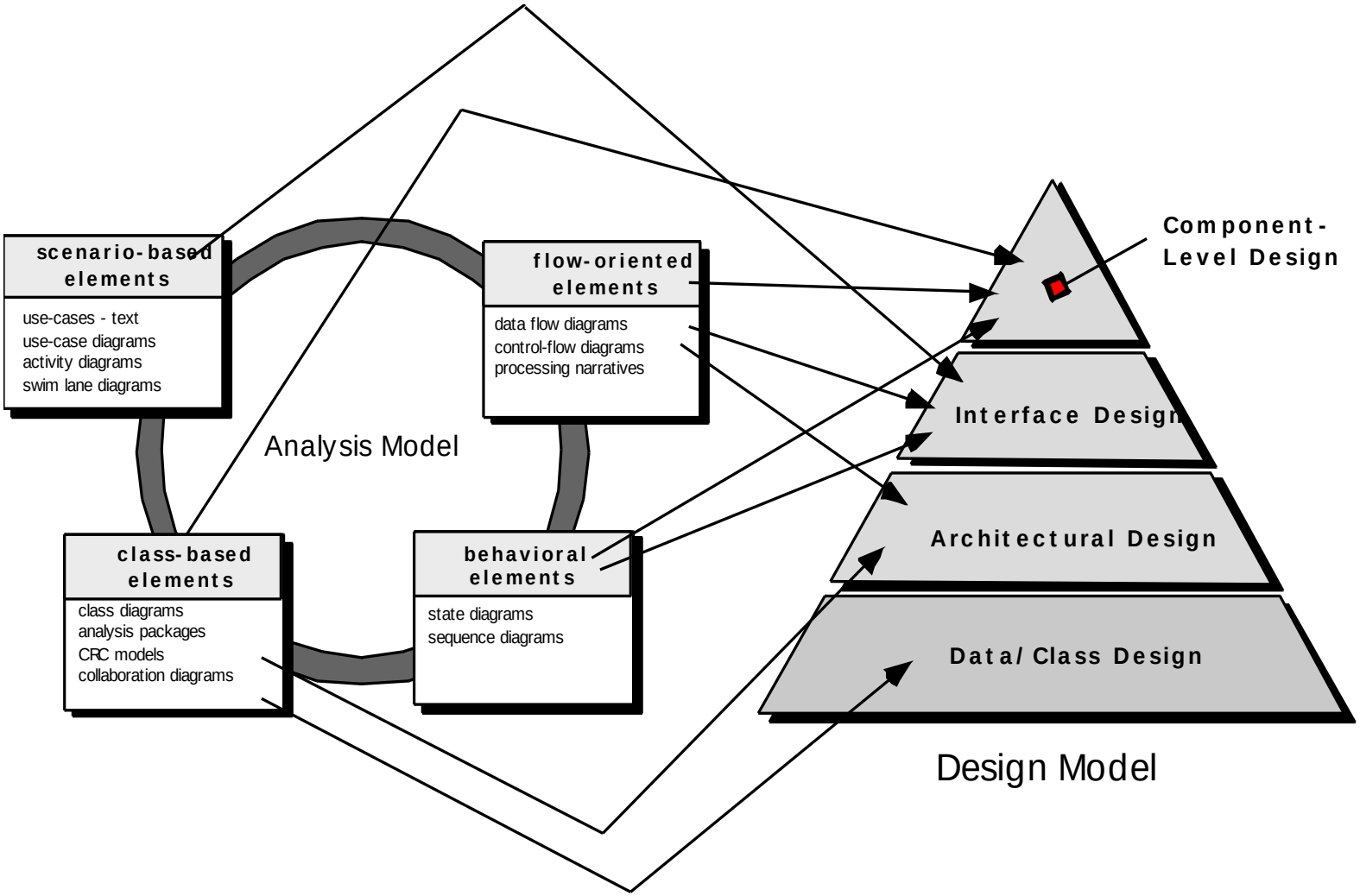
- **Design Concepts:** The Design Process, Design Concepts, Design Model; →12.2, 12.3, 12.4
- **Architectural Design:** Software Architecture, Architectural Genres, Architectural Styles, Architectural Considerations, Architectural Decisions, Architectural Design; → 13.1, 13.2, 13.3, 13.4, 13.5, 13.6
- **User Interface:** The Golden Rules of User Interface Analysis and Design, WebApp and Mobile Interface Design; → 15.1, 15.2,15.5
- **Applying design modeling by taking requirement specification from Unit-2**

# Design Concepts

# Software Engineering Design

- Data/Class design – transforms analysis classes into implementation classes and data structures
- Architectural design – defines relationships among the major software structural elements
- Interface design – defines how software elements, hardware elements, and end-users communicate
- Component-level design – transforms structural elements into procedural descriptions of software components

# Analysis Model -> Design Model



# 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.
- 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 Attributes: FURPS—functionality, usability, reliability, performance, and supportability/*maintainability*

# Quality Guidelines

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

# Design Principles

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

# Evolution of Software Design

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down “structured” manner
- Object-oriented approach to design derivation
- Recent emphasis in software design has been on software architecture  and the design patterns that can be used to implement software architectures and lower levels of design abstractions
- Growing emphasis on achieving more effective modularity and architectural structure in the designs
  - aspect-oriented methods
  - model-driven development
  - test-driven development

# Evolution of Software Design

- Common characteristics:
  - a mechanism for the translation of the requirements model into a design representation
  - a notation for representing functional components and their interfaces
  - heuristics for refinement and partitioning
  - guidelines for quality assessment.
- Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design

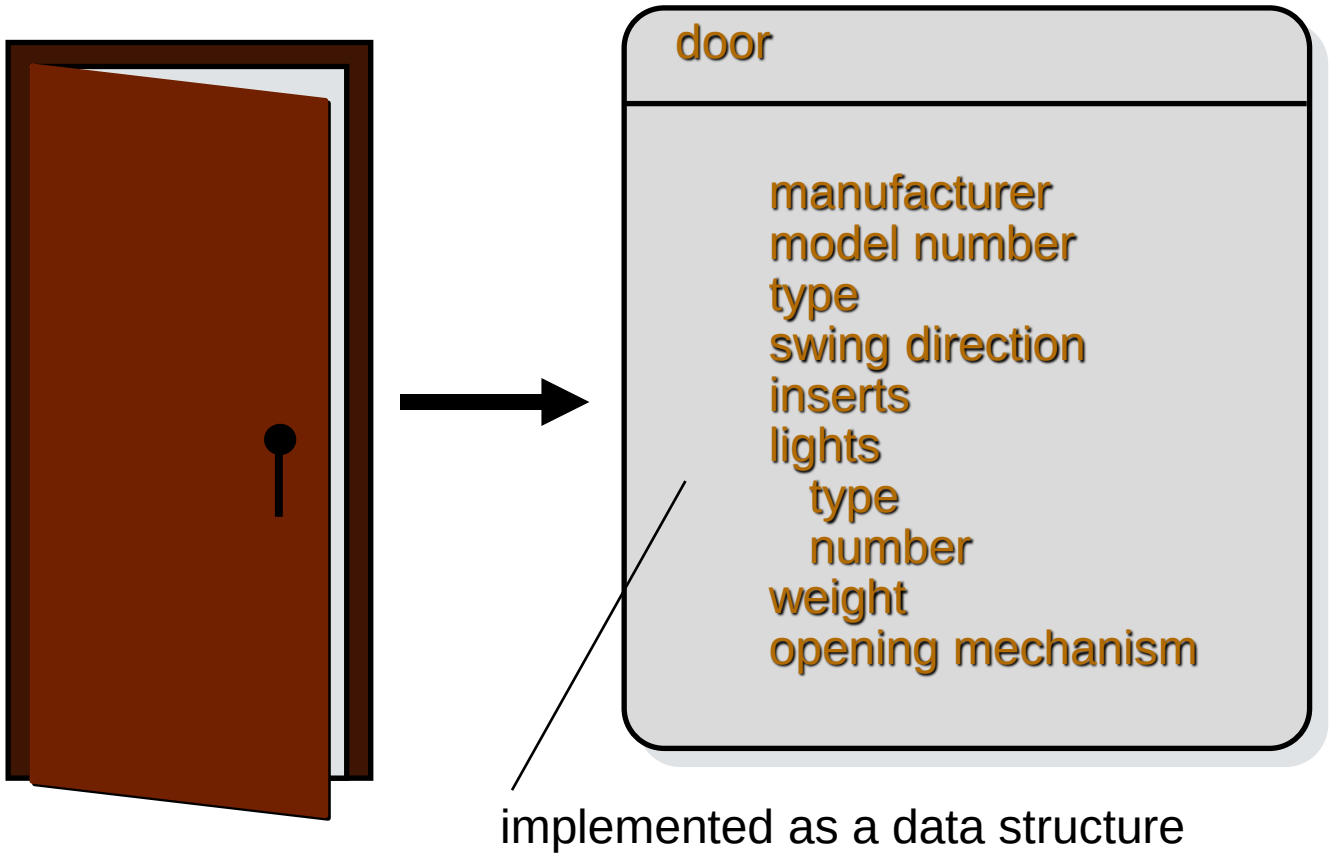
# Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   - Translate analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select design patterns for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create behavioral model of the interface.
   - Define interface objects, control mechanisms.
   - Review the interface design and revise as required.
7. Conduct component-level design.
   - Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.
   - Review each component and correct all errors uncovered.
8. Develop a deployment model.

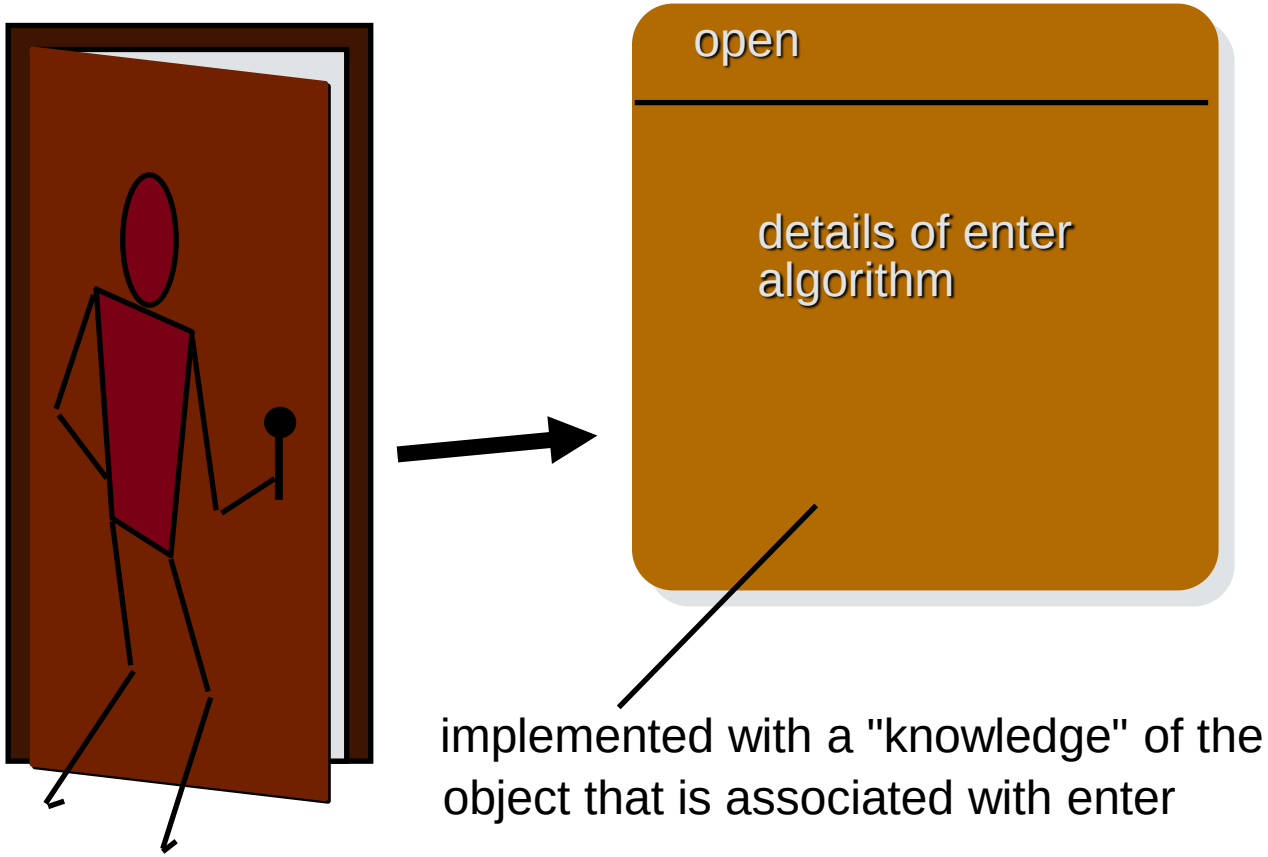
# Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts
- Design Classes—provide design detail that will enable analysis classes to be implemented

# Data Abstraction



# Procedural Abstraction



# Architecture

- "The overall structure of the software and the ways in which that structure provides conceptual integrity for a system" …specification of following properties
- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

# Architecture

- Represent architectural design using one or more of a number of different models based on the specification of previous properties using *architectural description languages (ADLs)*
- **Structural models** represent architecture as an organized collection of program components
- **Framework models** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- **Dynamic models** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events
- **Process models** focus on the design of the business or technical process that the system must accommodate
- **Functional models** can be used to represent the functional hierarchy of a system

# Patterns

- A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns”
- The intent of each design pattern is to provide a description that enables a designer to determine

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

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

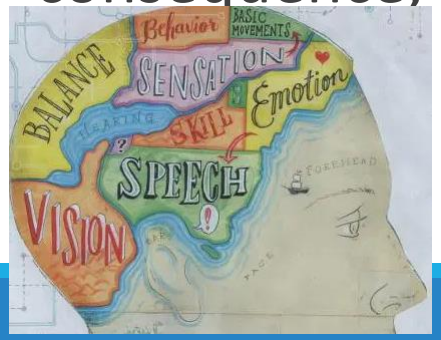
  (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

# Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- divide-and-conquer strategy
- related design concepts: modularity, aspects, functional independence, and refinement

