UNIT-3

DESIGN CONCEPTS

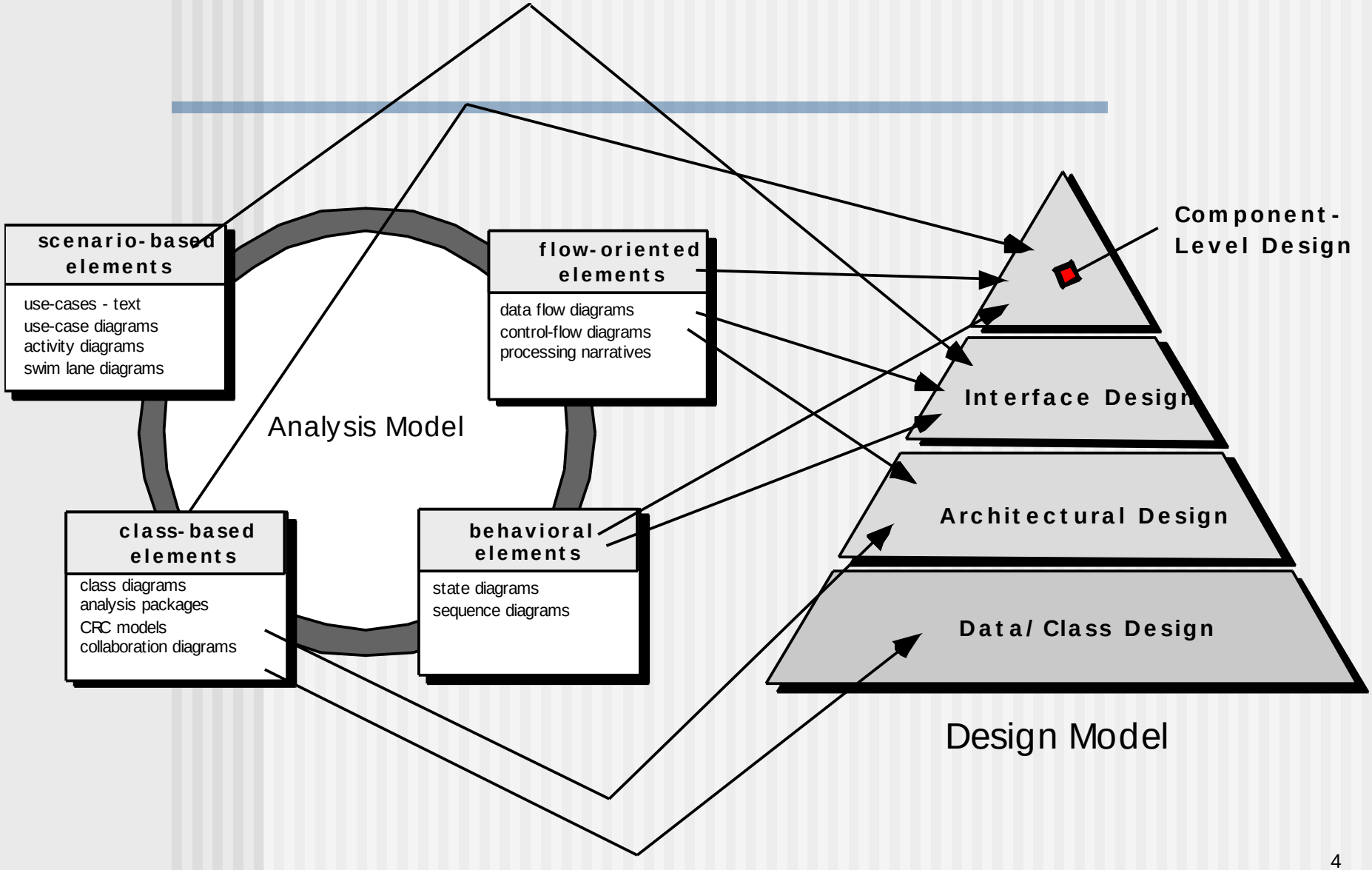
Design

- Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal.* He said:
  - Good software design should exhibit:
  - *Firmness:* A program should not have any bugs that inhibit its function.
  - *Commodity:* A program should be suitable for the purposes for which it was intended.
  - *Delight:* The experience of using the program should be pleasurable one.

Design

- Software design **sits** at the **technical kernel** of software engineering and is applied regardless of the software process model that is used.
- Each of the elements of the requirements model provides information that is necessary to create the **four design models required for a complete specification of design**.
- The flow of information during software design is illustrated in Figure (below slide).
- The requirements model, manifested by **scenario-based, class-based, flow-oriented, and behavioural elements**, feed the design task. Using design notation and design methods, design produces a **data/class design, an architectural design, an interface design, and a component design.**

Analysis Model -> Design Model



Design

- The **data/class design transforms class models into design class** realizations and the requisite **data structures required to implement** the software.
- The **objects and relationships defined in the CRC diagram** and the **detailed data content** depicted by **class attributes** and other notation **provide the basis for the data design action.**
- Part of class design may occur in conjunction with the design of software architecture.
- More detailed class design occurs as each software component is designed.

Design

- The **architectural design** defines the **relationship between major structural elements of the software**, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.
- The architectural design representation—the framework of a computer-based system—is derived from the requirements model.
- The **interface design** describes **how the software communicates with systems that interoperate** with it, and with humans who use it.

Design

- An **interface** implies a **flow of information** (e.g., data and/or control) and a **specific type of behaviour**.
- Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

- The **component-level design transforms structural elements of the software architecture into a procedural description of software components**.
- **Information obtained from the class-based models, flow models, and behavioural models** serve as the ***basis for component design.***

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. For smaller systems, design can sometimes be developed linearly.
- 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 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

- **Quality Attributes.** Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:
- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, the ease with which a system can be installed, and the ease with which problems can be localized.

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” 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.

From Davis [DAV95]

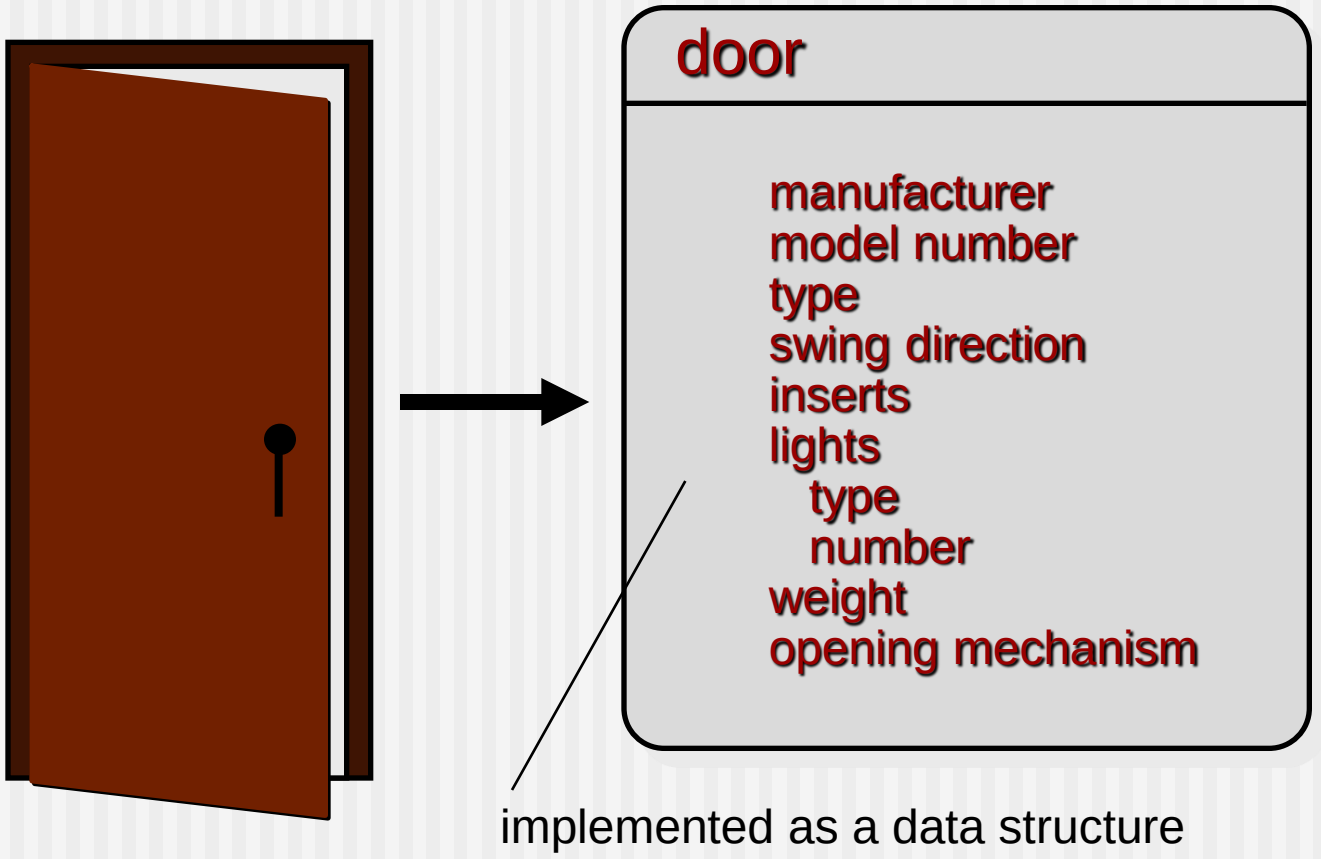
Fundamental Concepts

- 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—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

Design concepts provides the software designer with a foundation from which more sophisticated design methods can be applied.

- Each helps you answer the following questions:
- What criteria can be used to **partition software into individual components?**
- How is **function or data structure detail separated from a conceptual representation** of the software?
- What **uniform criteria define the technical quality** of a software design?

Data Abstraction



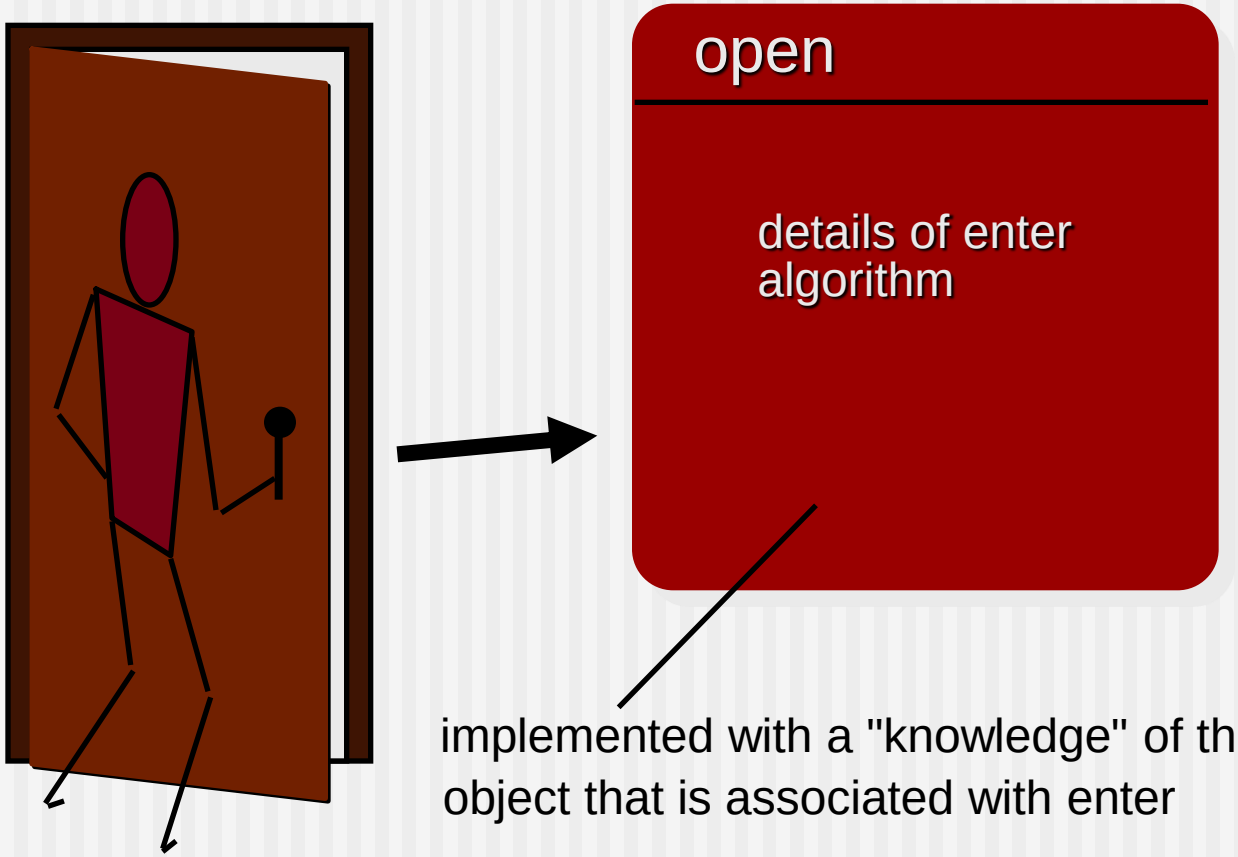
Data Abstraction

- **At the highest level** of abstraction, a solution is **stated in broad terms** using the language of the problem environment.
- **At lower levels** of abstraction, a more **detailed description** of the solution is provided.
- There are two types of abstraction : **Procedural abstraction, data abstraction.**
- A ***procedural abstraction*** *refers to a* ***sequence of instructions*** that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.
- An **example of a procedural abstraction** would be the word ***open for a door****.*
- *Open implies a long sequence* of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)

Data Abstraction (contd..)