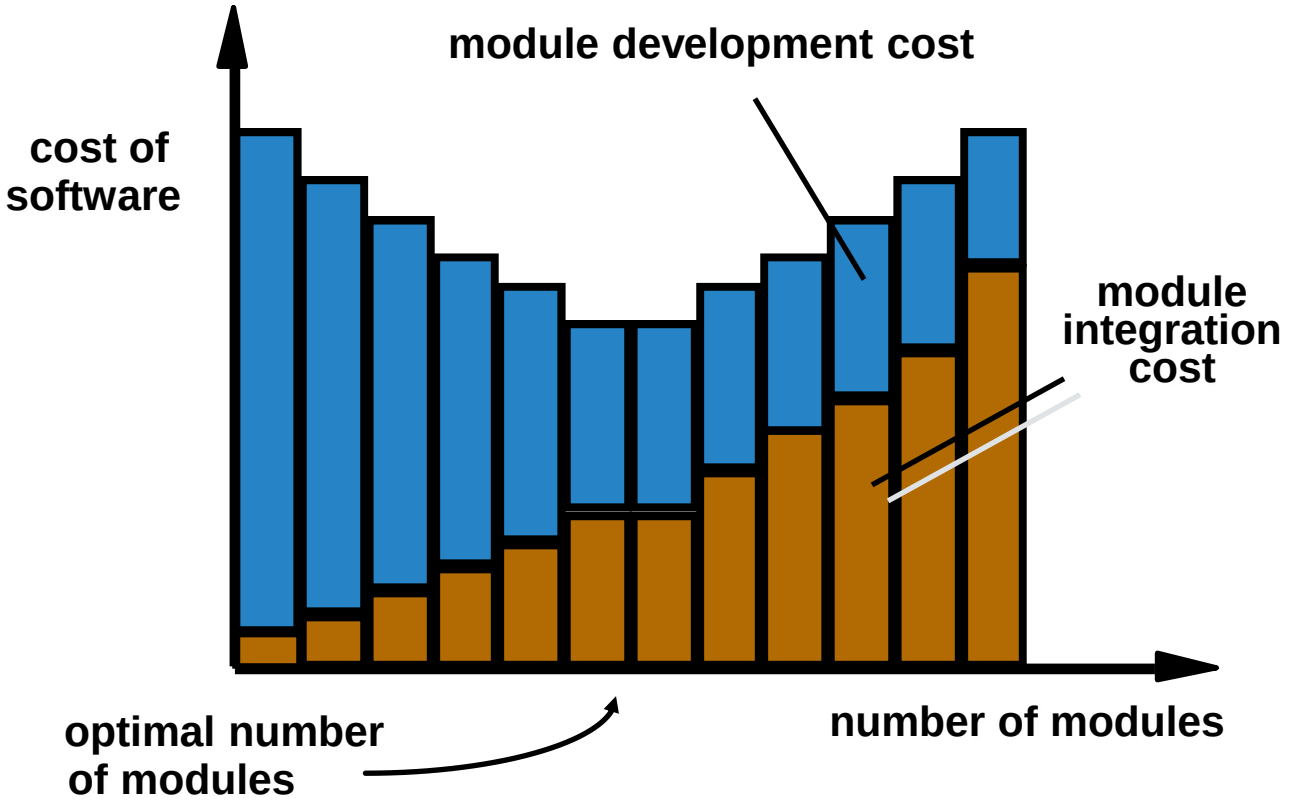
# Modularity

- "modularity is the single attribute of software that allows a program to be intellectually manageable".
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

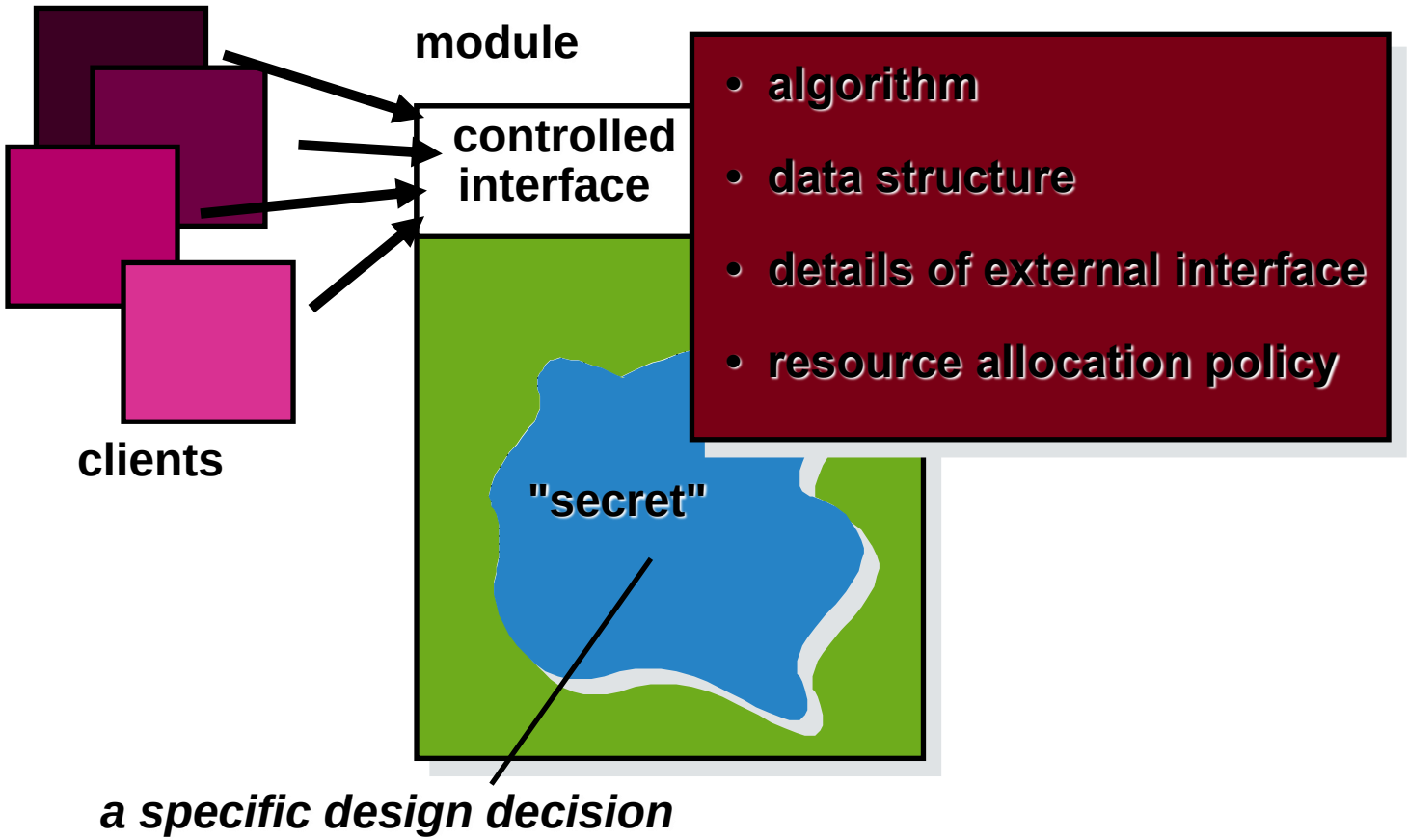


# Modularity: Trade-offs

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



# Information Hiding

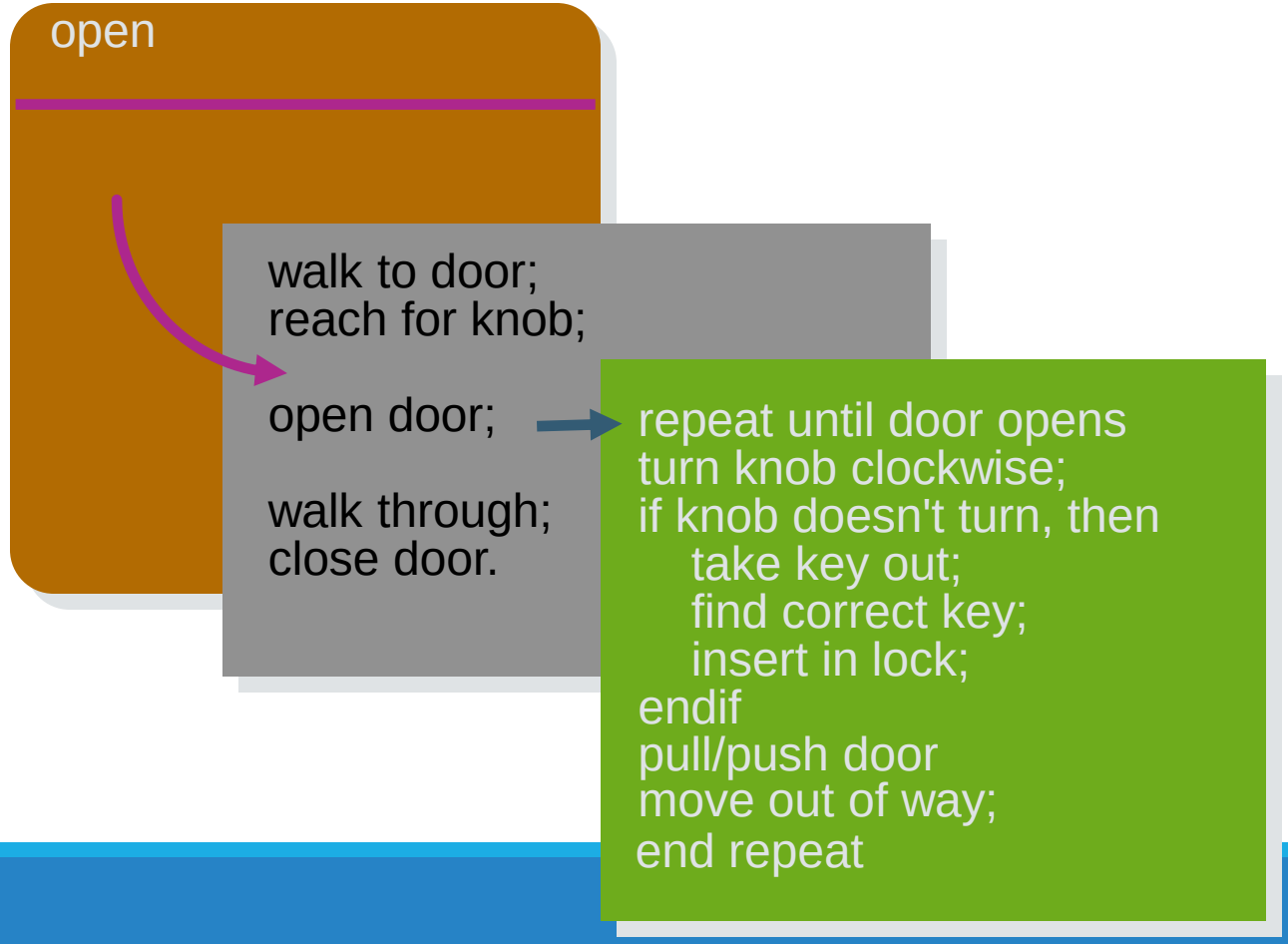


# Why Information Hiding?

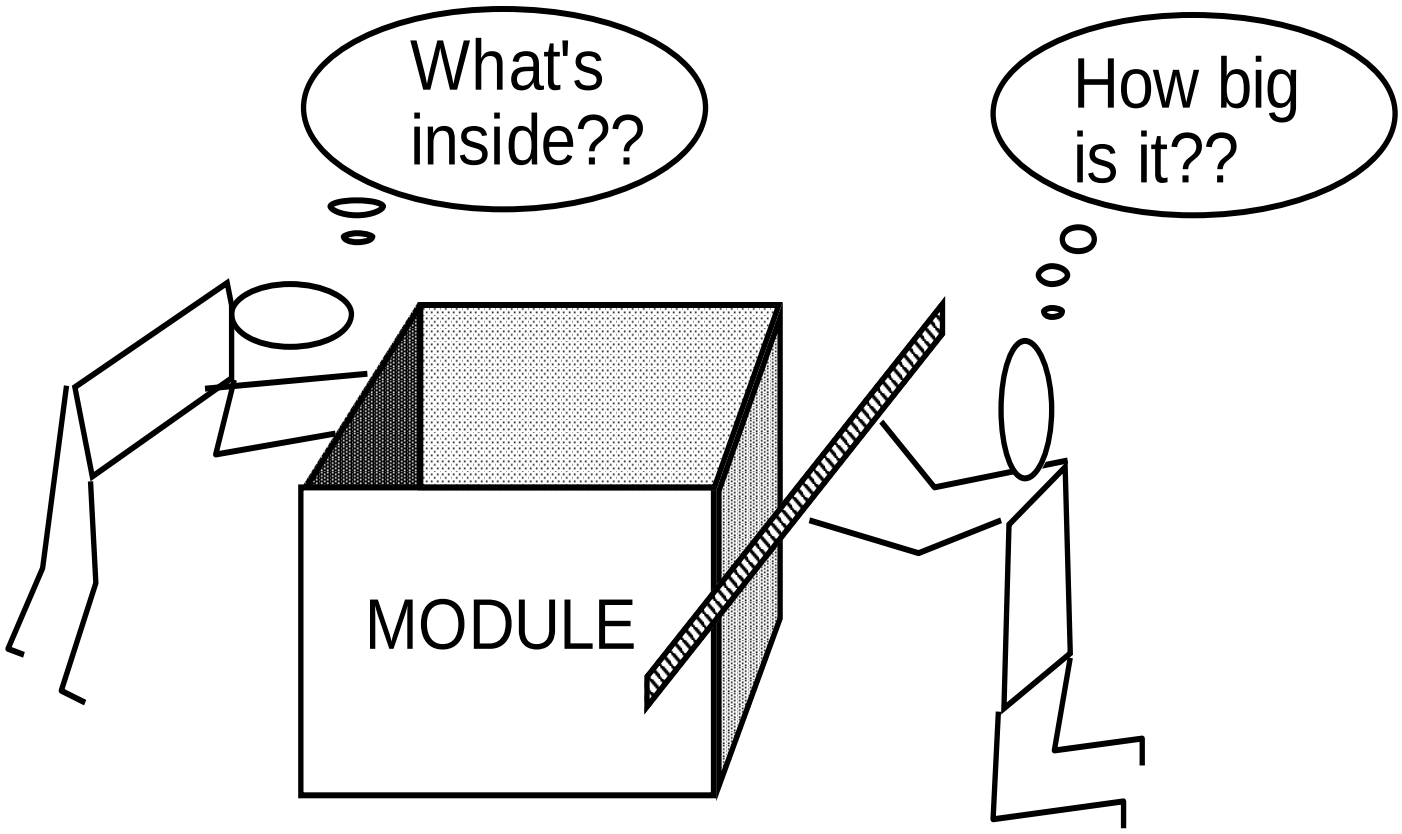
- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

# Stepwise Refinement

- Process of *elaboration*
- Abstraction and refinement are complementary



# Sizing Modules: Two Views



# Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

# Aspects

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.
  - include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts

# Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

# 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
  - or any other design failure that can be corrected to yield a better design.

# OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

# Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

# Design Class Characteristics

- Complete - includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

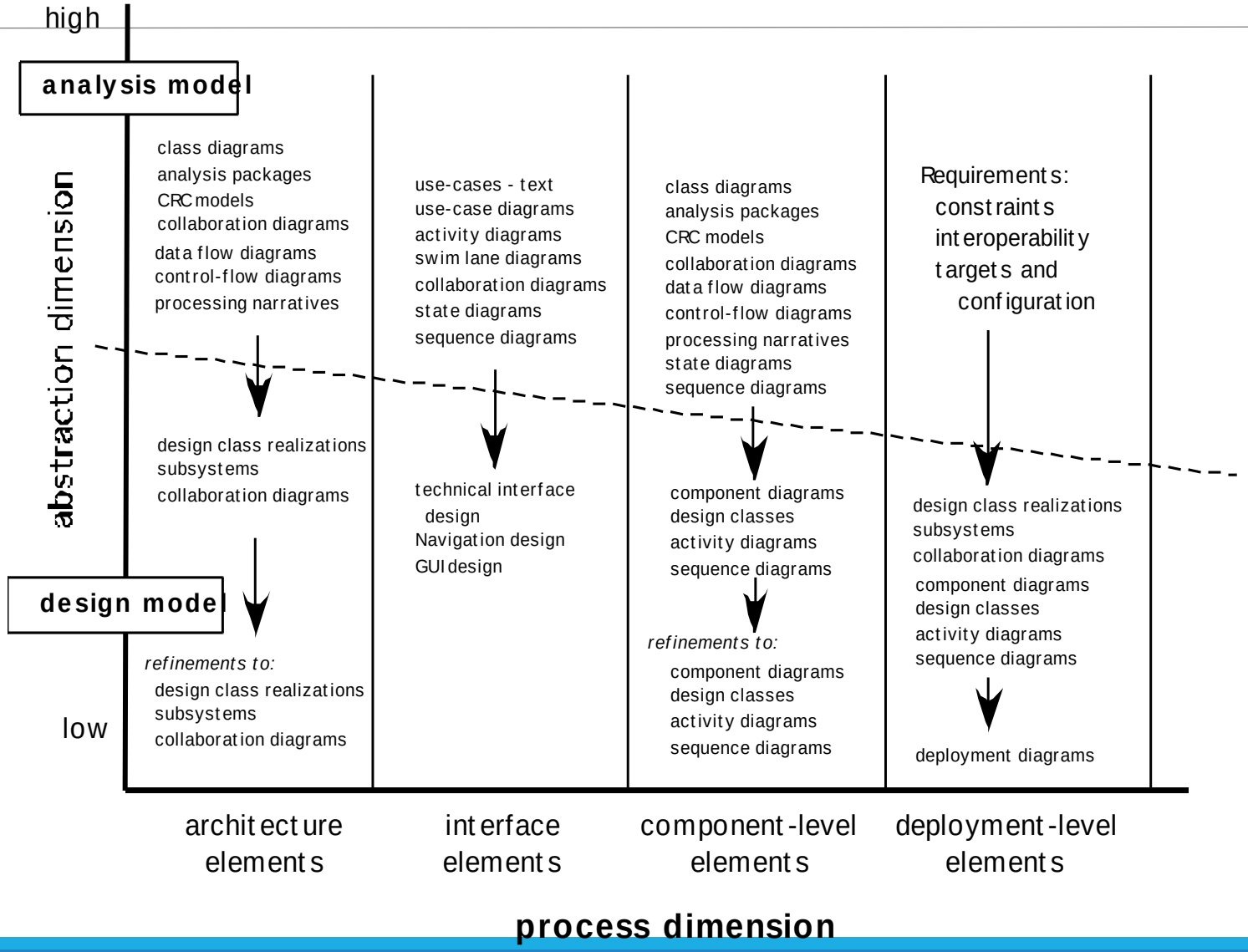
# Dependency Inversion

- *High-level modules (classes) should not depend [directly] upon low-level modules in* hierarchical architecture
- *Both should depend on abstractions*
  - *Abstractions should not depend on details*
  - *Details should depend on abstractions*

# Design for Test

- whether software design or test case design should come first
  - test-driven development (TDD) write tests before implementing any other code
  - if design comes first, then the design (and code) must be developed with locations in the detailed design

# The Design Model



# Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (Chapters 9 and 12)
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

# Data Modeling

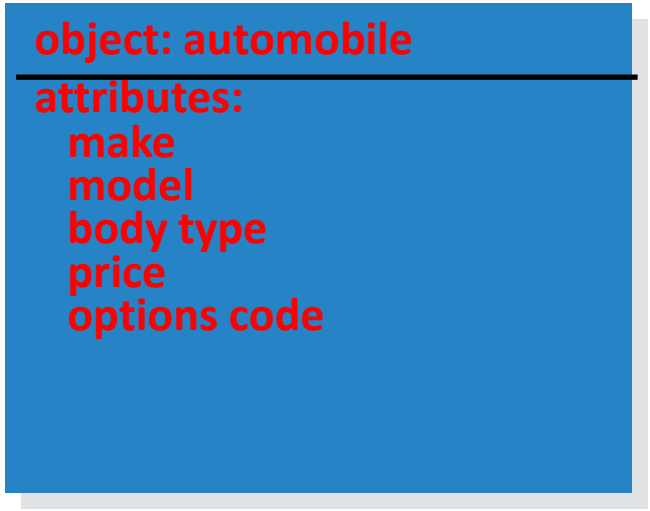
- examines data objects independently of processing
- focuses attention on the data domain
- creates a model at the customer's level of abstraction
- indicates how data objects relate to one another

# What is a Data Object?

- a representation of almost any composite information that must be understood by software.
  - *composite information*—something that has a number of different properties or attributes
- can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

# Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object



# What is a Relationship?

- Data objects are connected to one another in different ways.
  - A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - A person *is insured to drive* a car
- The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.**
- Several instances of a relationship can exist
- Objects can be related in many different ways

# Architectural Elements

- The architectural model  is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns and styles

# Interface Elements

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaboration diagrams)

# Interface Elements

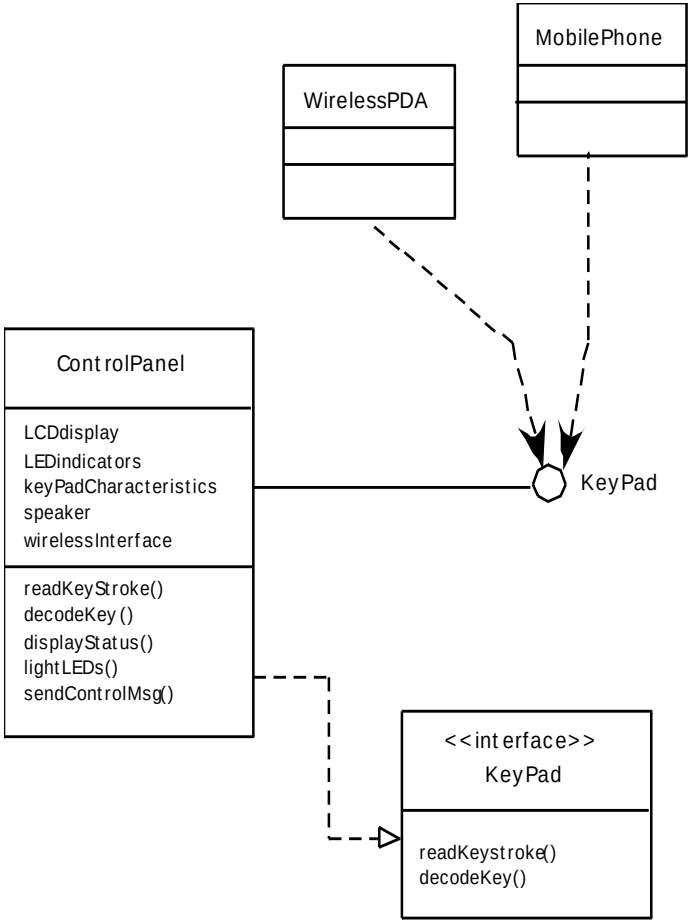
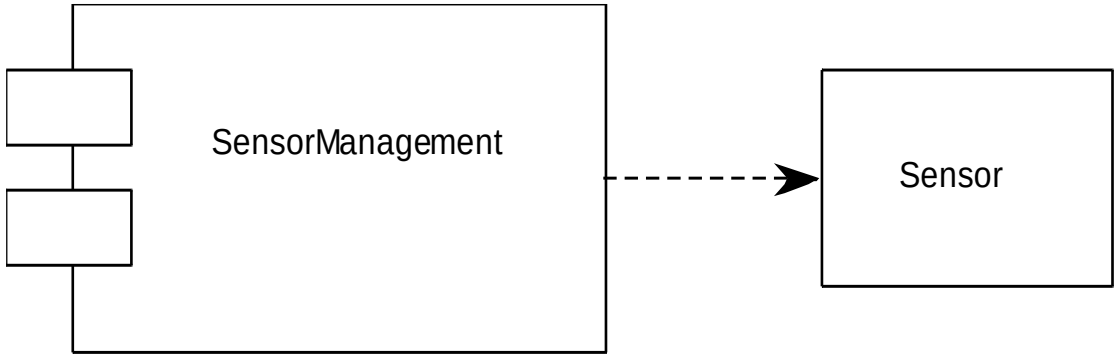



Figure 9.6 UML interface representation for **ControlPanel**

# Component Elements

- Describes the internal detail of each software component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations
- Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

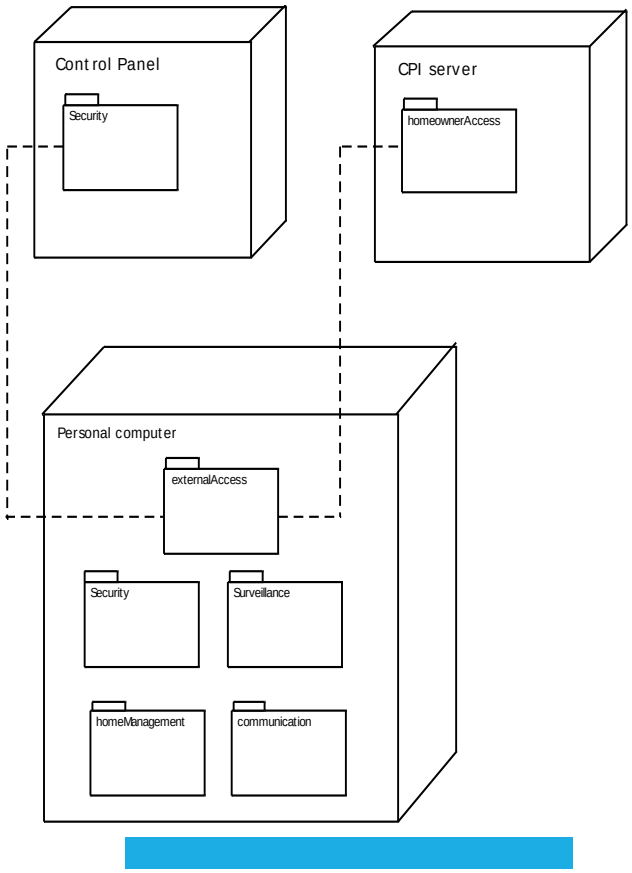
# Component Elements



# Deployment Elements

- Indicates how software functionality and subsystems will be allocated within the physical computing environment
- Modeled using UML deployment diagrams
- *Descriptor form* deployment diagrams show the computing environment but does not indicate configuration details
- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

# Deployment Elements



# 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.


47

# 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 mode of how the system is structured and how its components work together"


48

# Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive System*, [IEE00]
  - to establish a conceptual framework and vocabulary for use during the design of software architecture,
  - to provide detailed guidelines for representing an architectural description, and
  - to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a "a collection of products to document an architecture."
  - The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."


49

# Architectural Genres

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
  - Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.