- A ***data abstraction*** *is a named collection of data that describes a data object.* ***In*** **the context of the procedural abstraction** ***open, we can define a data abstraction*** called **door.**
- **Like any data object, the data abstraction for door would encompass** a set of attributes that **describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).**
- It follows that the procedural abstraction ***open*** would make use of information contained in the attributes of the data abstraction **door.**

Procedural Abstraction



Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.”

- Architecture is the **structure or organization of program components (modules),** the manner in which these components interact, and the structure of data that are used by the components.
- One **goal** of software design is to **derive an architectural rendering of a system.**
- This **rendering serves** as a framework from which more **detailed design activities** are conducted.
- A set of architectural patterns enables a software engineer **to solve common design problems**.

Architecture(contd.,)

- Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:
- **Structural properties.** This aspect of the architectural design representation **defines the components of a system** (e.g., modules, objects, filters) and the **manner in which those components are packaged and interact with one another**. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves **requirements for performance, capacity, reliability, security, adaptability, and other system characteristics**.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the **ability to reuse architectural building blocks.**

Architecture(contd.,)

- Given the specification of these properties, the architectural design can be represented using one or more of a number of different models .
- ***Structural models*** represent **architecture** as an organized collection of program components.
- ***Framework models*** *increase the* ***level of design abstraction*** *by attempting to identify* repeatable architectural design frameworks that are encountered in similar types of applications.
- ***Dynamic models*** *address the* ***behavioural 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

Design Pattern Template

***Pattern name*—describes the essence of the pattern in a short but expressive name**

***Intent*—describes the pattern and what it does**

***Also-known-as*—lists any synonyms for the pattern**

***Motivation*—provides an example of the problem**

***Applicability*—notes specific design situations in which the pattern is applicable**

***Structure*—describes the classes that are required to implement the pattern**

***Participants*—describes the responsibilities of the classes that are required to implement the pattern**

***Collaborations*—describes how the participants collaborate to carry out their responsibilities**

***Consequences*—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented**

***Related patterns*—cross-references related design patterns**

Patterns(contd.)

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

- Brad Appleton defines a *design pattern in the following manner: “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”

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.**

Separation of Concerns (contd.)

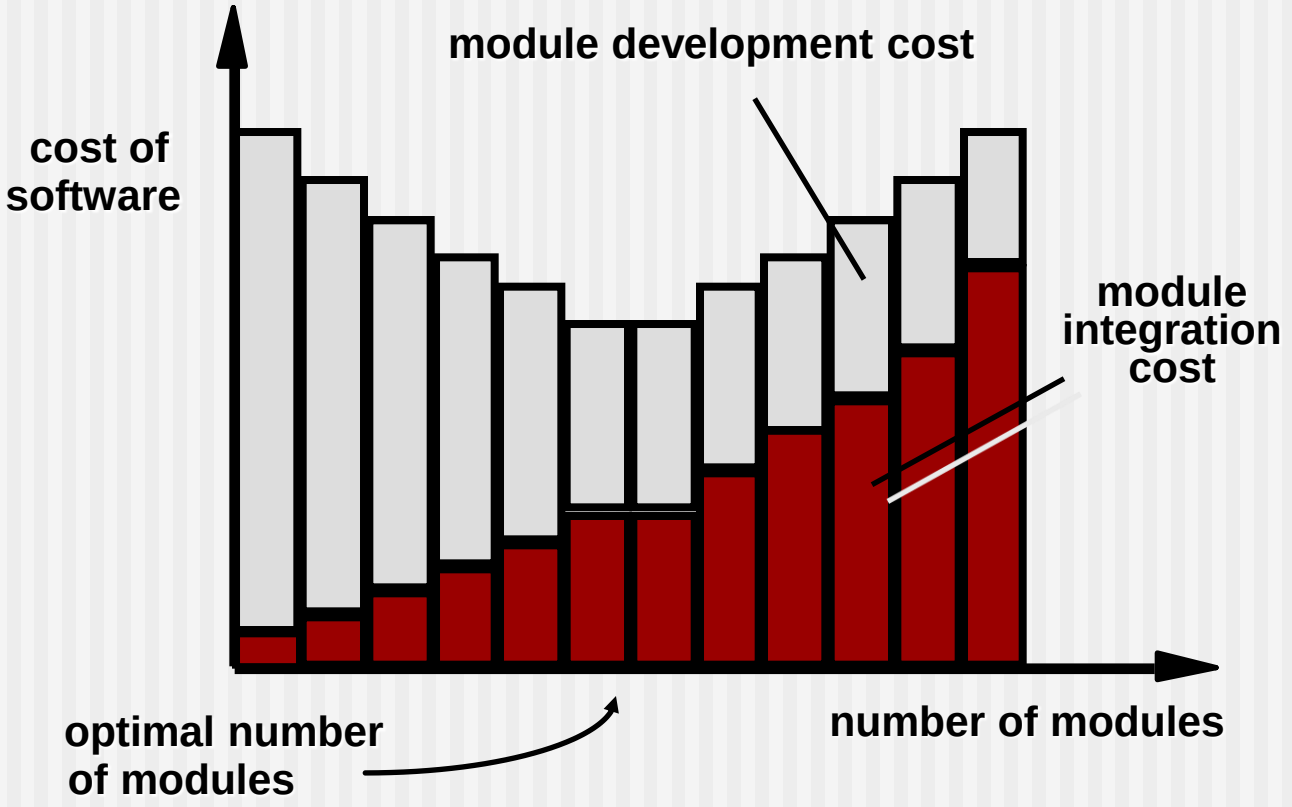
- For two problems, *p1 and p2, if the perceived complexity of p1 is greater than the* perceived complexity of *p2, it follows that the* ***effort required to solve p1 is greater*** **than the effort required to solve** ***p2.***
- *As a general case, this result is intuitively obvious.* It does take **more time to solve a difficult problem.**
- It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy.

Modularity

- Modularity is the most **common manifestation of separation of concerns.**
- Software is divided into separately named and addressable components, sometimes called *modules, that are integrated to satisfy problem requirements.*
- **“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?

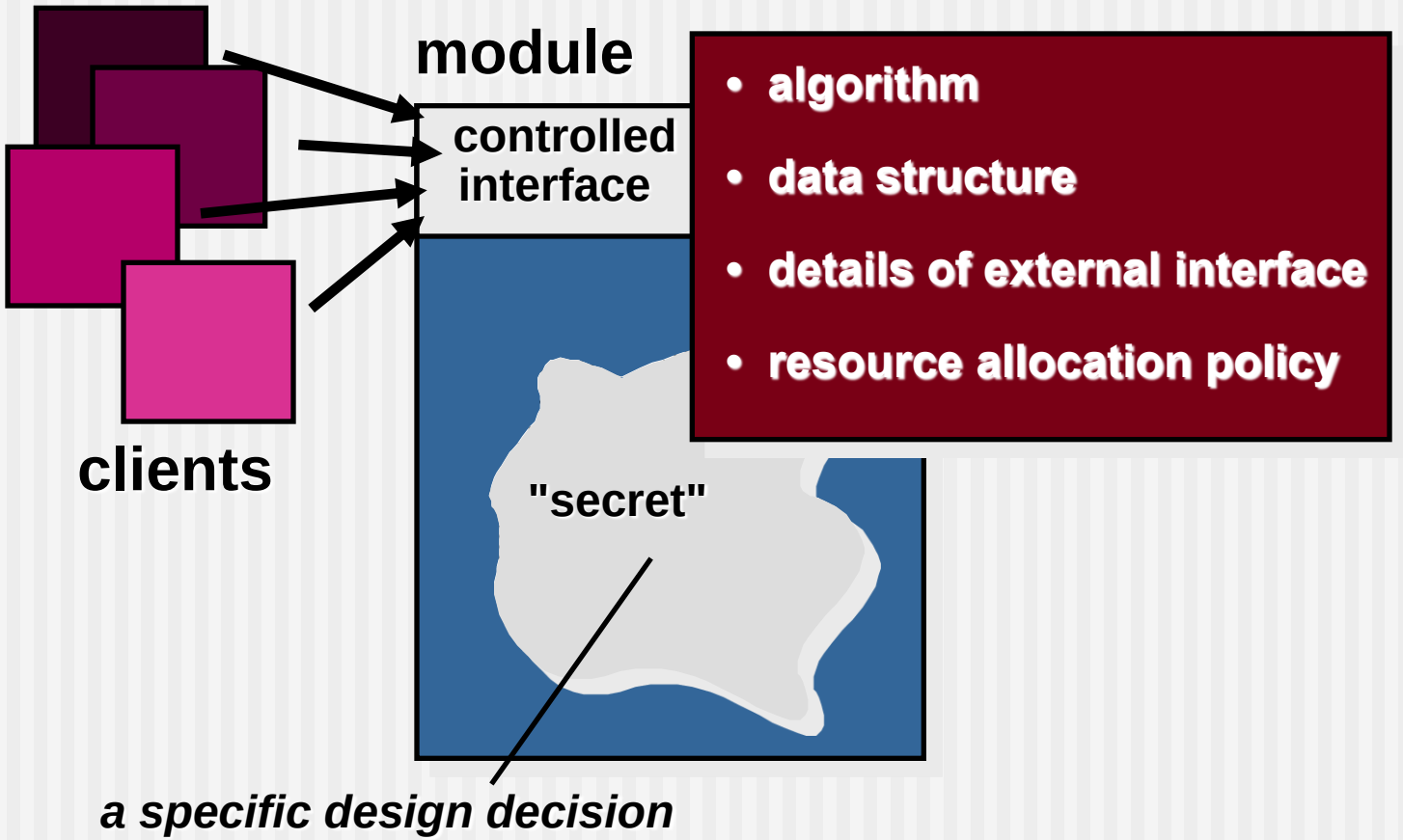


Modularity(contd.)

- **The effort (cost) to develop an individual software module does decrease as the total number of modules increases.**

- Given the same set of requirements, more modules means smaller individual size.
- As the **number of modules grows**, the **effort (cost)** associated with **integrating the modules also grows.** These characteristics lead to a total cost or effort curve shown in the figure.
- There is a number, *M, of modules that would result in minimum development* cost, but we do not have the necessary sophistication to predict *M with assurance.*
- Project should be modularized, but care should be taken to stay in the vicinity of *M.*
- ***Undermodularity or overmodularity*** *should be avoided. But how to* know the vicinity of *M? How modular should you make software?*

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

Information Hiding(contd.)