50

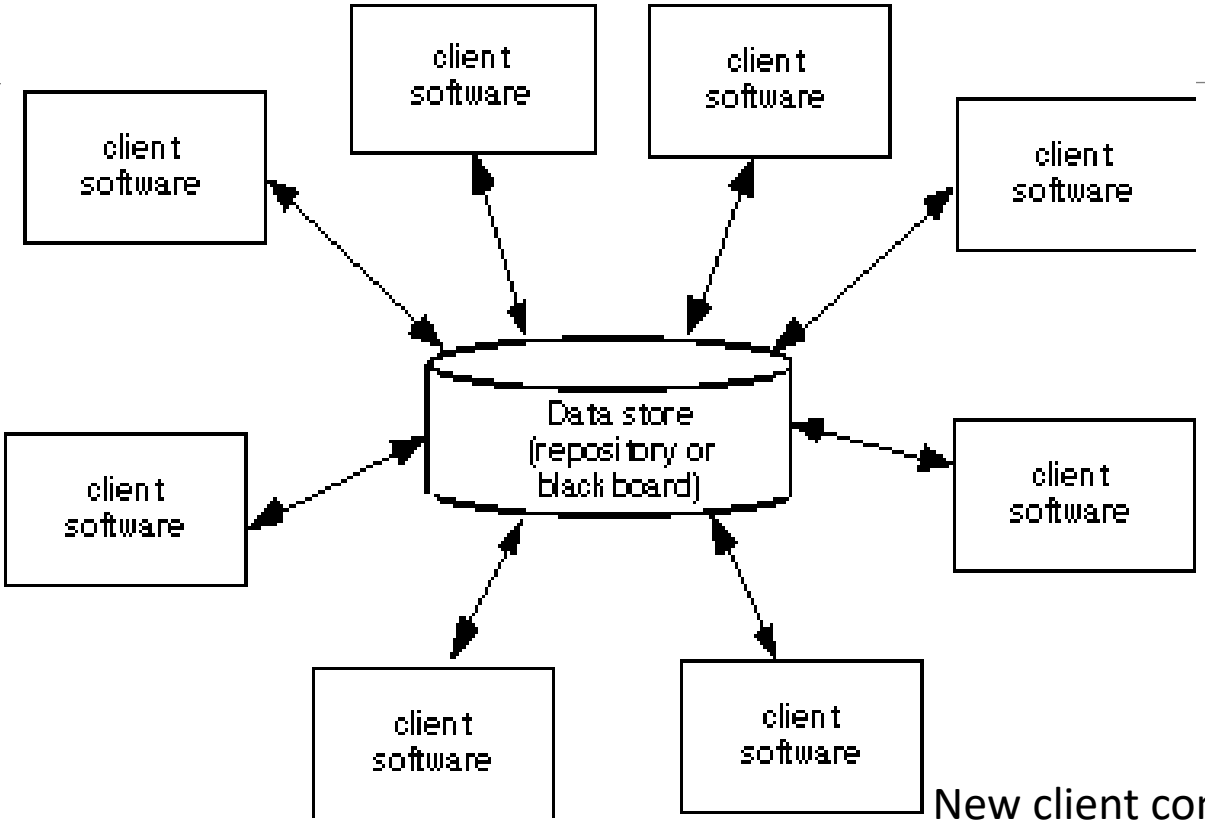
# Architectural Styles

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


51

# Data-Centered Architecture




New client component easily added

Existing component can be changes

Data passed among clients using the black-board mechanism

Client component independently execute process

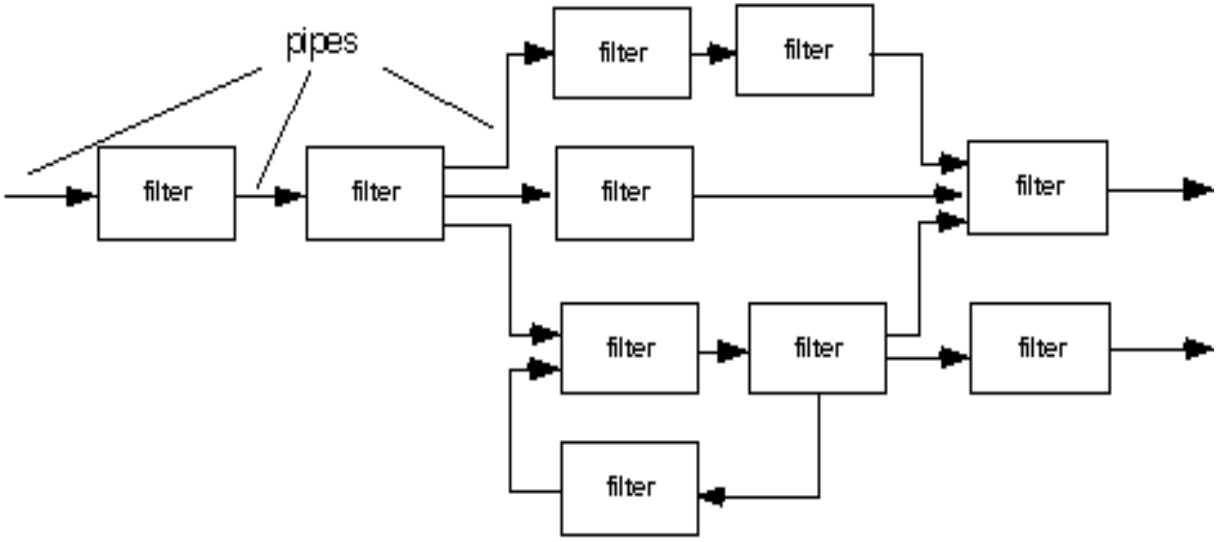


52

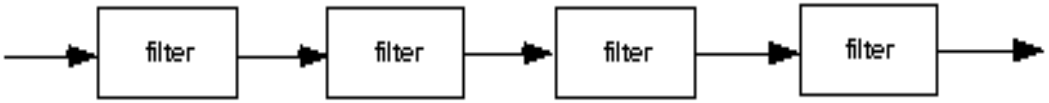
# Data Flow Architecture

Applied when input data are to be transferred through a series of computational or manipulative components into output data.

Filter = components, Pipes = transmit data from one component to next.




(a) pipes and filters

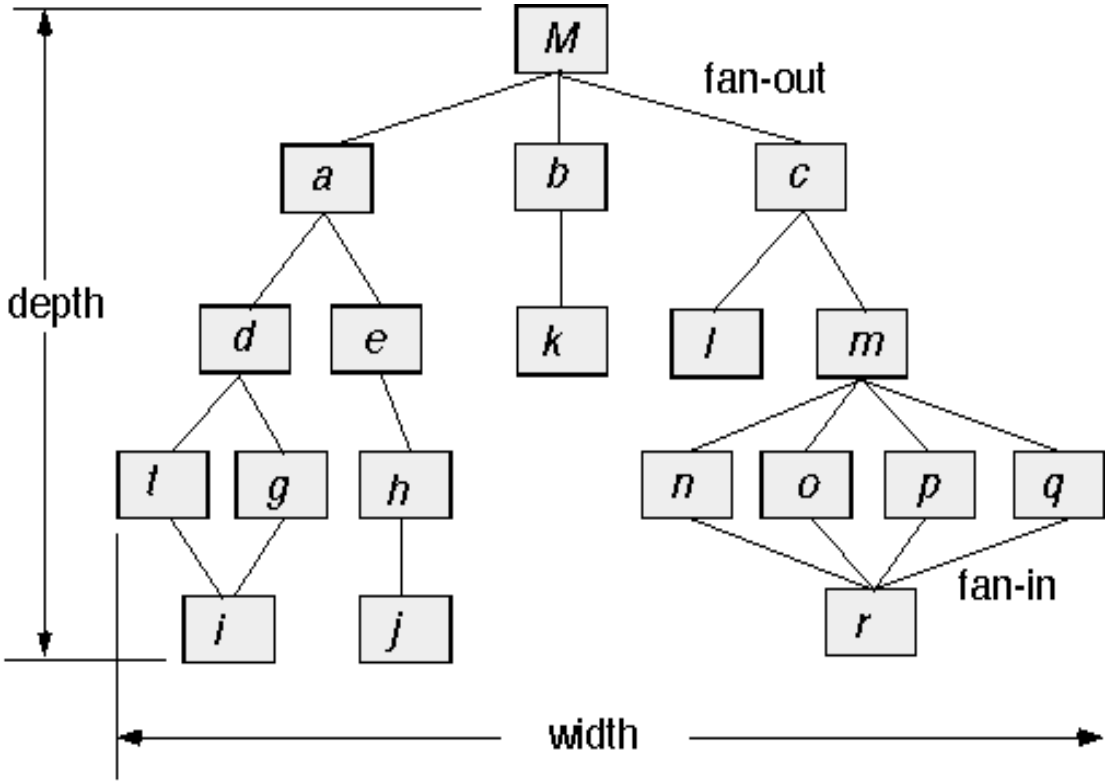



(b) batch sequential

# Call and Return Architecture

Main program.subprogram architectures: decomposes function into control hierarchy

Remote procedure call architectures: components are distributed across multiple computers on a network

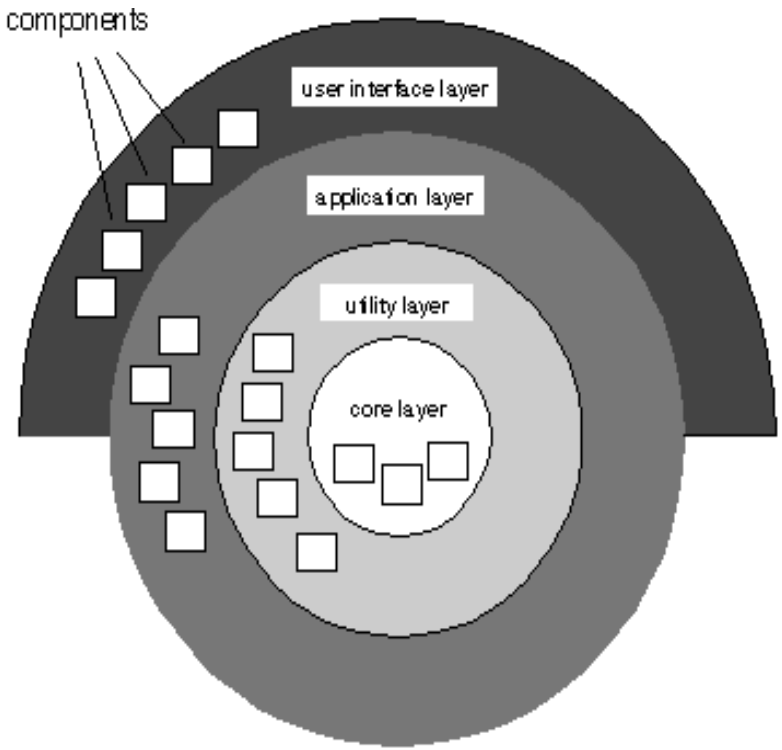





54

# Layered Architecture

Each layer accomplishing operations that progressively become closer to the machine instruction set