- The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.”
- In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

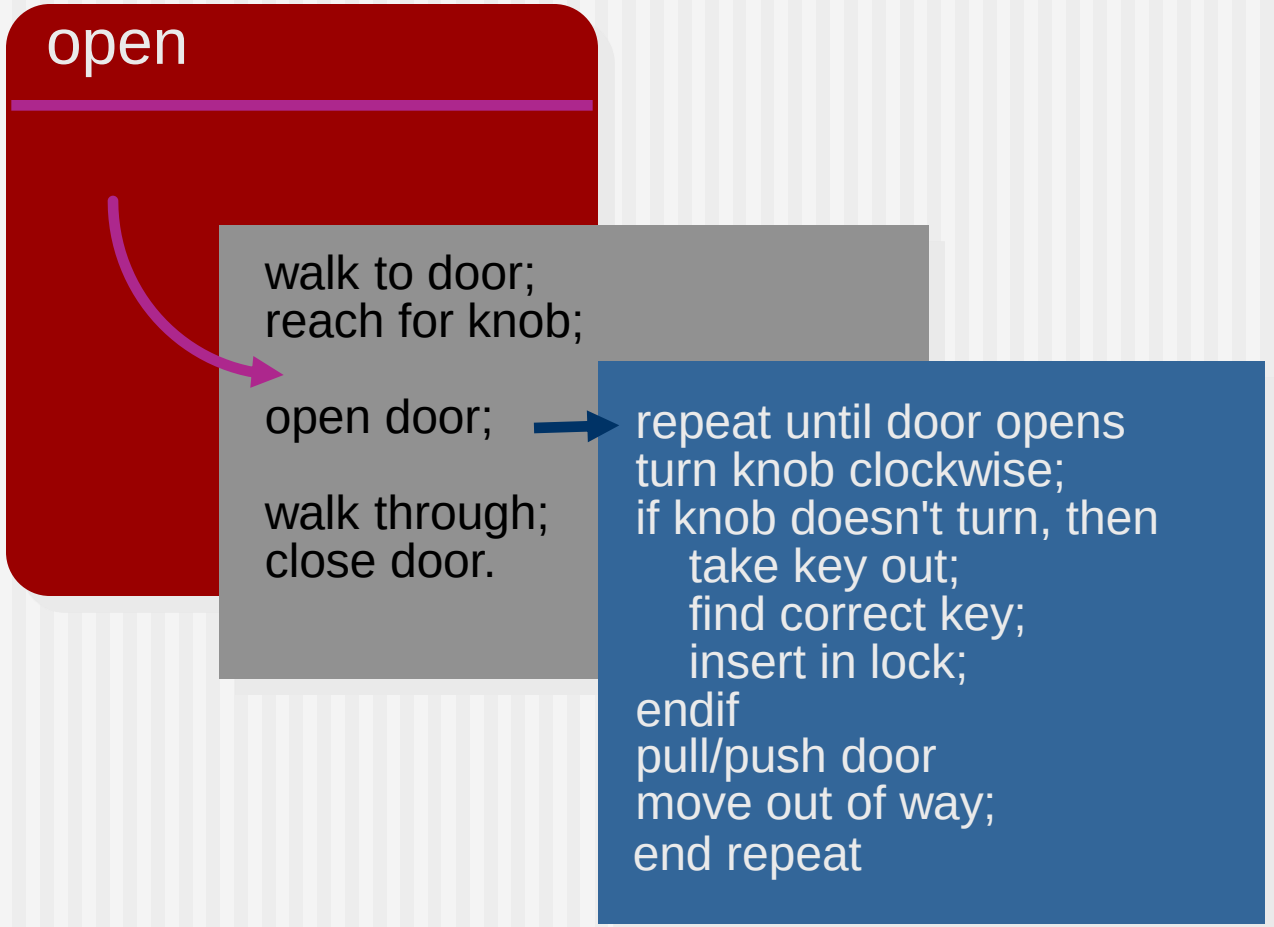
Information Hiding(contd.)

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

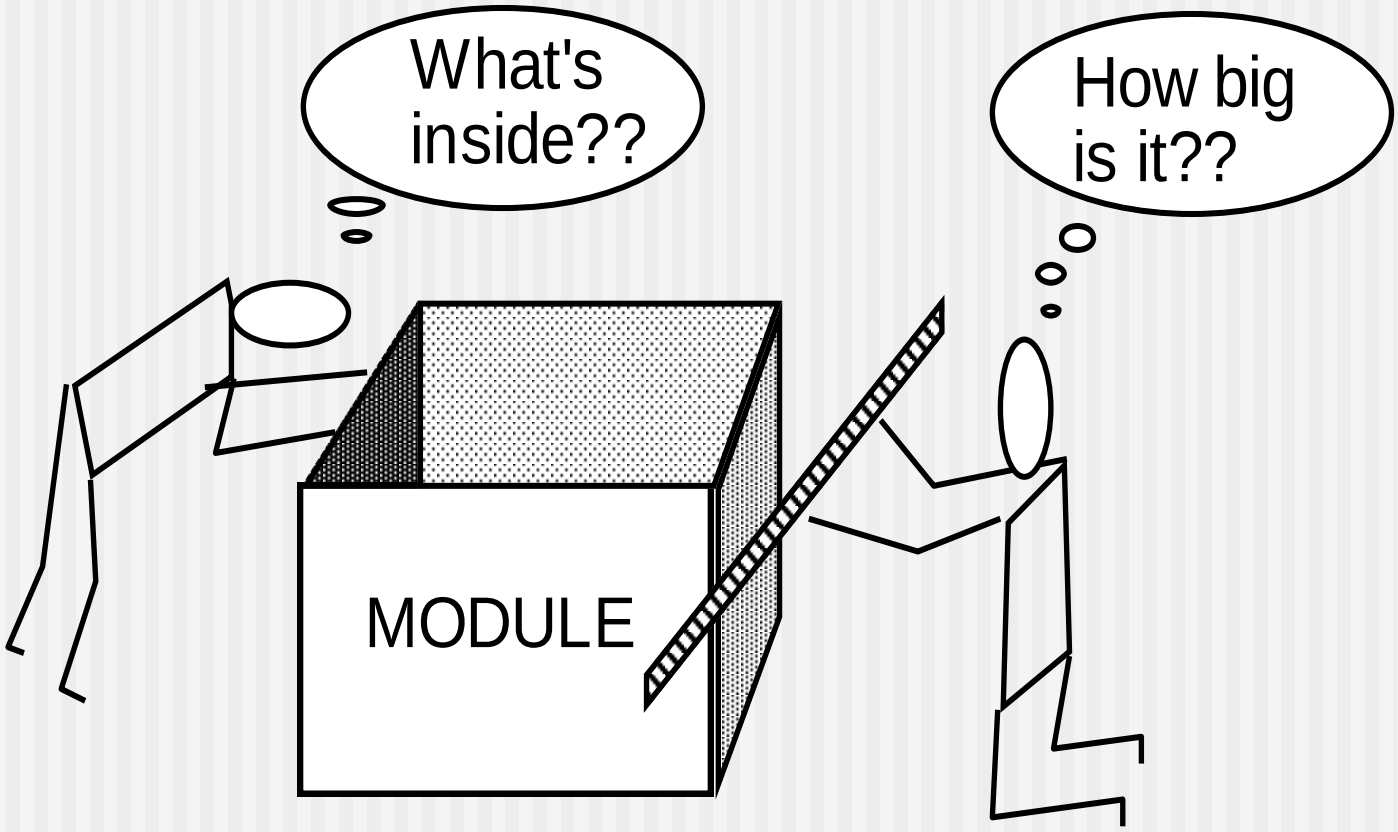
Information Hiding advantages

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

Stepwise Refinement



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

Functional Independence(contd.)

- Software with effective modularity, that is, independent modules, is easier to develop because **function can be compartmentalized and interfaces are simplified**

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

- Always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.

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

Refinement

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

Refinement (contd.)

- Refinement is actually a **process of *elaboration***. *Begin with a statement of* function (or description of information) that is defined at a high level of abstraction.

- The statement **describes function or information** conceptually but **provides no information about the internal workings of the function** or the internal structure of the information.

- Then elaborate on the original statement, **providing more and more detail as each successive refinement (elaboration) occurs.**

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.
- An *aspect* is a representation of a cross-cutting concern.

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

- Fowler defines refactoring in the following manner:

  - "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.

Refactoring(contd.)

- For example, a **first design** iteration might yield a component that exhibits **low cohesion** (i.e., it performs three functions that have only limited relationship to one another).
- After careful consideration, decide the component should be **refactored into three separate components, each exhibiting high cohesion**.

OO Design Concepts

- Design classes
  - Entity classes (bank account)
  - Boundary classes (Windows, screens and menus are examples of boundaries that interface with users.)
  - Controller classes (business logic or other)
- Inheritance—all responsibilities of a super class 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.

Five different types of design classes, each representing a different layer of the design architecture, can be developed

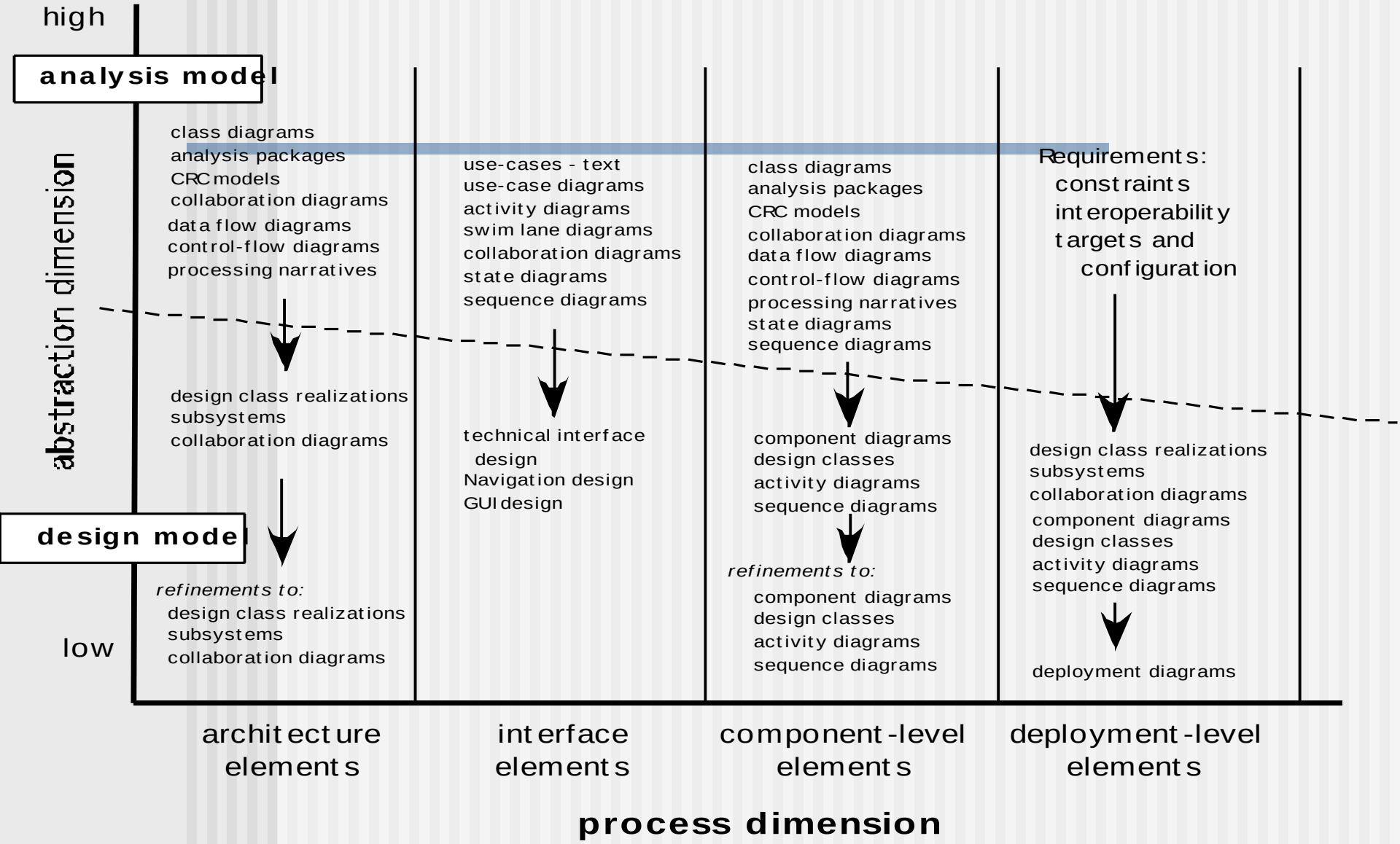
- ***User interface classes*** ***define all abstractions that are necessary for human computer*** **interaction (HCI).** In many cases, HCI occurs within the context of a ***metaphor (e.g., a checkbook, an order form, a fax machine), and the design*** **classes** for the interface may be visual representations of the elements of the metaphor.
- ***Business domain classes*** *are often* ***refinements of the analysis classes*** *defined* earlier. **The** ***classes identify the attributes and services (methods)*** that are required to **implement** some element of the **business domain**.
- ***Process classes*** *implement* ***lower-level business abstractions*** *required to fully* manage the business domain classes.
- ***Persistent classes*** *represent data stores (e.g., a database) that will* ***persist*** **beyond the execution of the software.**
- ***System classes*** *implement software management and control functions that* enable the system to **operate and communicate within its computing environment and with the outside world**.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

- ***Complete and sufficient*. A design class should be the complete encapsulation** of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **Scene defined for video-editing software is complete** only **if it contains all attributes and methods** that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that **are sufficient to achieve the intent of the class, no more and no less.**
- ***Primitiveness*. Methods associated with a design class should be focused** on accomplishing ***one service for the class***. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip for video-editing** software might have attributes start-point and end-point to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, ***setStartPoint() and setEndPoint(), provide the only means for establishing start and end points* for the clip.**

- ***High cohesion*. A cohesive design class has a small, *focused set of responsibilities*** and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip might contain a set of** methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.
- ***Low coupling*. Within the design model, it is necessary for design classes to** collaborate with one another. However, collaboration should be kept to an acceptable minimum. *If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time.* In general, design classes within a subsystem should **have only limited knowledge of other classes.** This **restriction**, called the ***Law of Demeter**, suggests that a method should only send* messages to methods in neighbouring classes.