55

# 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.

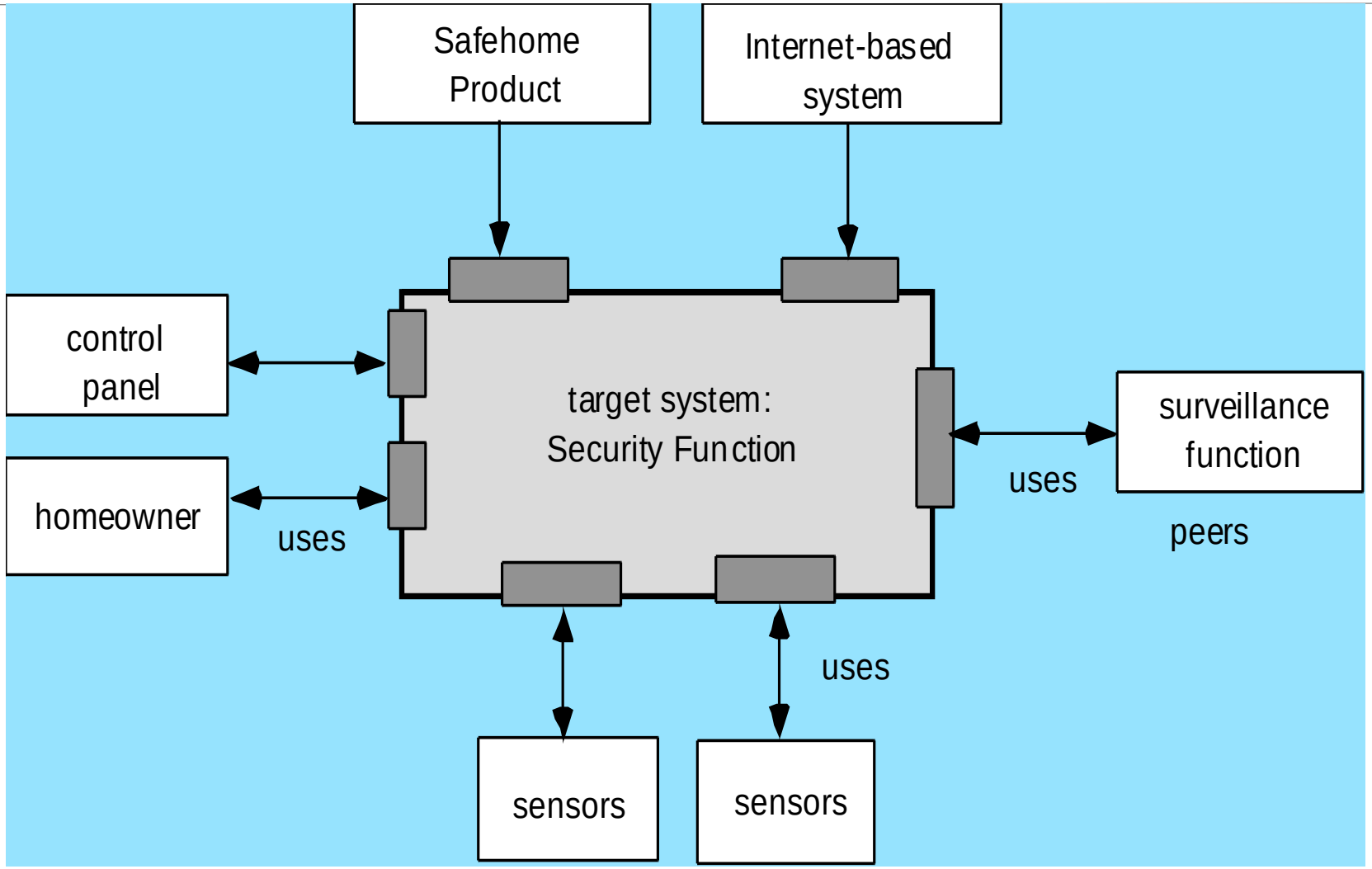

56

# 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


57

# Architectural Context






58

# Archetypes

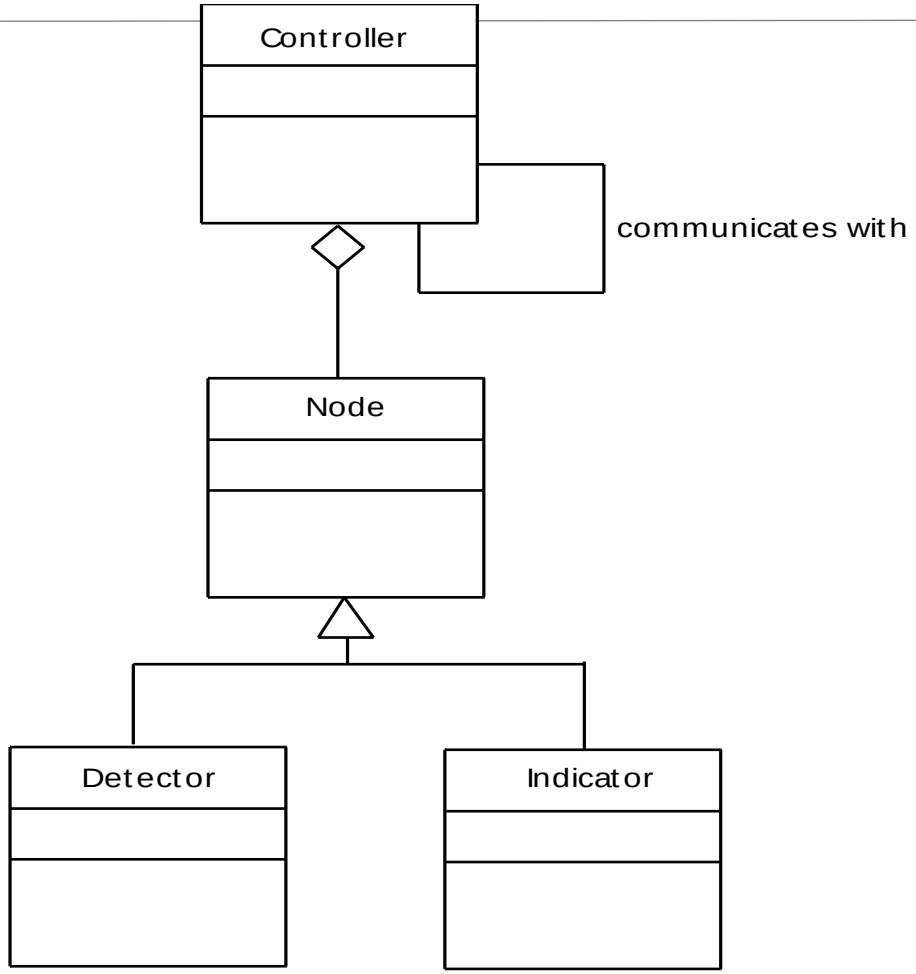
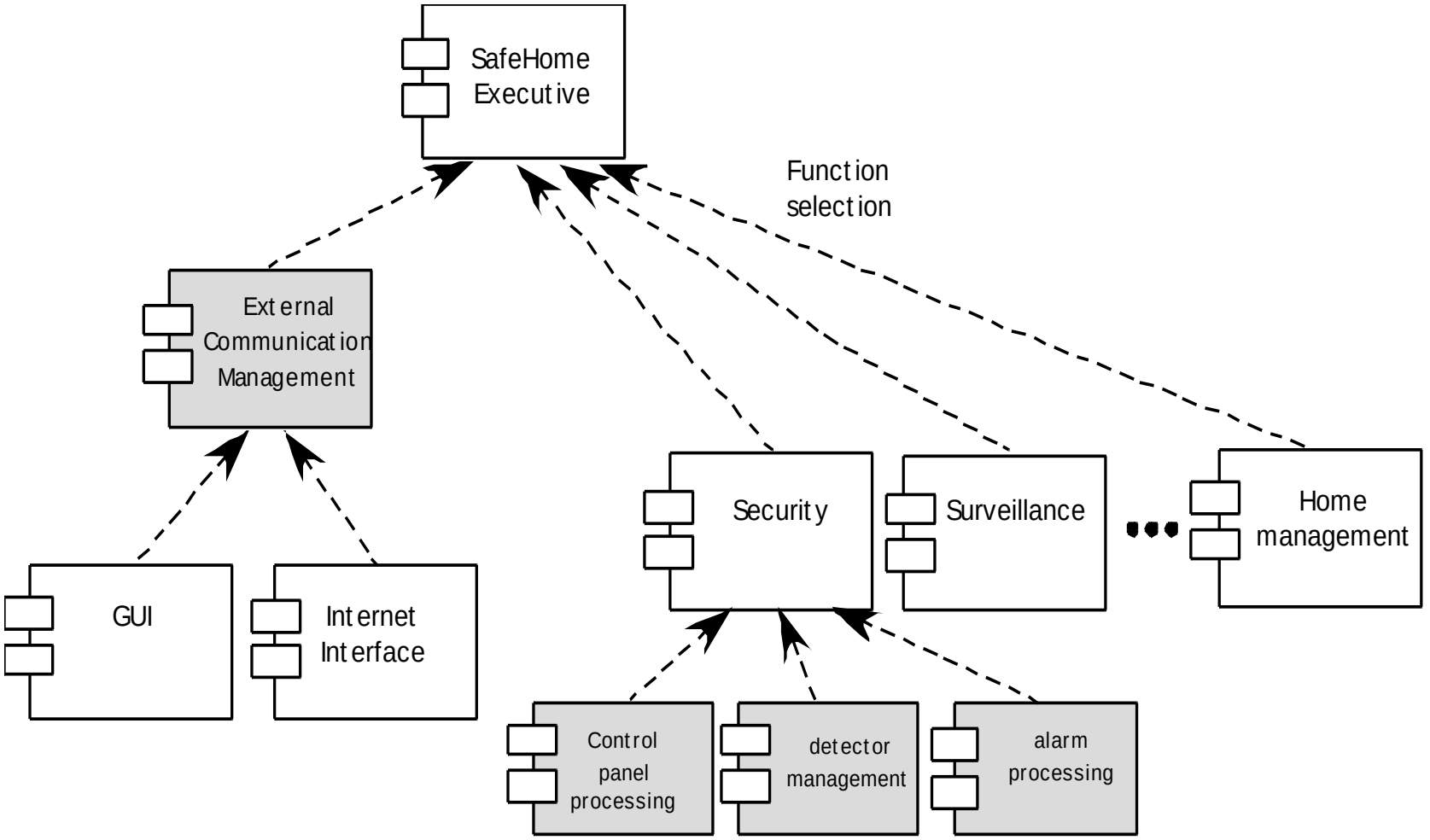



Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])



59

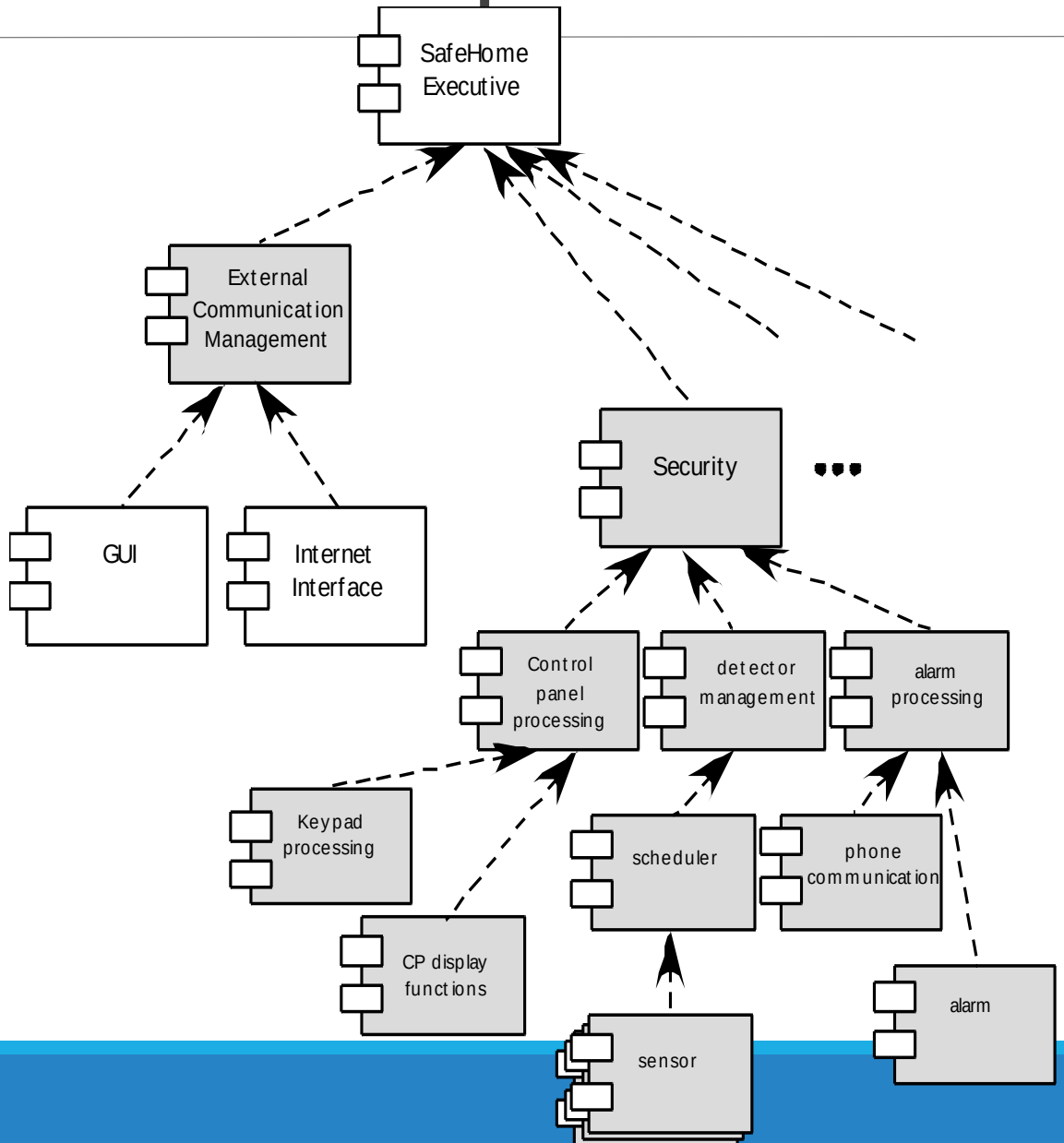
# Component Structure






60

# Refined Component Structure



# Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
   - module view
   - process view
   - data flow view
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.


62

# Architectural Complexity

- the overall complexity of a proposed architecture is assessed by considering the dependencies between components within the architecture
  - *Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
  - *Flow dependencies* represent dependence relationships between producers and consumers of resources.
  - *Constrained dependencies* represent constraints on the relative flow of control among a set of activities.


63

# ADL

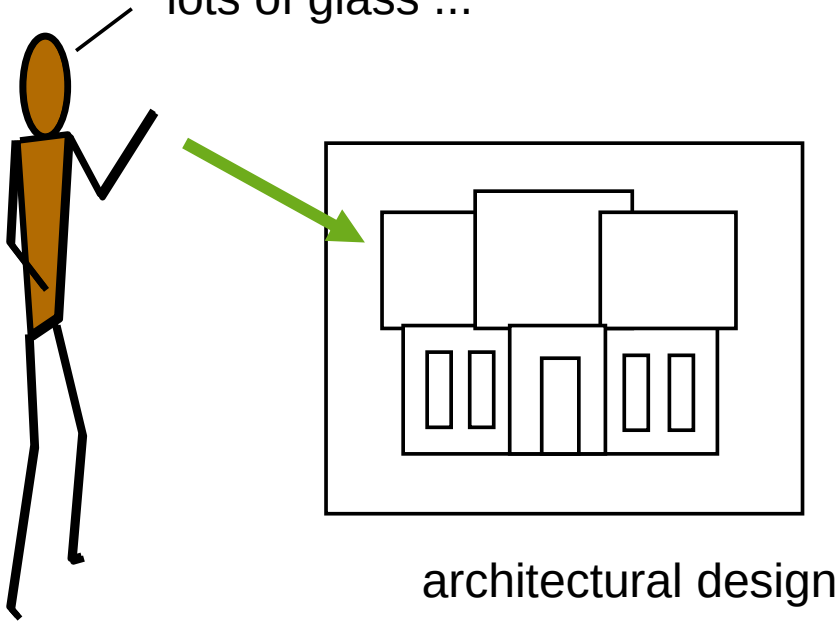
- *Architectural description language* (ADL) provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks and
  - represent interfaces (connection mechanisms) between components.


64

# An Architectural Design Method

***customer requirements***

"four bedrooms, three baths,
lots of glass ..."