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”
- 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 elements

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

Data elements(contd.,)

- The elements of the design model use many of the same UML diagrams that were used in the analysis model.
- The difference is that these diagrams are **refined and elaborated** as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

Data elements(contd.,)

- **Data design** (sometimes referred to as *data architecting) creates a model of data and/or information that is represented at a **high*** **level of abstraction (the customer/user's view of data)**
- At the **program component level**, the **design of data structures and the associated algorithms** required to manipulate them is essential to the creation of high-quality applications.

Data elements(contd.,)

- At the **application level**, the **translation of a data model** (derived as part of requirements engineering) **into a database is pivotal** to achieving the business objectives of a system.
- At the **business level**, the **collection of information stored in disparate databases** and **reorganized into a "data warehouse"** enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements

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

Interface Design Elements

- There are three important elements of interface design:
  - (1) **the user interface** (UI);
  - (2) **external interfaces** to other systems, devices, networks, or other producers or consumers of information; and
  - (3) **internal interfaces** between various design components.
- These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

Interface Design Elements(contd.,)

- If the classic input-process-output approach to design is chosen, the **interface** of each software component is designed **based on data flow representations and the functionality described in a processing narrative.**
- In UML, an interface is defined as “An **interface** is a specifier for the externally-visible [public] **operations of a class**, **component, or other classifier (including subsystems) without specification of internal structure.**
- ” An interface is a set of operations that describes some **part of the behaviour of a class** and provides access to these operations.

Interface Design Elements(contd.,)

- For example, the *SafeHome security function makes use of a control panel that allows* a homeowner to control certain aspects of the security function.
- In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone.
- The **ControlPanel class provides the behavior associated with a keypad,** and therefore, it must implement the operations *readKeyStroke () and decodeKey ().*

Interface Design Elements(contd.,)

- If these operations are to be provided to other classes (in this case, **WirelessPDA** and **MobilePhone), it is useful to define an interface as shown in the figure.**
- The interface, named **KeyPad, is shown as an <<interface>> stereotype or as a small,** labelled circle connected to the class with a line.
- The **interface is defined with no attributes and the set of operations** that are necessary to achieve the behaviour of a keypad.

Interface Elements

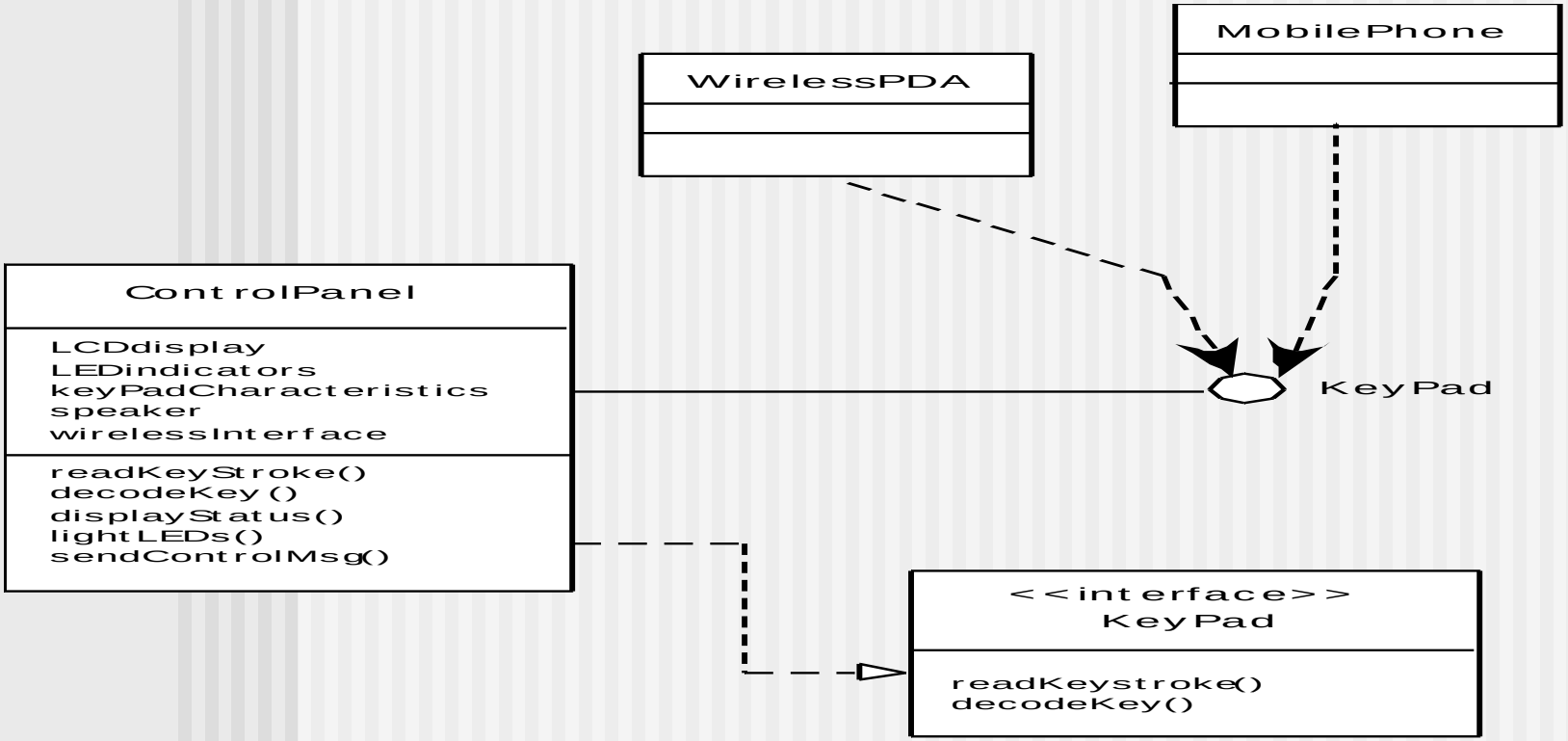
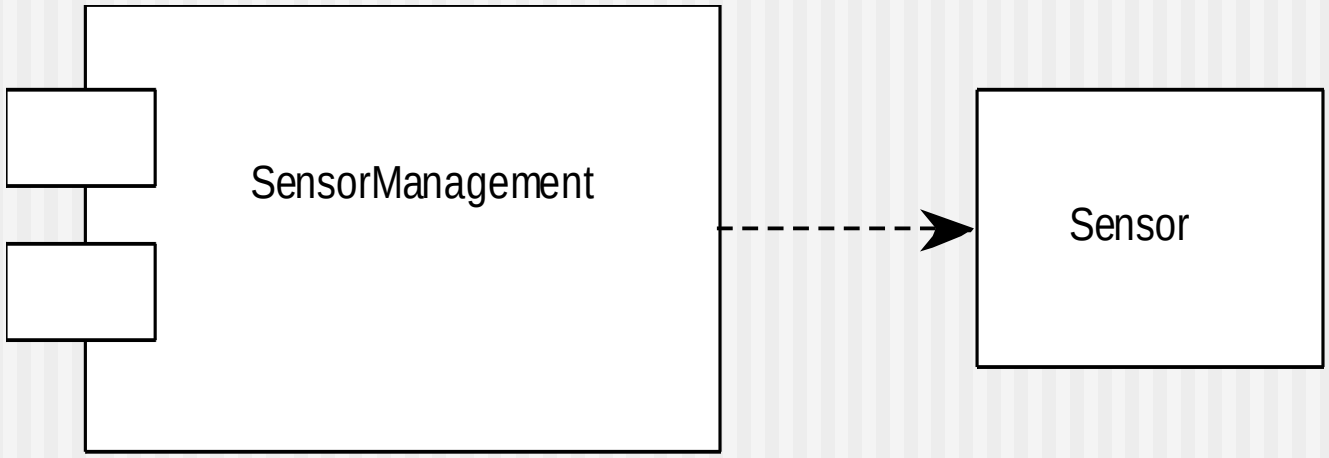


Figure 9.6 UML interf ControlPanel

Interface Elements(contd.,)

- The dashed line with an open triangle at its end indicates that the **ControlPanel class provides KeyPad operations as part of its behaviour. In UML, this** is characterized as a *realization. That is, part of the behaviour of* ***ControlPanel will*** be implemented by realizing **KeyPad operations.**
- **These operations will be provided** to other classes that access the interface.

Component Elements



Component Elements(contd.,)

- The component-level design for software fully describes the ***internal detail of each software component***.
- To accomplish this, the component-level design **defines data structures for all local data objects and algorithmic detail** for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).
- A component named **SensorManagement (part of the *SafeHome security function) is*** represented.
- A dashed arrow connects the component to a class named **Sensor that** is assigned to it. The **SensorManagement component performs all functions associated** with ***SafeHome sensors including monitoring and configuring them.***

Component Elements(contd.,)

- A **UML *activity diagram* can be used to represent processing logic.**
- Detailed procedural flow for a component can be represented using either pseudocode (a programming language-like representation or some other diagrammatic form (e.g., flowchart or box diagram).

Deployment Elements

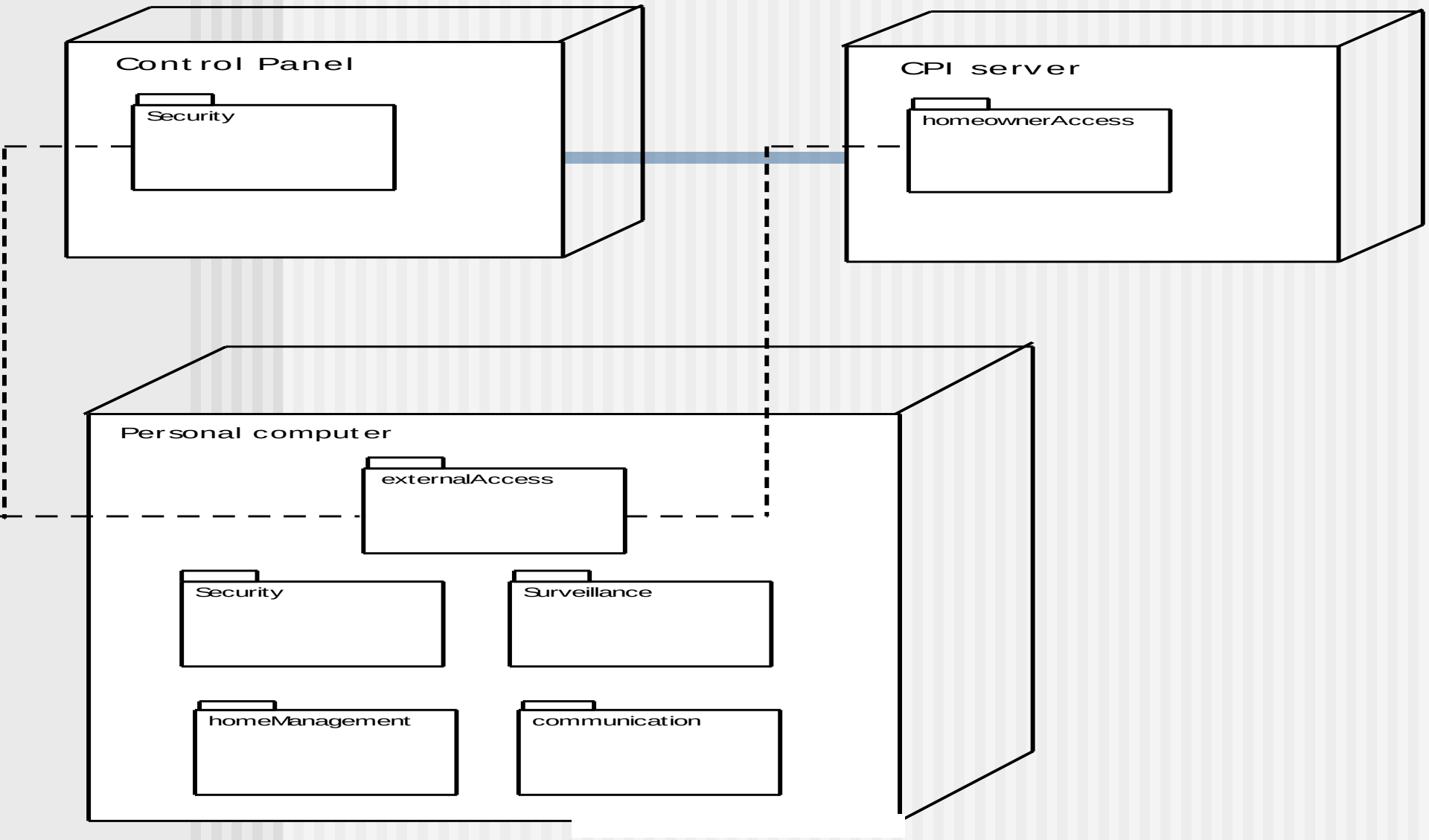


Figure 9.8 UML deployment diagram for *SafeHome*

Deployment Elements(contd.,)

- Deployment-level design elements 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 are configured* to operate within three primary computing environments—a home-based PC, the *SafeHome control panel, and a server housed at CPI Corp. (providing Internet-based* access to the system).

Deployment Elements(contd.,)

- The subsystems (functionality) housed within each computing element are indicated.
- For example, the personal computer houses subsystems that **implement security, surveillance, home management, and communications features.**
- An external access **subsystem** has been designed to manage all attempts to **access the *SafeHome system from an external source.***
- *Each subsystem would be elaborated to* indicate the components that it implements.

Deployment Elements(contd.,)

- The diagram shown in Figure above is in ***descriptor form***. *This means that the deployment* diagram shows the computing environment but does not explicitly indicate configuration details.
- For example, the "**personal computer**" is not further identified.
- It could be a **Mac or a Windows-based PC, a Sun workstation, or a Linux-box.**
- **These details are provided when the deployment diagram is revisited in *instance form*** during the latter stages of design or as construction begins.
- Each instance of the deployment (a specific, named hardware configuration) is identified.

Chapter 9

Architectural Design

Architectural Styles

- An architectural style is a **transformation that is imposed on the design of an *entire system***
- The **intent** is to **establish a structure for all components of the system.**
- An **architectural pattern**, like an architectural style, imposes a ***transformation on the design of an architecture.***
- However, a pattern differs from a style in a number of fundamental ways:

Architectural Styles(contd.,)

- (1) The scope of a pattern is less broad, focusing on **one aspect of the architecture** rather than the architecture in its entirety;
- (2) A **pattern imposes a rule on the architecture,** describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)
- (3)Architectural patterns **tend to address specific behavioural issues** within the context of the architecture
  - (e.g., how real-time applications handle synchronization or interrupts).

Architectural Styles(contd.,)

Each style describes a system category that encompasses:

(1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system,

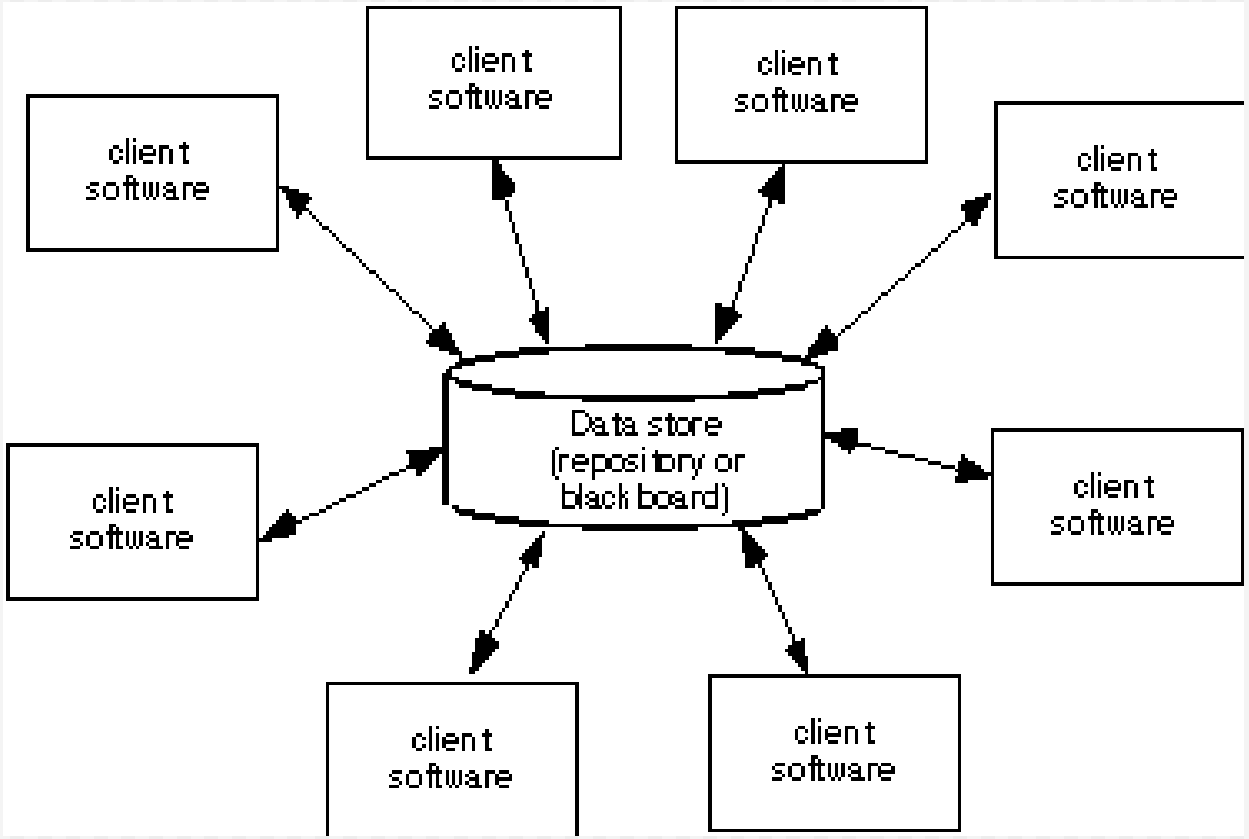
(2) a **set of connectors** that enable “communication, coordination and cooperation” among components,

(3) constraints that define how components can be integrated to form the system, and

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

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

Data-Centered Architecture



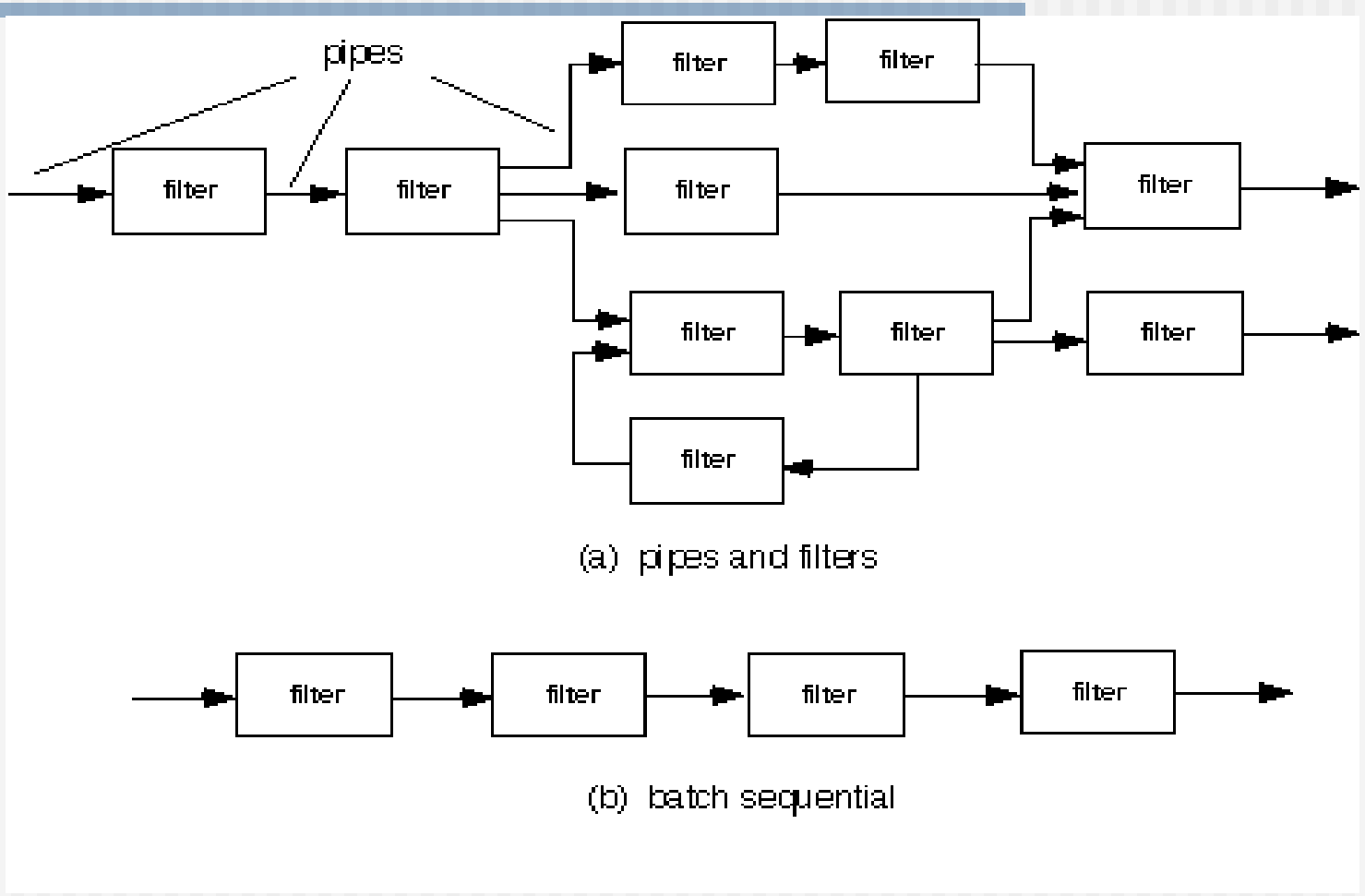
Data-Centered Architecture

- A data store (e.g., a file or database) **resides at the centre of this architecture** and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Figure above illustrates a typical data-centred style.
- Client software accesses a central repository.
- In **some cases** the data repository is **passive**. That is, **client software accesses the data independent of any changes to the data** or the actions of **other client software.**

Data-Centered Architecture(contd.,)

- Data-centred architectures promote ***integrability*** . *That is,* ***existing*** **components can be changed and new client components added to the architecture** without concern about other clients because the client components operate independently.
- **Data can be passed** among clients **using** the **blackboard mechanism** (i.e., the blackboard component serves to coordinate the transfer of information between clients).
- Eg for black board style is **Online chatting**
- **Client components independently execute processes**.
- Example of data-centred architectures is the **Web architecture**