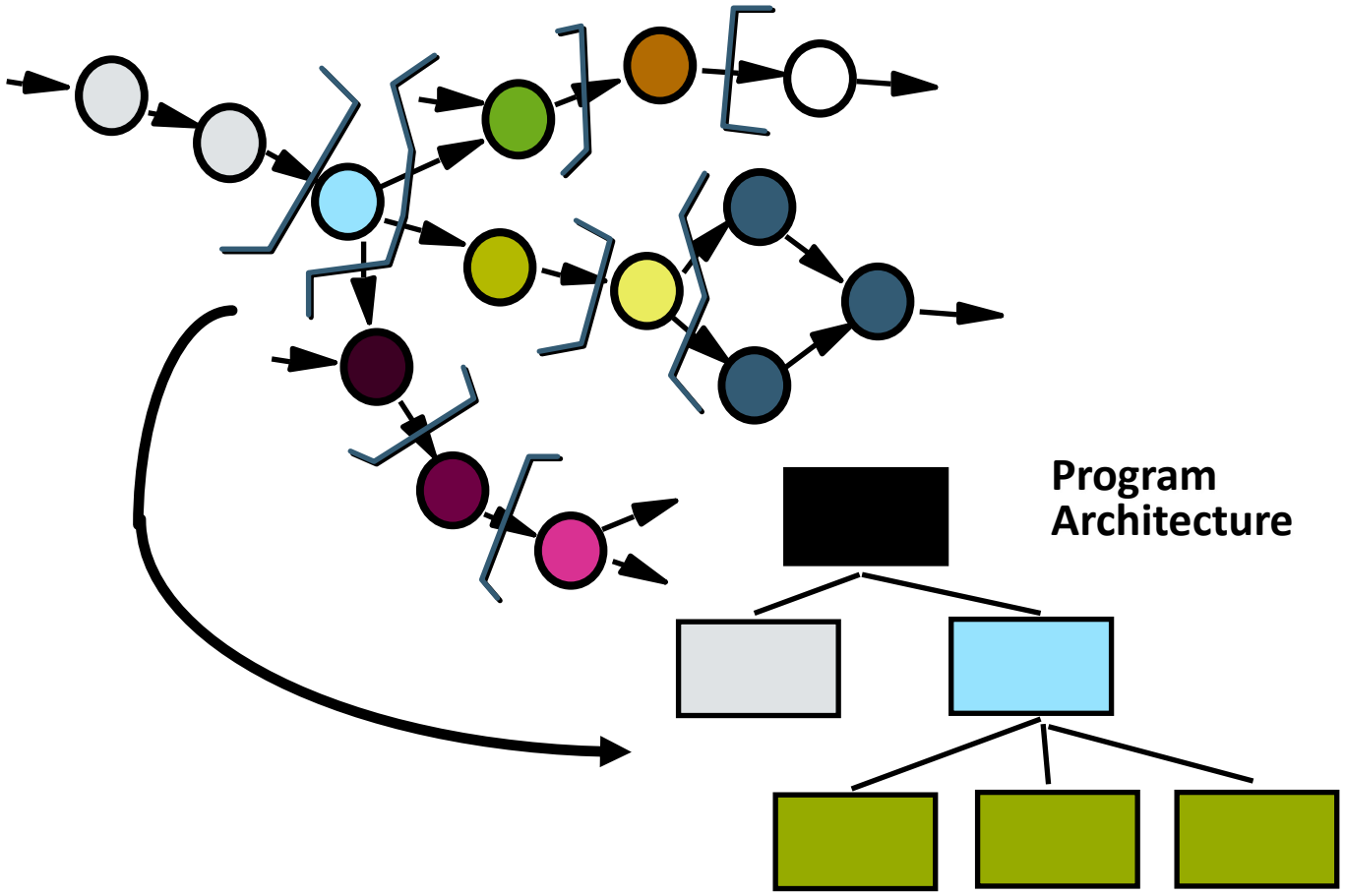


architectural design



65

# Deriving Program Architecture

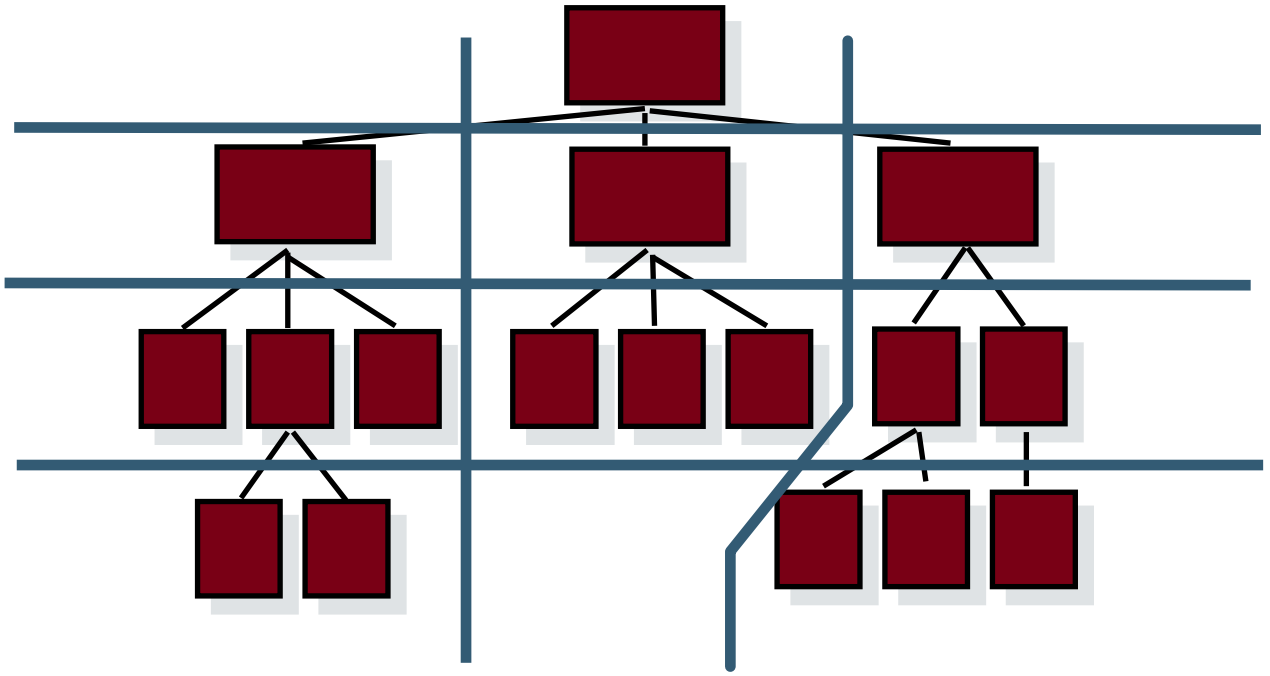





66

# Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required

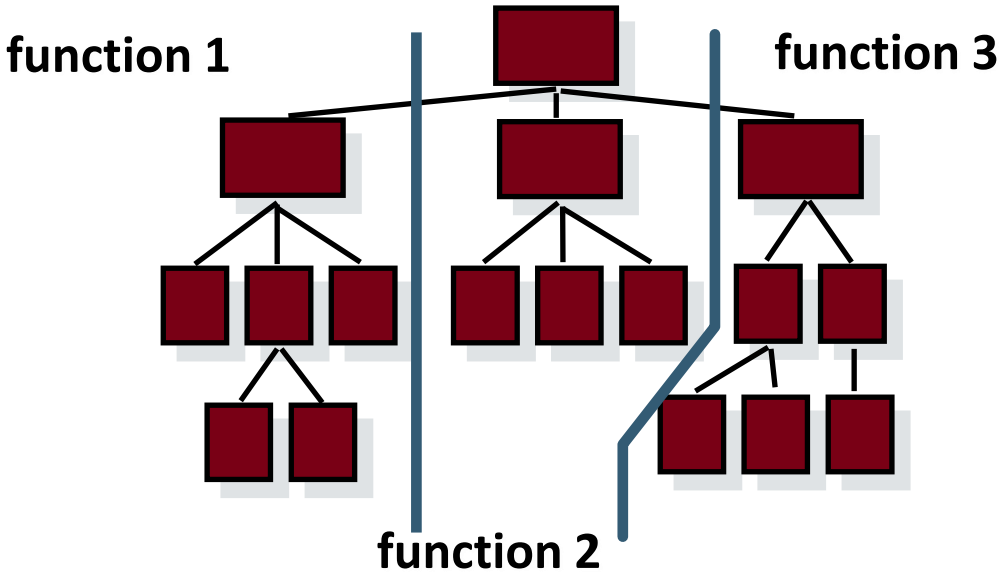




67

# Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions

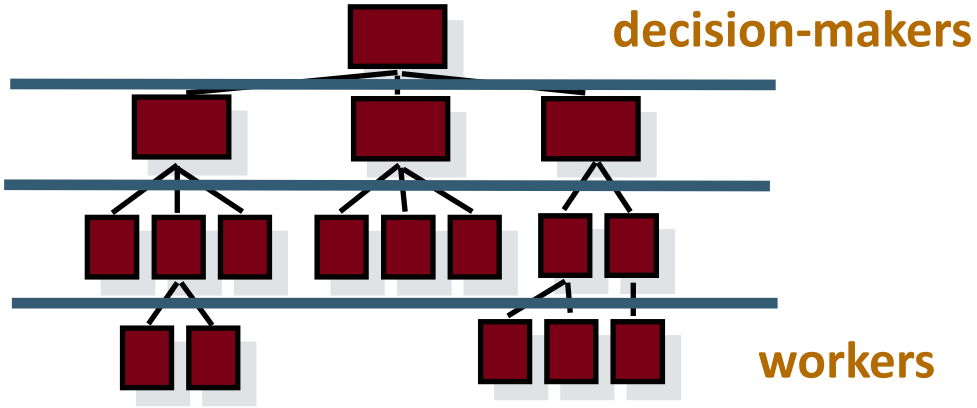




68

# Vertical Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture






69

# Why Partitioned Architecture?

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

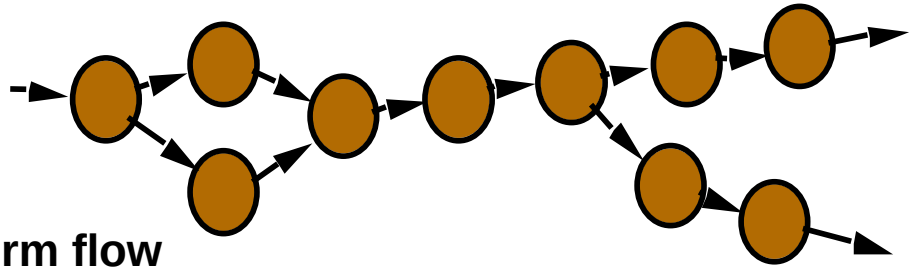

70

# Structured Design

- objective: to derive a program architecture that is partitioned
- approach:
  - a DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- notation:  structure chart

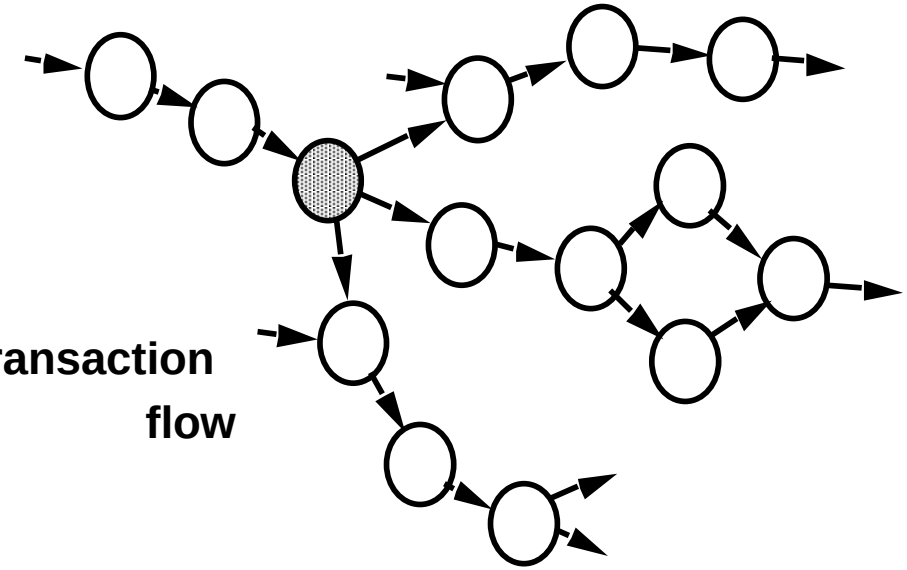

71

# Flow Characteristics




**Transform-oriented design**, processes are divided into *input and data preprocessing* functions, *data processing functions*, and output related functions

**Transaction-oriented design**, in this case the design consists of an *input module*, a *dispatcher module*, *transaction processing modules* one module for each type of transaction/command/or request

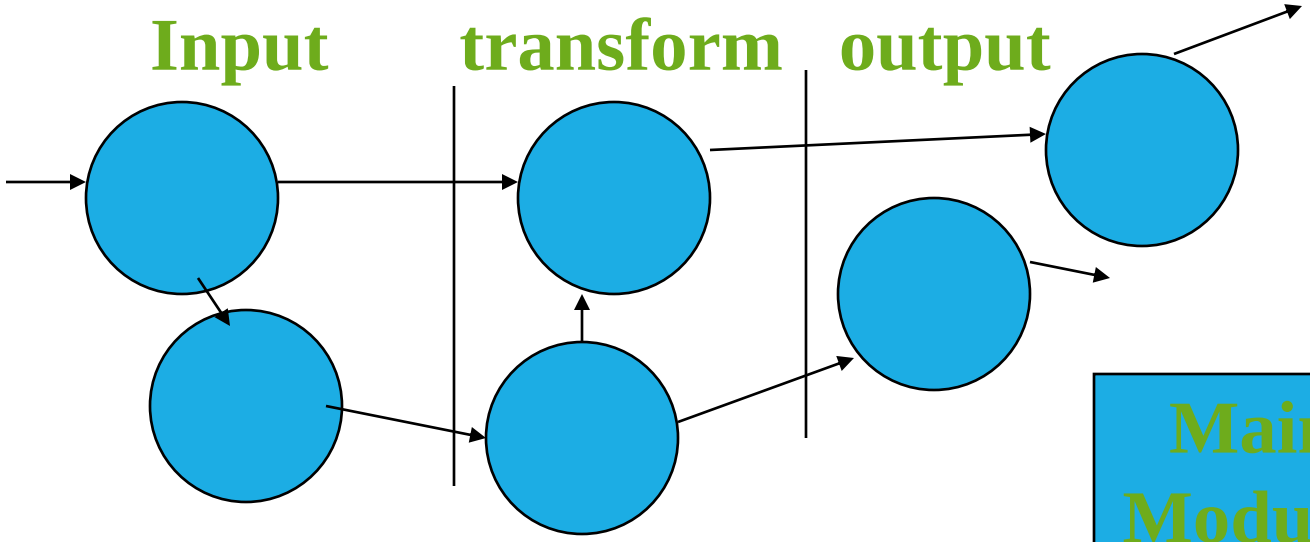




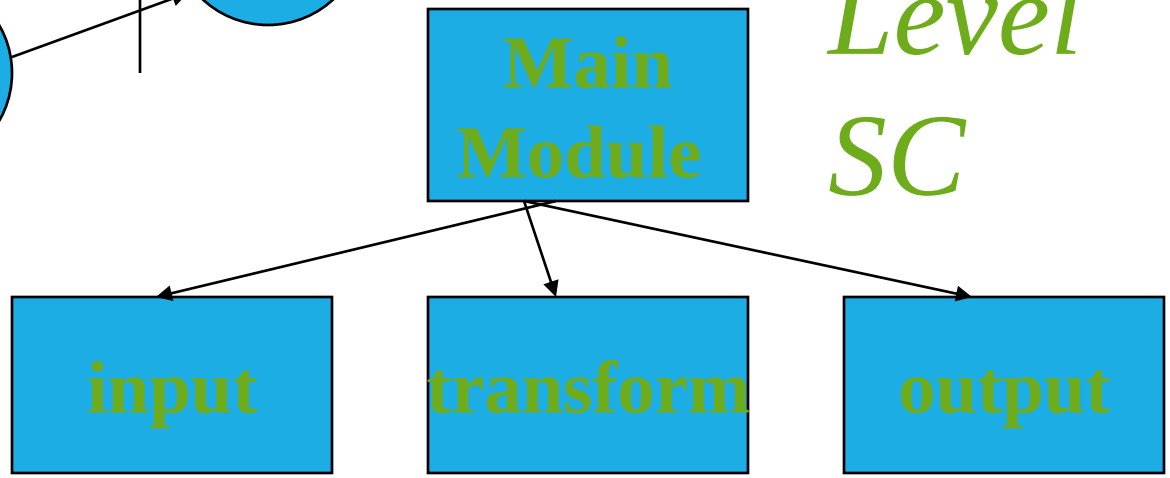

72

# Software Design Methodologies

Transform-Oriented



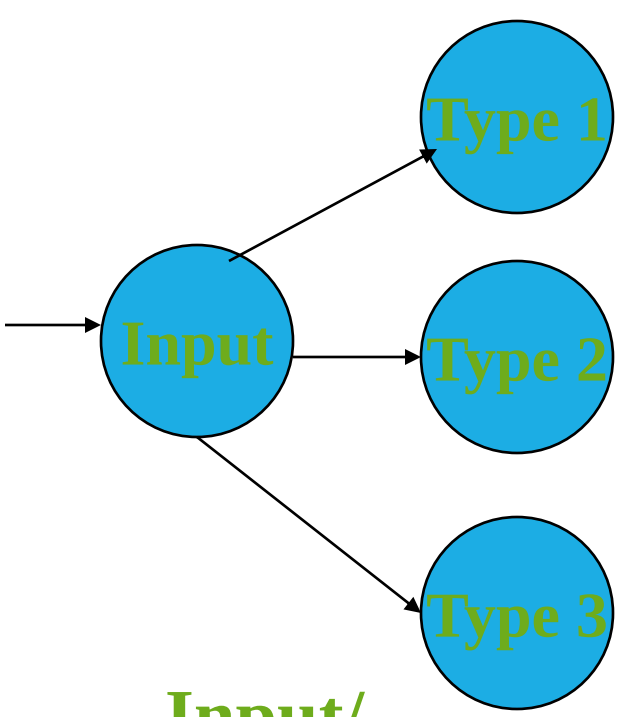

*DFD/CFD*



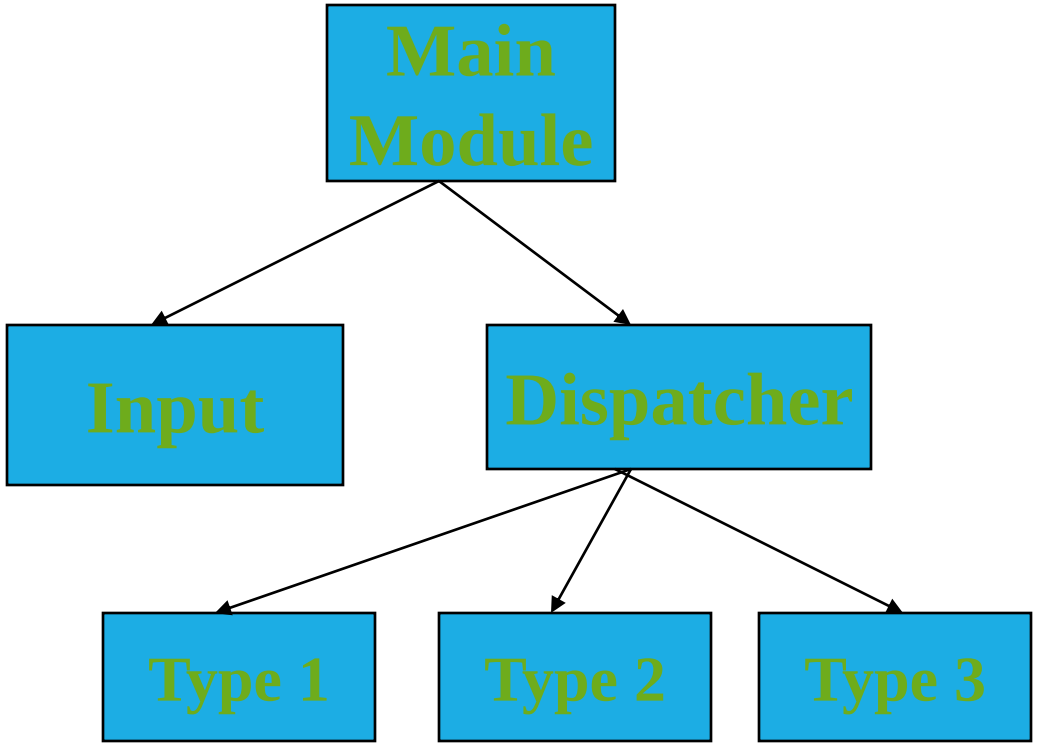

*1st Level SC*

# Software Design Methodologies

- Transaction driven




**Input/**
**process transactions**



# General Mapping Approach

- isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center
- working from the boundary outward, map DFD transforms into corresponding modules
- add control modules as required
- refine the resultant program structure using effective modularity concepts

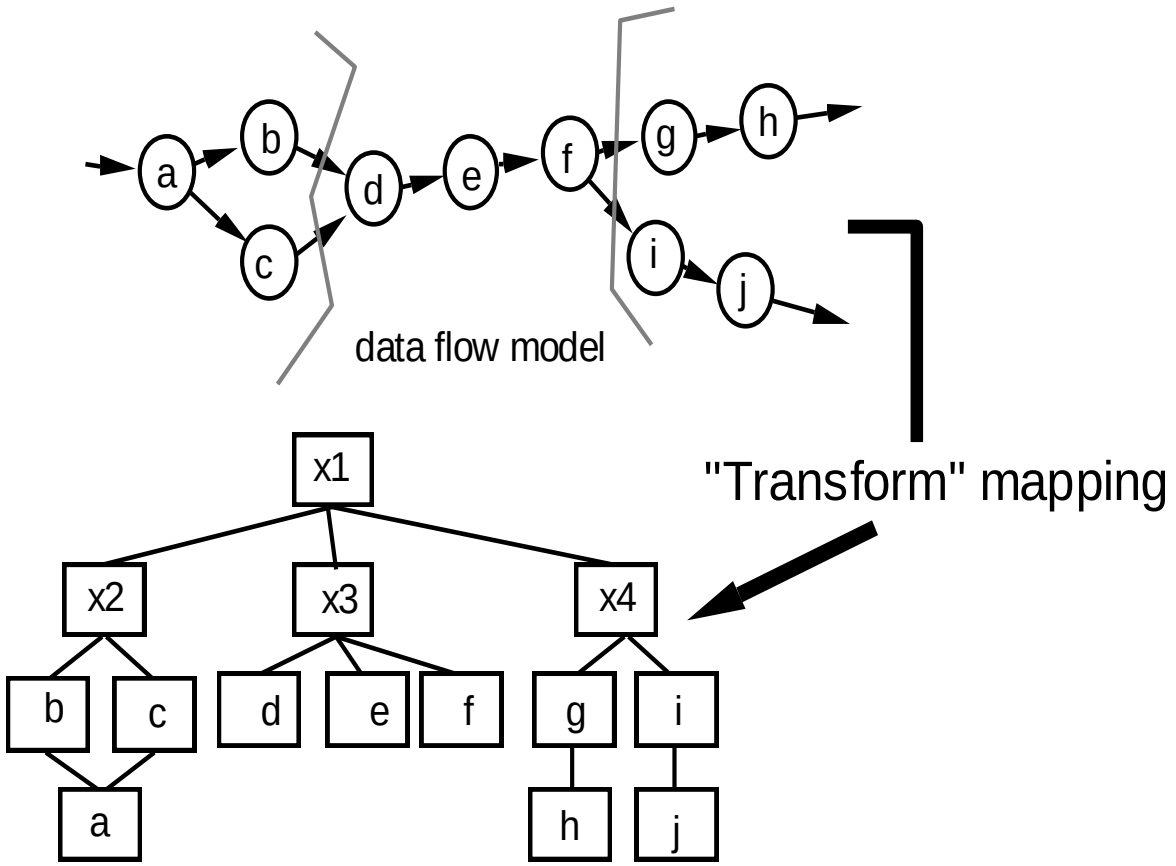

75

# General Mapping Approach

- **Isolate the transform center by specifying incoming and outgoing flow boundaries**
- **Perform "first-level factoring."**
  - The program architecture derived using this mapping results in a top-down distribution of control.
  - *Factoring* leads to a program structure in which top-level components perform decision-making and low-level components perform most input, computation, and output work.
  - Middle-level components perform some control and do moderate amounts of work.
- **Perform "second-level factoring."**