Data Flow Architecture



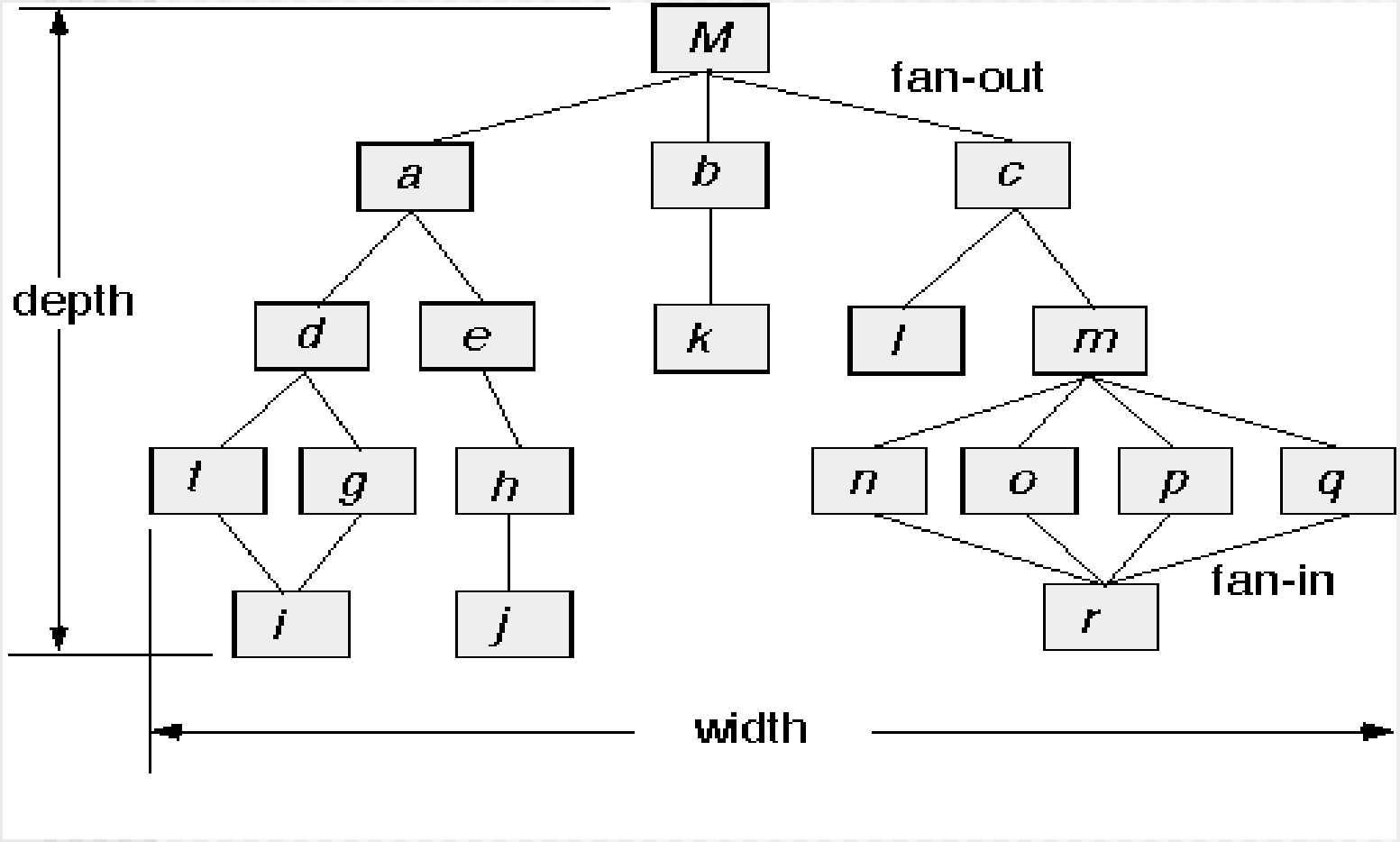
Data Flow Architecture(contd.,)

- This architecture is applied when input data are to be **transformed through a series of <u>computational or manipulative components</u>** into output data.
- A pipe-and-filter pattern (Figure above) has a set of components, called ***filters, connected by pipes that transmit data from one component to the next***.
- ***Each*** **filter** works independently of those components upstream and downstream, is **designed** to expect **data input of a certain form, and produces data output to the next filter of a specified form.**
- The **filter** does **<u>not require knowledge</u>** of the workings of its **<u>neighbouring filters</u>.**

Data Flow Architecture(contd.,)

- If the data flow **degenerates into a single line** of transforms, it is termed **batch sequential**.
- This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.
- Examples of pipes and filter
  - **lex/yacc-based compiler (scan, parse, generate…)**
  - **Unix pipes**
  - **Image / signal processing**
- Examples of batch processing
  - **Payroll computations**
  - **Tax reports**

Call and Return Architecture



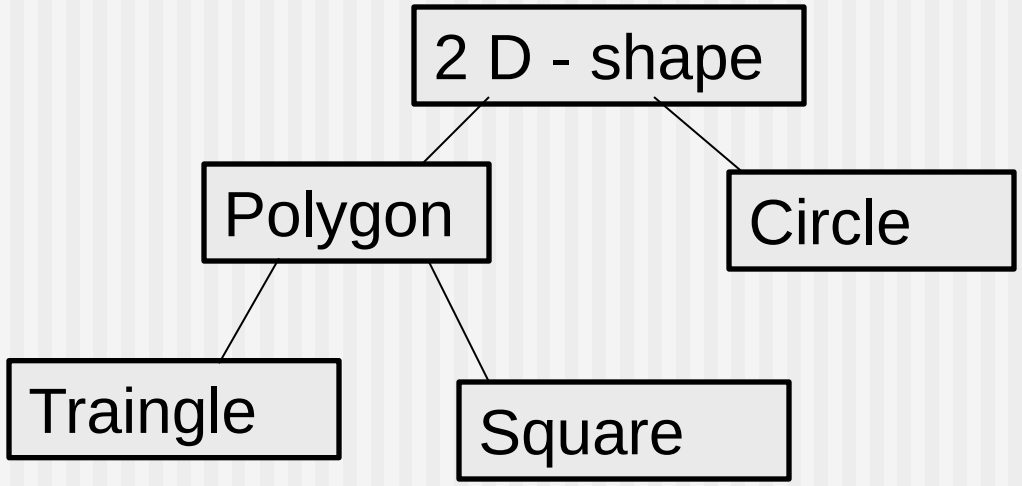
Call and Return Architecture(contd.,)

- This architectural style **enables** you to achieve a program structure that is relatively **easy to modify and scale**.
- A number of substyles exist within this category:
  - ***Main program/subprogram architectures***. *This classic program structure* decomposes function into a control hierarchy where a **"main" program invokes a number of program components that in turn may invoke still other components**. Figure above illustrates an architecture of this type.
  - ***Remote procedure call architectures***. *The components of a main* program/subprogram architecture are **distributed across multiple computers on a network.**

Call and Return Architecture(contd.,)

- Example dynamic binding object 2D-Shape

- 2D-Shape defines method Perimeter(): Real
  - Same operation, but implemented differently for Triangle, Square, and Circle.
  - If an object A is of any subtype of 2D-Shape a consumer may call A.Perimeter(), and the appropriate method will be chosen at run-time

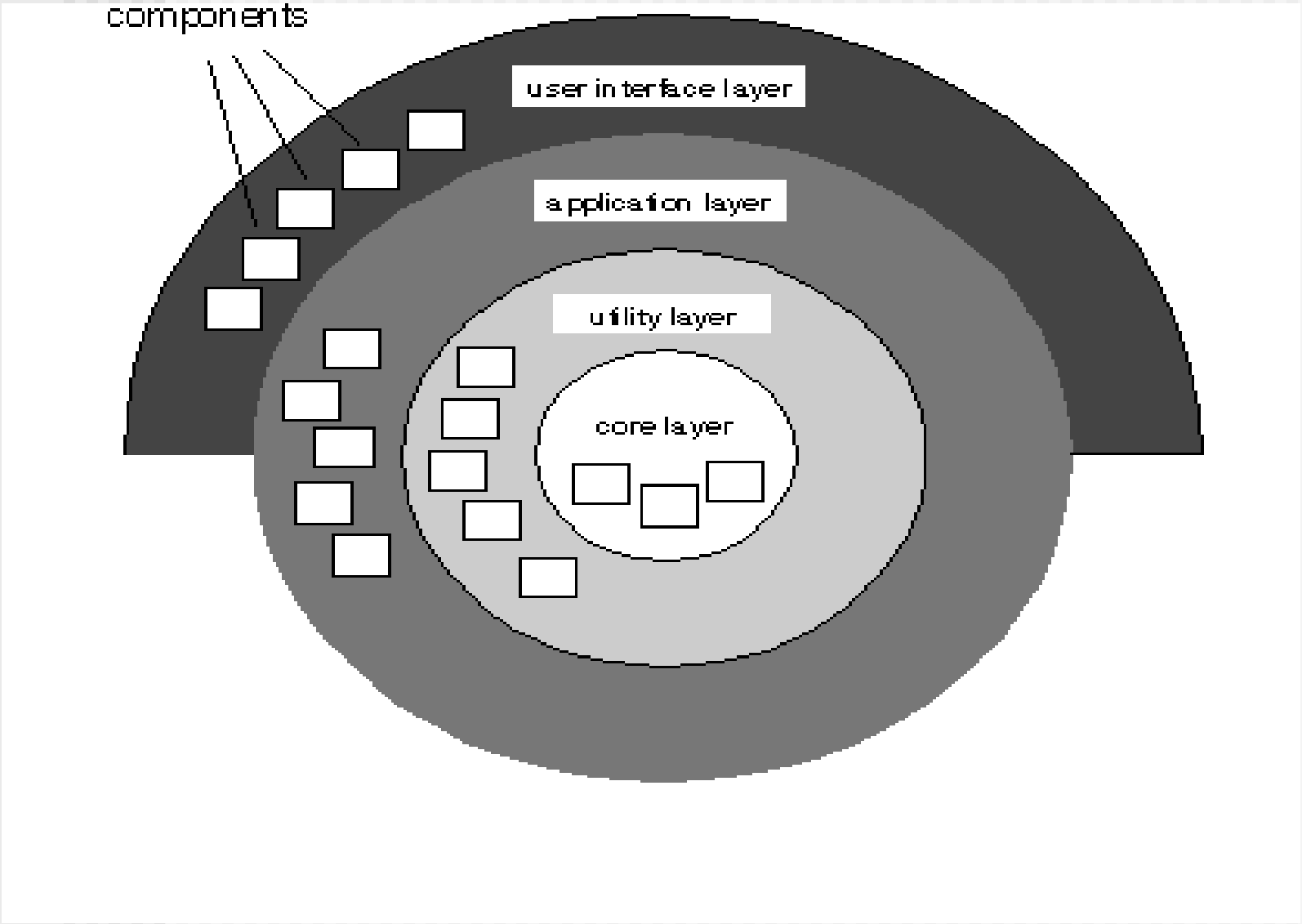


Object Oriented Style

Object Oriented Style(contd.,)

- The object-oriented paradigm, emphasizes **the bundling of data and methods to manipulate and access that data** (Public Interface).
- **Components of a system summarize data and the operations that must be applied to manipulate the data.**
- **Communication and coordination** between components is accomplished via **message passing**.
- **Examples** of object-oriented architecture systems are the **IBM System 38, the Carnegie-Mellon experimental C.**

Layered Architecture



Layered Architecture(contd.,)

- The basic structure of a layered architecture is illustrated in Figure above.
- A number of different layers are defined, each accomplishing **operations that progressively become closer to the machine instruction set**.
- At the ***outer layer***, **components service user interface operations.**
- At the ***inner layer***, **components perform operating system interfacing.**
- ***Intermediate layers*** **provide utility services and application software functions.**
- **Eg: OSI Network layer** is an example of Layered Architecture

- These architectural styles are only a small subset of those available.

- Once requirements engineering uncovers the characteristics and constraints of the system to be built, **the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen.**
- In many cases, **more than one pattern** might be appropriate and alternative architectural styles can be designed and evaluated.
- For example, a **layered style-**appropriate for most systems-can be **combined** with a **data-centered** architecture in many database applications.

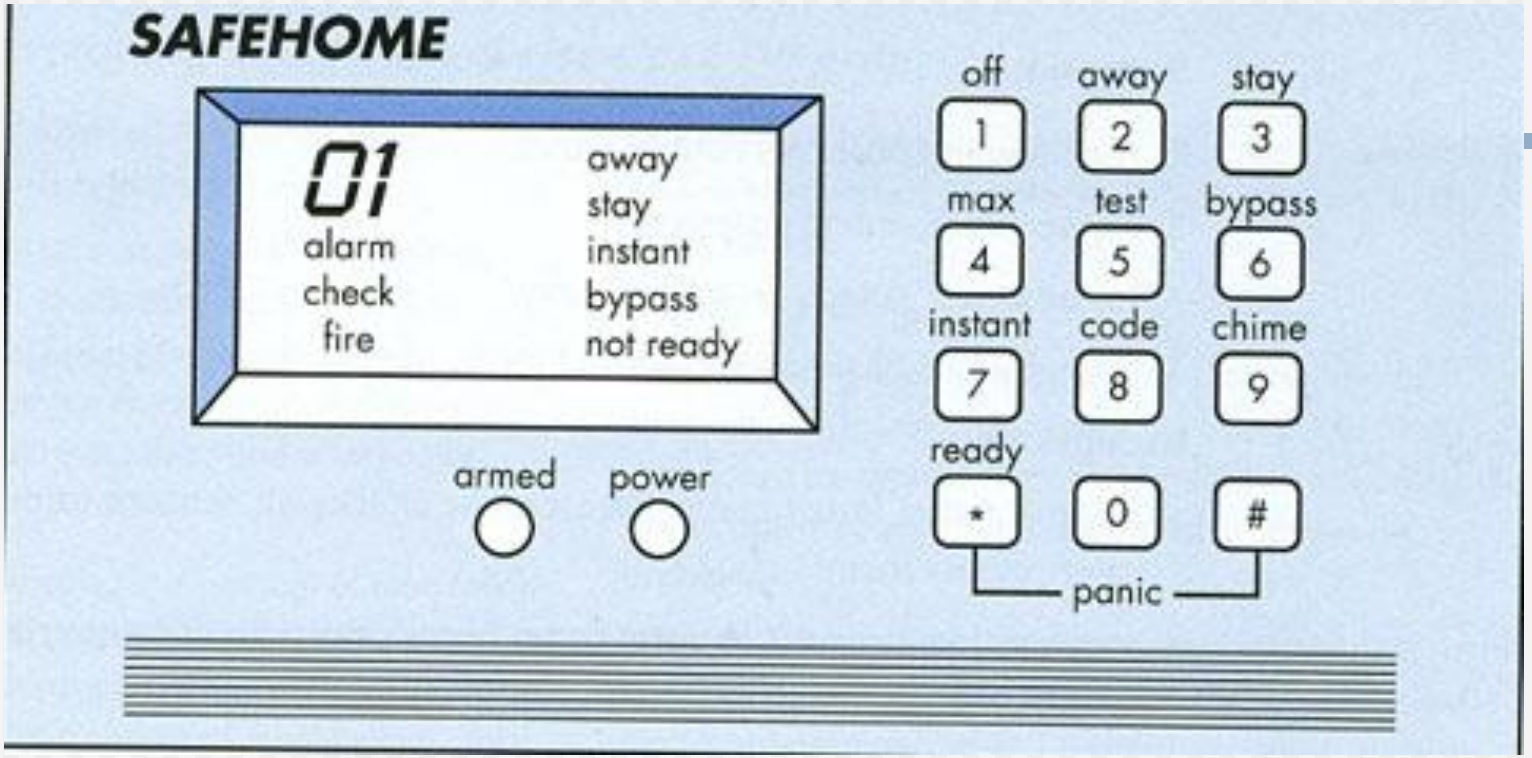
Architectural Patterns

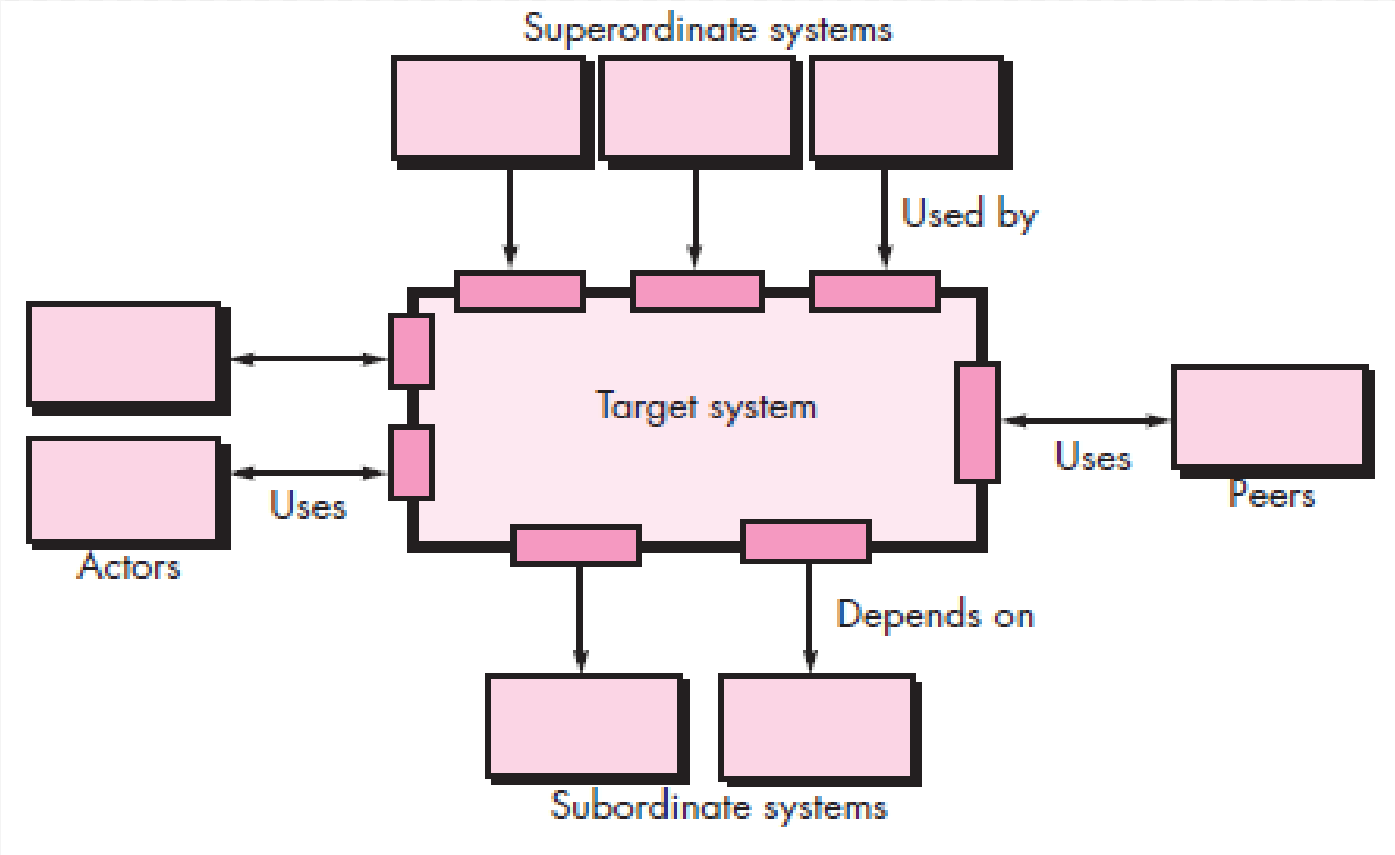
- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database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

Architectural Design

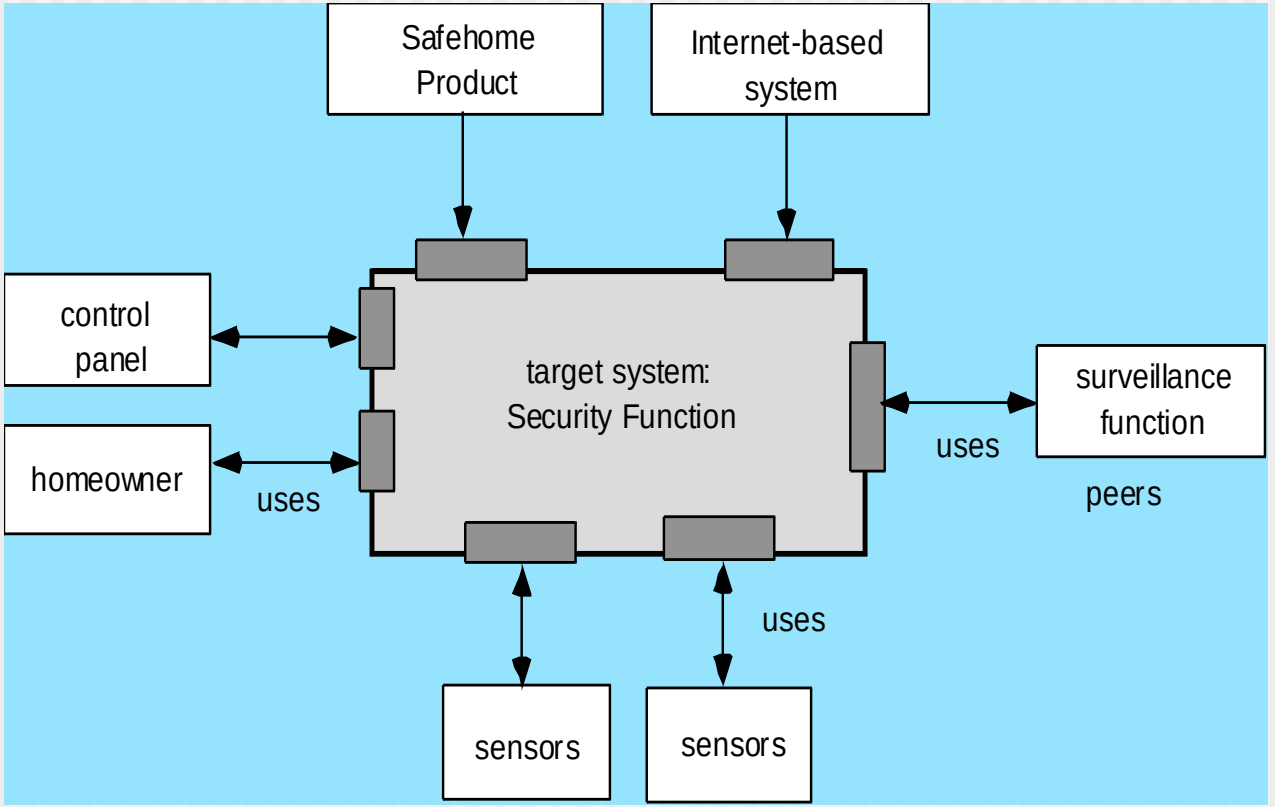
- The software to be developed must be placed into context
  - the design should **define the external entities** (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
  - **An archetype is an abstraction (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.
- This process continues iteratively **until a complete architectural structure has been derived.**

Safehome Security Function




Superordinate systems
Used by
Target system
Uses
Peers
Uses
Actors
Depends on
Subordinate systems

Architectural Context of Safehome security function



Architectural Context(contd.,)

- At the architectural design level, **a software architect uses an *architectural context diagram*** **(**ACD) to model the manner in which ***software interacts with entities external to its boundaries.***
- The generic structure of the architectural context diagram is illustrated in Figure above.
- Referring to the figure, systems that interoperate with the *target system (the* system for which an architectural design is to be developed) are represented as:

Architectural Context(contd.,)

- ***Superordinate systems***—*those systems that* ***use the target system*** *as part of* **some higher-level processing** scheme.
- ***Subordinate systems***—*those systems that are used by the target system* ***and*** **provide data or processing** that are necessary to complete target system functionality.
- ***Peer-level systems***—*those systems that interact on a peer-to-peer basis (i.e.,* **information is either produced or consumed** by the peers and the target system.
- ***Actors***—*entities (people, devices) that interact with the target system by* producing or consuming **information that is necessary for requisite processing.**

- Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Architectural Context(contd.,)

- Consider the home security function of the *SafeHome product.*

- *The* ***overall SafeHome product controller and the Internet-based*** **system** are both **superordinate** to the security function and are shown above the function in Figure above.
- The **surveillance function** is a ***peer system*** *and uses (is used by)* the home security function in later versions of the product.
- The **homeowner and control panels** are **actors** that are both producers and consumers of information used/produced by the security software.
- Finally, **sensors** are used by the security software and are shown as **subordinate** to it.