76

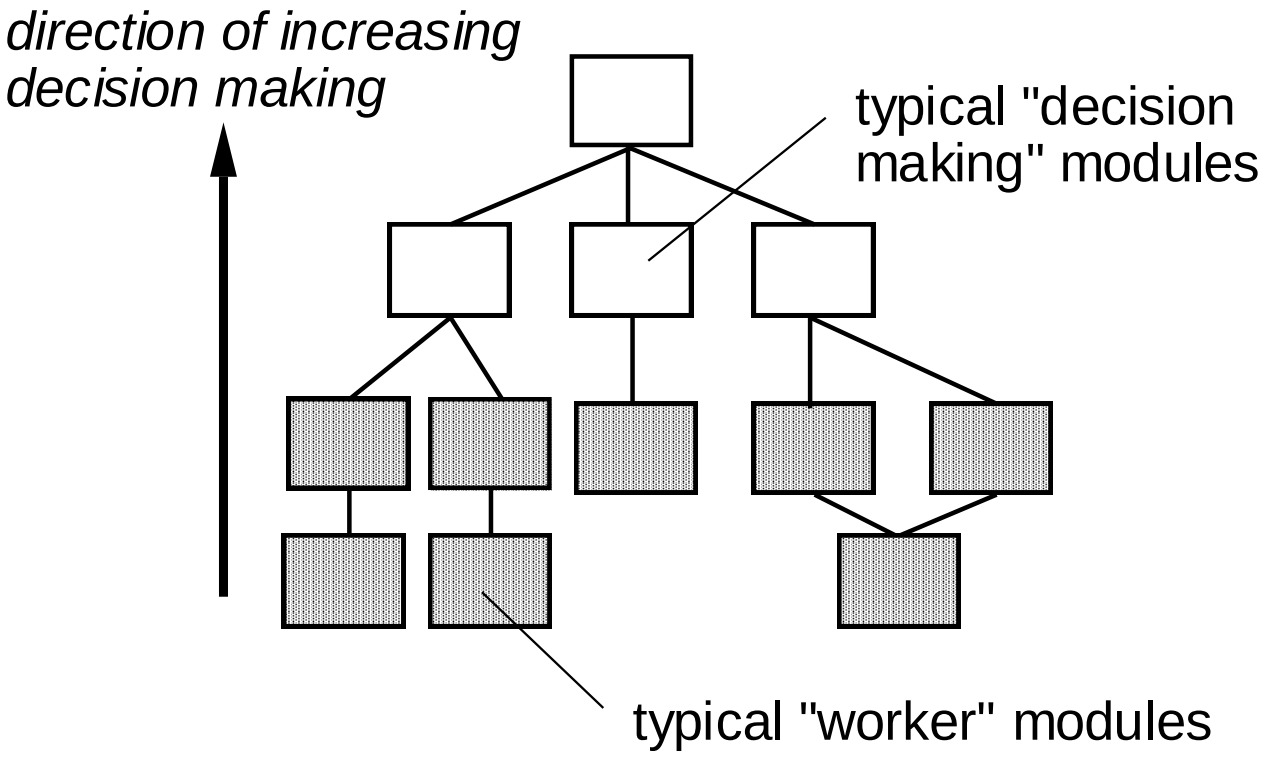
# Transform Mapping






77

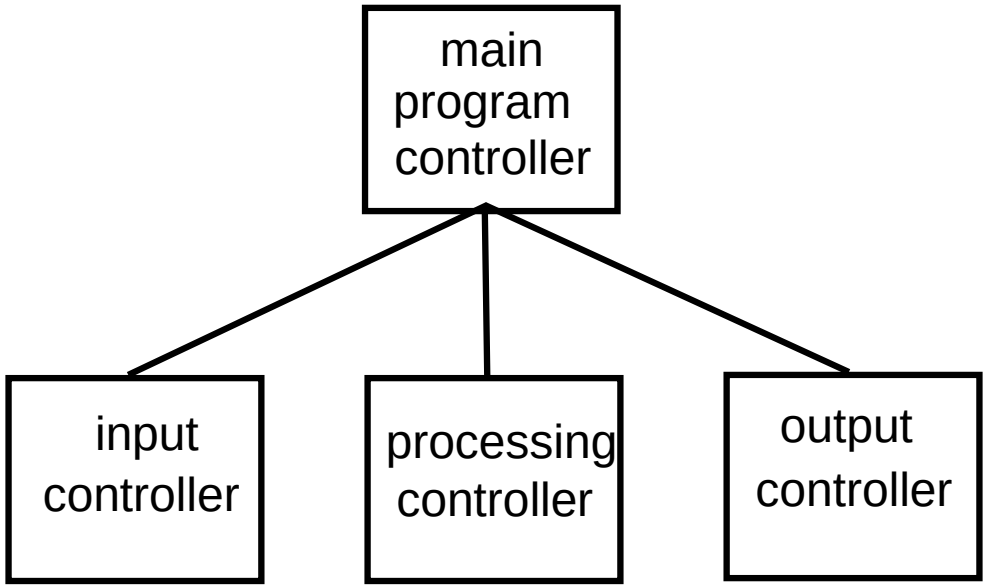
# Factoring





78

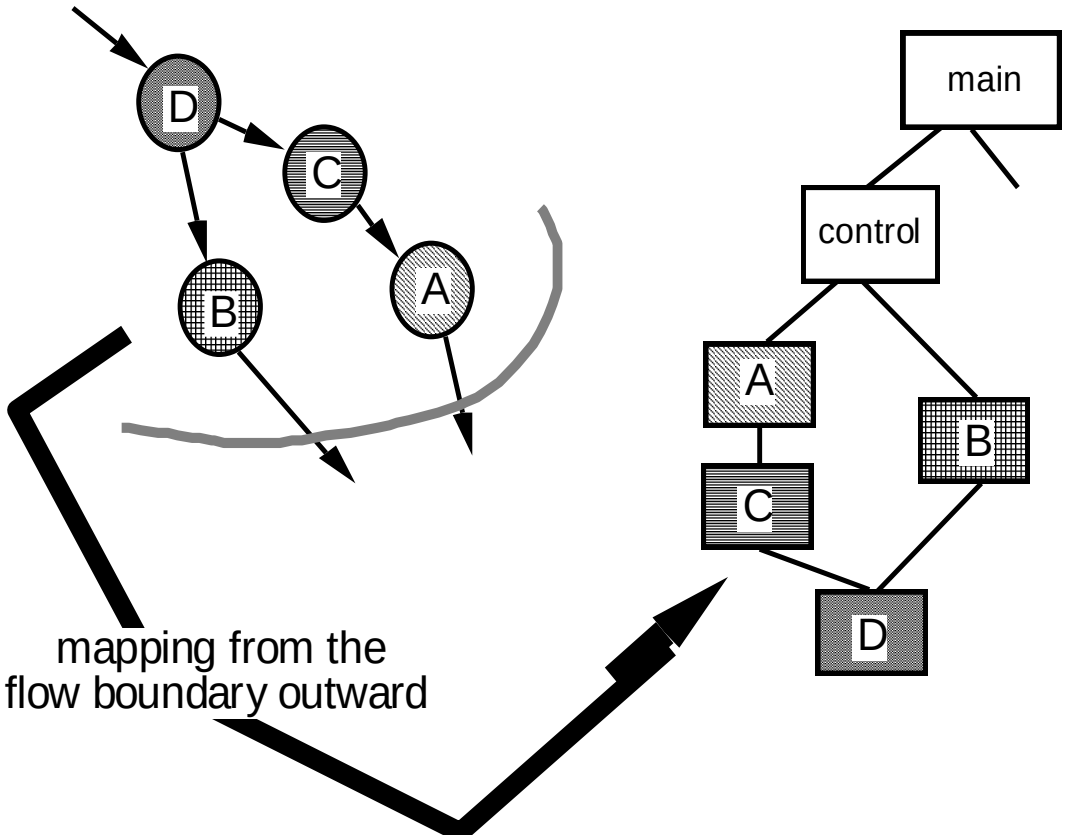
# First Level Factoring






79

# Second Level Mapping

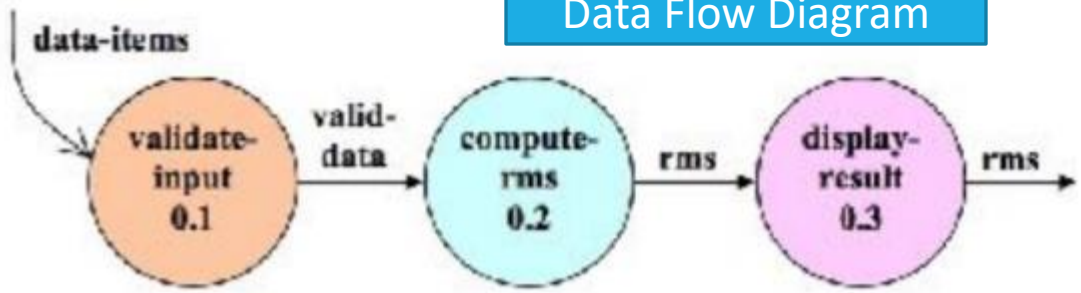


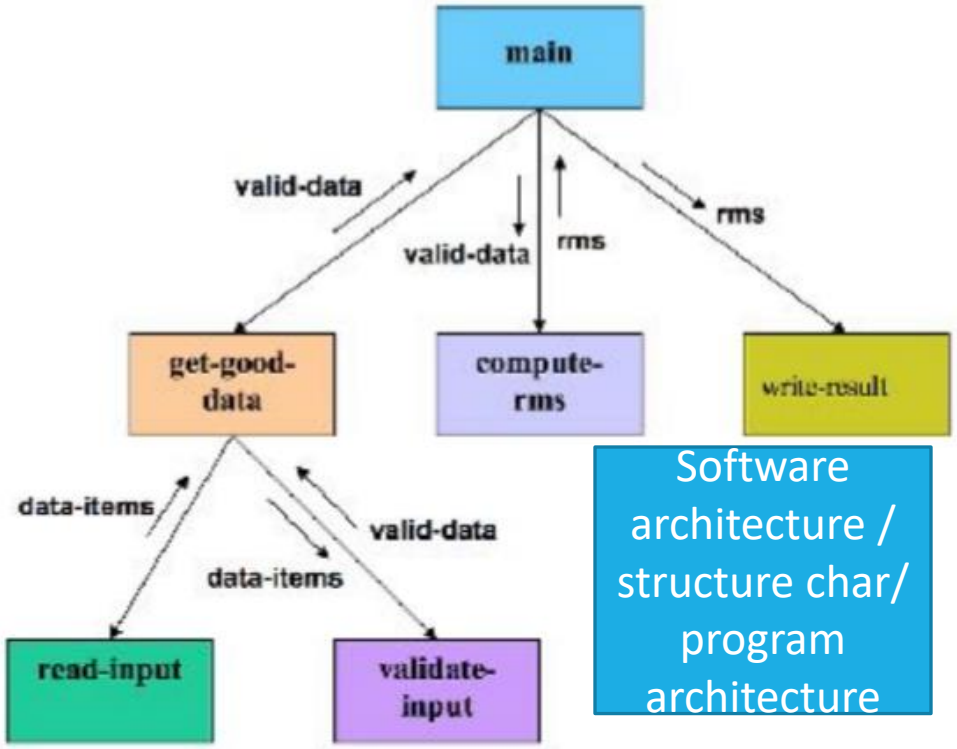



80

# Example DFD – structure chart

Data Flow Diagram




Software architecture / structure char/ program architecture

# Architectural considerations

- Provide software engineers with guidance as architecture decisions are made.
  - Economy
  - Visibility
  - Spacing
  - Symmetry
  - emergence

# Architectural decisions

- capture key design issues and the rationale behind chosen architectural solutions
- influences the system's nonfunctional characteristics and many of its quality attributes
- Architectural decisions are
  - software system organization
  - selection of structural elements and their interfaces as defined by their intended collaborations
  - the composition of these elements into increasingly larger subsystem
  - choices of architectural patterns
  - application technologies
  - middleware assets
  - programming language

# Architectural decisions

- service-oriented architecture decision (SOAD) modeling: a knowledge management framework that provides support for capturing architectural decision dependencies in a manner that allows them to guide future development activities.
- A guidance model contains knowledge about architectural decisions required when applying an architectural style in a particular application genre
  - based architectural information obtained from completed projects that employed the architectural style in that genre
- A decision model documents both the architectural decisions required and records the decisions actually made on previous projects with their justifications

# Architectural decisions

- The guidance model feeds the architectural decision model in a tailoring step that allows the architect to delete irrelevant issues, enhance important issues, or add new issues.
- A decision model can make use of more than one guidance model and provides feedback to the guidance model after the project is completed.

# Architectural Design for Web Apps

- WebApps - client-server applications
  - Structured  - multilayered architectures includes
    - a user interface or view layer
    - a controller layer which directs the flow of information to and from the client browser based on a set of business rules
    - a content or model layer that may also contain the business rules for the WebApp
- Influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client

# Architectural Design for Mobile Apps

- Mobile apps - structured using multilayered architectures that includes
  - a user interface layer
  - a business layer,
  - a data layer.
- choice of building a thin Web-based client or a rich client.
  - With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server.
  - With a rich client all three layers may reside on the mobile device itself

# Architectural Design for Mobile Apps

- number of considerations that can influence the architectural design of a mobile app:

(1) the type of web client (thin or rich) to be built,

(2) the categories of devices (e.g., smartphones, tablets) that are supported,

(3) the degree of connectivity (occasional or persistent) required,

(4) the bandwidth required,

(5) the constraints imposed by the mobile platform,

(6) the degree to which reuse and maintainability are important, and

(7) device resource constraints (e.g., battery life, memory size, processor speed).

# Interface Design

**Easy to learn?**

**Easy to use?**

**Easy to understand?**

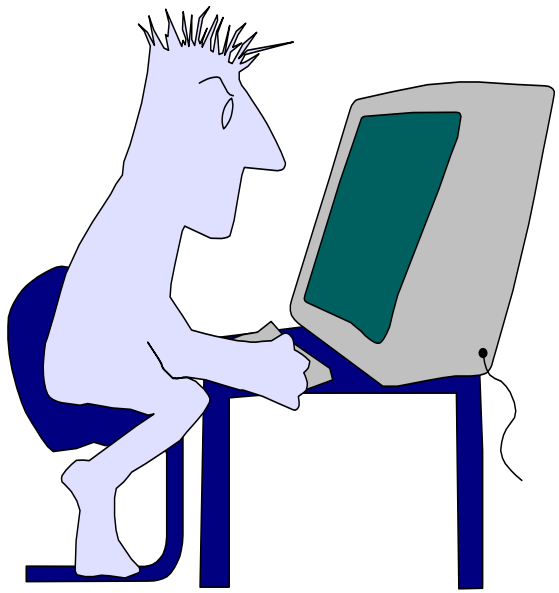


89

# Interface Design

***Typical Design Errors***

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





90

# Golden Rules

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


91

# 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.


92

# 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.


94

# User Interface  analysis and Design

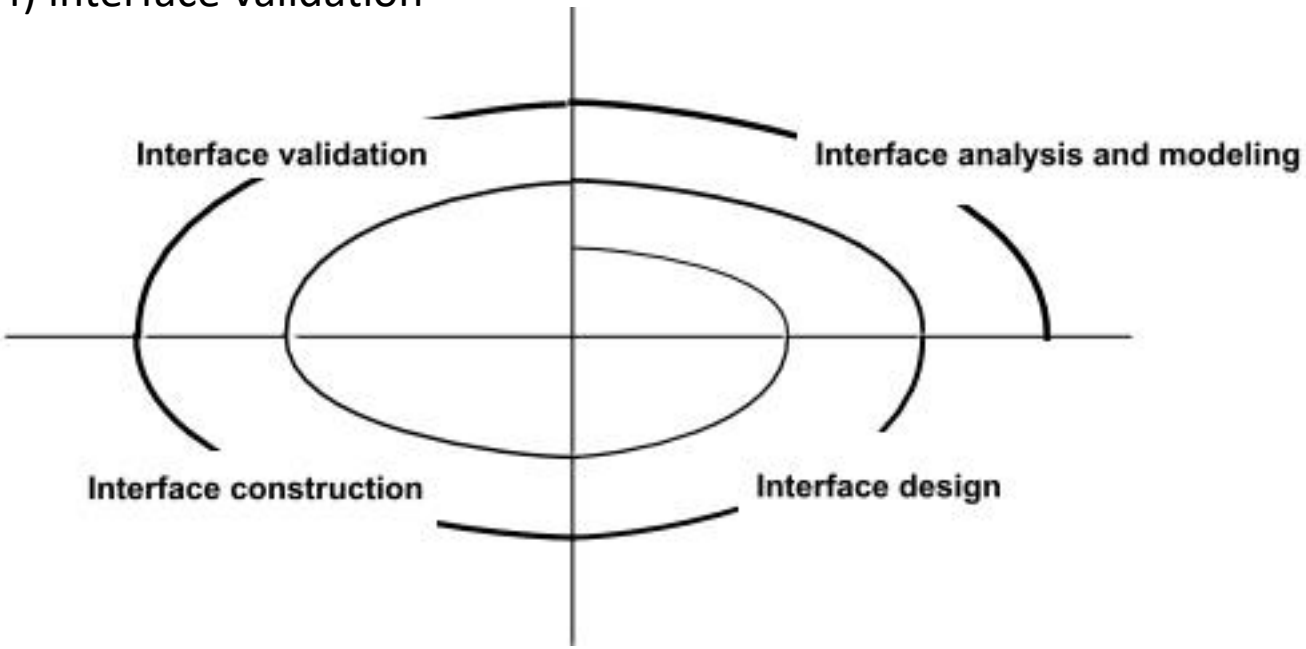
Four different models come into play when a user interface is to be analyzed and designed

- 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


95

# User Interface Design Process

- The analysis and design process for user interfaces is iterative and can be represented using a spiral model
- It begins at the interior of the spiral and encompasses four distinct framework activities: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation



# Interface analysis

- focuses on the profile of the users who will interact with the system.
- Skill level, business understanding, and general receptiveness to the new system are recorded;
- different user categories are defined.
- For each user category, requirements are elicited.
- Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).
- Finally, analysis of the user environment focuses on the characteristics of the physical work environment (e.g., location, lighting, position constraints).
- The information gathered as part of the analysis action is used to create an analysis model for the interface.
- Using this model as a basis, the design activity commences.

# interface design

- 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.

# Interface construction

- begins with the creation of a prototype that enables usage scenarios to be evaluated.
- As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

# Webapp and mobile interface design

- The user interface of a Web or mobile app is its “fi rst impression.
- A well-designed interface improves the user’s perception of the content or services provided by the site
- *Where am I?* The interface should
  - provide an indication of the WebApp that has been accessed
  - inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
  - Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

# Effective WebApp Interfaces

- Bruce Tognozzi  suggests…
  - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.


101

# Interface Design Principles-I

- Anticipation—A WebApp should be designed so that it anticipates the use's next move.
- Communication—The interface should communicate the status of any activity initiated by the user
- Consistency—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.


102

# Interface Design Principles-II

- Flexibility - The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the application in a somewhat random fashion.
- Focus—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for WebApps.
- Latency reduction—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.


103

# Interface Design Principles-III

- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable by young and old.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them."


104

# Interface Design Workflow

- Information contained within the requirements model forms the basis for the creation of a screen layout that depicts
  - graphical design and placement of icons
  - definition of descriptive screen text,
  - specification and titling for windows,
  - specification of major and minor menu items.
- The following tasks represent a rudimentary workflow:
  - Review information contained in the analysis model and refine as required.
  - Develop a rough sketch of the WebApp interface layout.
  - Map user objectives into specific interface actions.
  - Define a set of user tasks that are associated with each action.
  - Storyboard screen images for each interface action.
  - Refine interface layout and storyboards using input from aesthetic design.
  - Identify user interface objects that are required to implement the interface.
  - Develop a procedural representation of the user's interaction with the interface.
  - Develop a behavioral representation of the interface.
  - Describe the interface layout for each state.


105