Archetypes

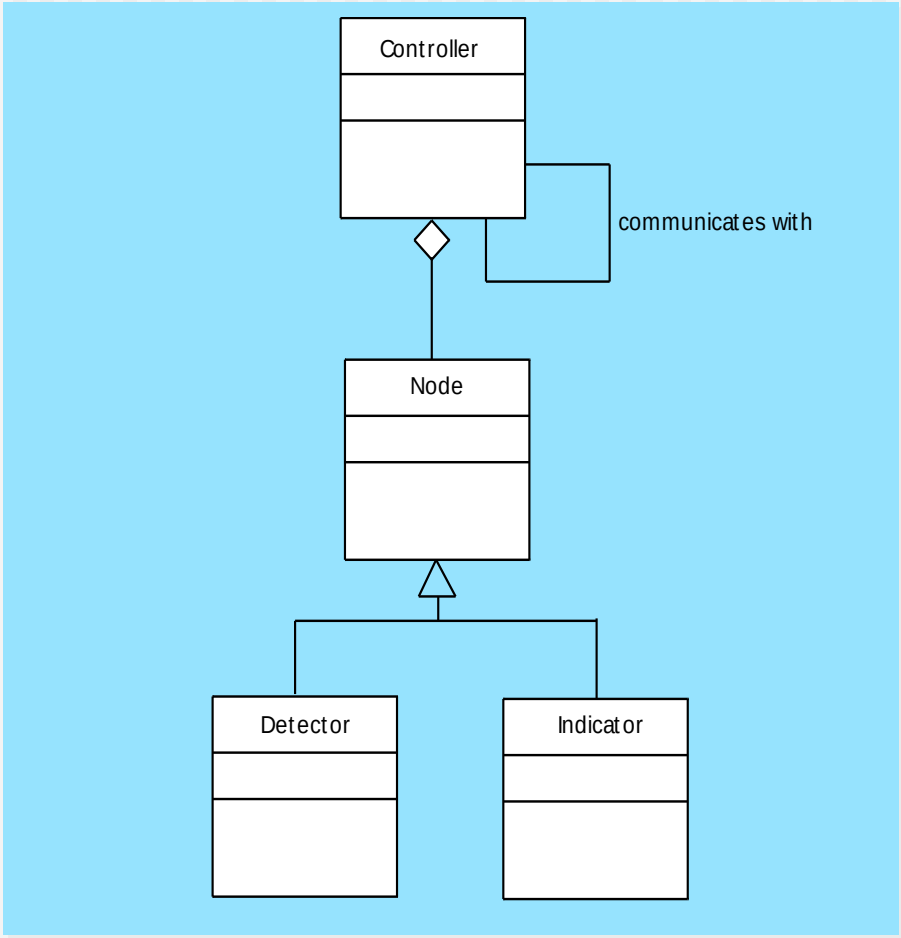


Figure 10.7 UML relationships for SafeHomesecurity function archetypes (adapted from [BOS00])

Defining 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.
- A relatively small set of archetypes is required to design even relatively complex systems.
- The **target system architecture is composed of these archetypes, which represent stable elements of the architecture** but may be instantiated many different ways based on the behaviour of the system.
- Archetypes can be derived by examining the analysis classes defined as part of the requirements model.

Defining Archetypes(contd.,)

- In the *SafeHome* home security function, archetypes are defined as follows:
  - **Node. Represents a cohesive collection of *input and output elements* of** the home security function. For example a node might be comprised of (1) various **sensors** and (2) a variety of **alarm (output) indicators**.
  - **Detector. An abstraction that encompasses all sensing equipment that feeds information** into the target system.
  - **Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell)** for indicating that an alarm condition is occurring.
  - **Controller. An abstraction that depicts the mechanism that allows the** arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.
- The archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds.
- **Detector** might be refined into a class hierarchy of sensors.

Refining the Architecture into Components

- **As the software architecture is refined into components**, the **structure of the system begins to emerge**.
- **Sources of Architectural elements**
- **Analysis classes represent entities** within the application (business) domain that must be addressed within the software architecture.
- Hence, the **application domain** is one source for the derivation and refinement of components.
- Another source is the **infrastructure domain.**
- The architecture must accommodate many infrastructure components that enable application components but **have no business connection to the application domain.**
  - For example, **memory management components, communication components, data base components, and task management components are often integrated into the software architecture.**

Refining the Architecture into Components(contd.,)

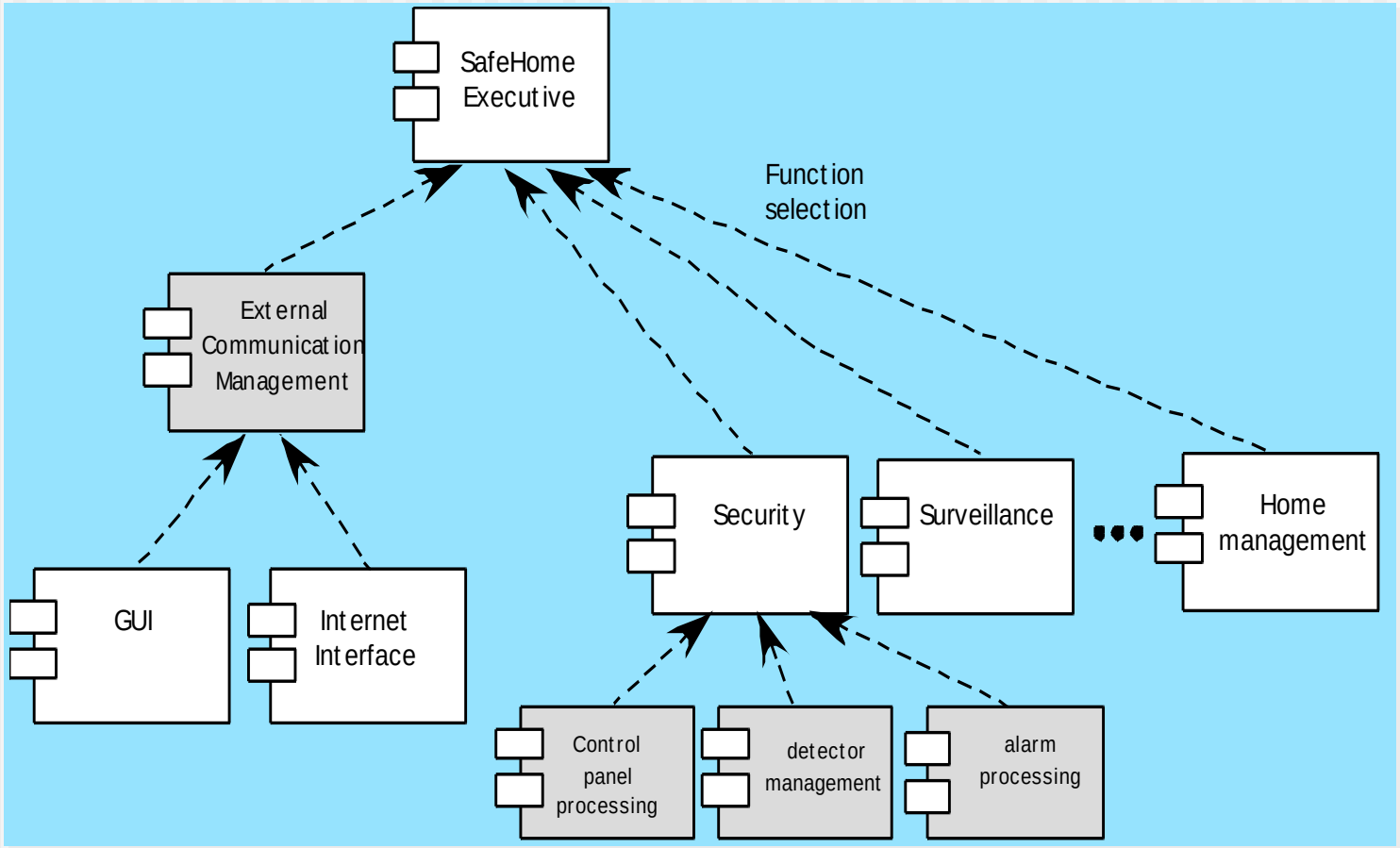
- The *SafeHome home security function example, define the* set of top-level components that address the following functionality:
  - ***External communication management****—coordinates* ***communication*** *of the* security function **with external entities** such as other ***Internet-based systems and external alarm notification.***
  - ***Control panel processing****—manages* ***all control panel functionalities.***
  - ***Detector management****—coordinates access to* ***all detectors attached to the*** **system.**
  - ***Alarm processing****—****verifies and acts on all alarm conditions****.*

Refining the Architecture into Components(contd.,)

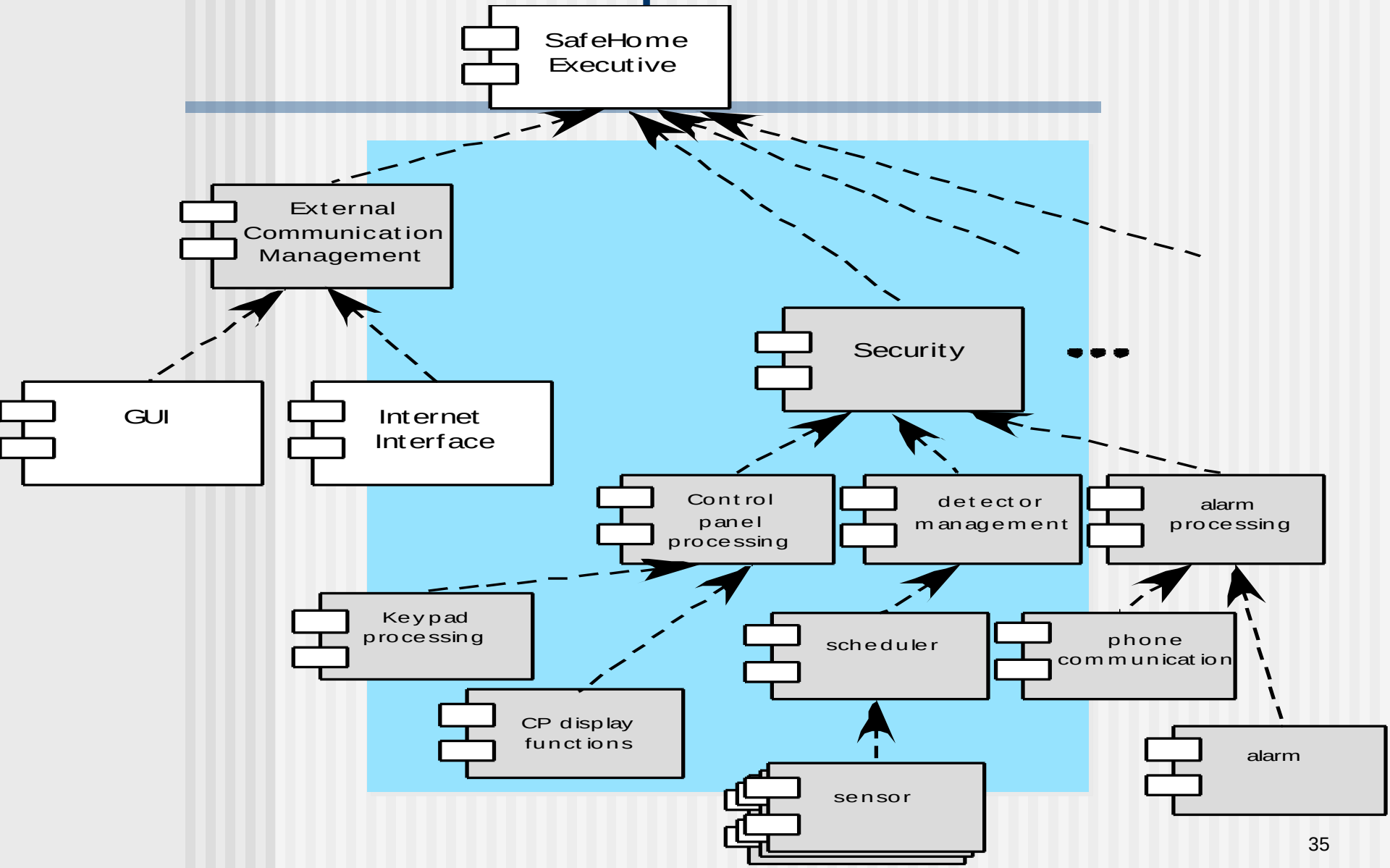
- Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome architecture.*
- ***Design classes****, with appropriate* **attributes and operations**, would be **defined for each**.
- **The design details of all attributes and operations would not be specified until component-level design**

- **Transactions are acquired by *external communication management*** as they move in from components that process the *SafeHome GUI and the internet interface.*
- This information is managed by a *SafeHome executive component* that selects the appropriate product function (in this case security).
- The ***control panel processing component interacts with the homeowner*** *to arm/disarm the security* function.
- The ***detector management component polls sensors*** *to detect an* alarm condition,
- The ***alarm processing component produces output*** *when an* alarm is detected.

Component Structure



Refined Component Structure



Describing Instantiations of the System

- The architectural design that has been modelled to this point is **still relatively high level**.
- The context of the system has been represented, **archetypes that indicate the important abstractions within the problem domain have been defined**, the overall structure of the system is apparent, and the major software components have been identified.
- Further refinement, since design is iterative, is still necessary.

Describing Instantiations of the System(contd.,)

- To accomplish this, an actual instantiation of the architecture is developed.
- By this, the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
- **Components shown** in Figure above **are elaborated to show additional detail.**
- For example, the ***detector management component interacts with a scheduler*** *infrastructure* component that **implements polling of each *sensor object*** *used by the security* system.

Agility and Architecture

- To avoid rework, user stories are used to create and evolve an architectural model (walking skeleton) before coding
- Hybrid models which allow software architects contributing users stories to the evolving storyboard
- Well run agile projects include delivery of work products during each sprint
- Reviewing code emerging from the sprint can be a useful form of architectural review