SOFTWARE DESIGN

- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- Design principles establish an overriding philosophy that guides you in the design work you must perform.
- Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.
- Design is pivotal to successful software engineering.
- The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.

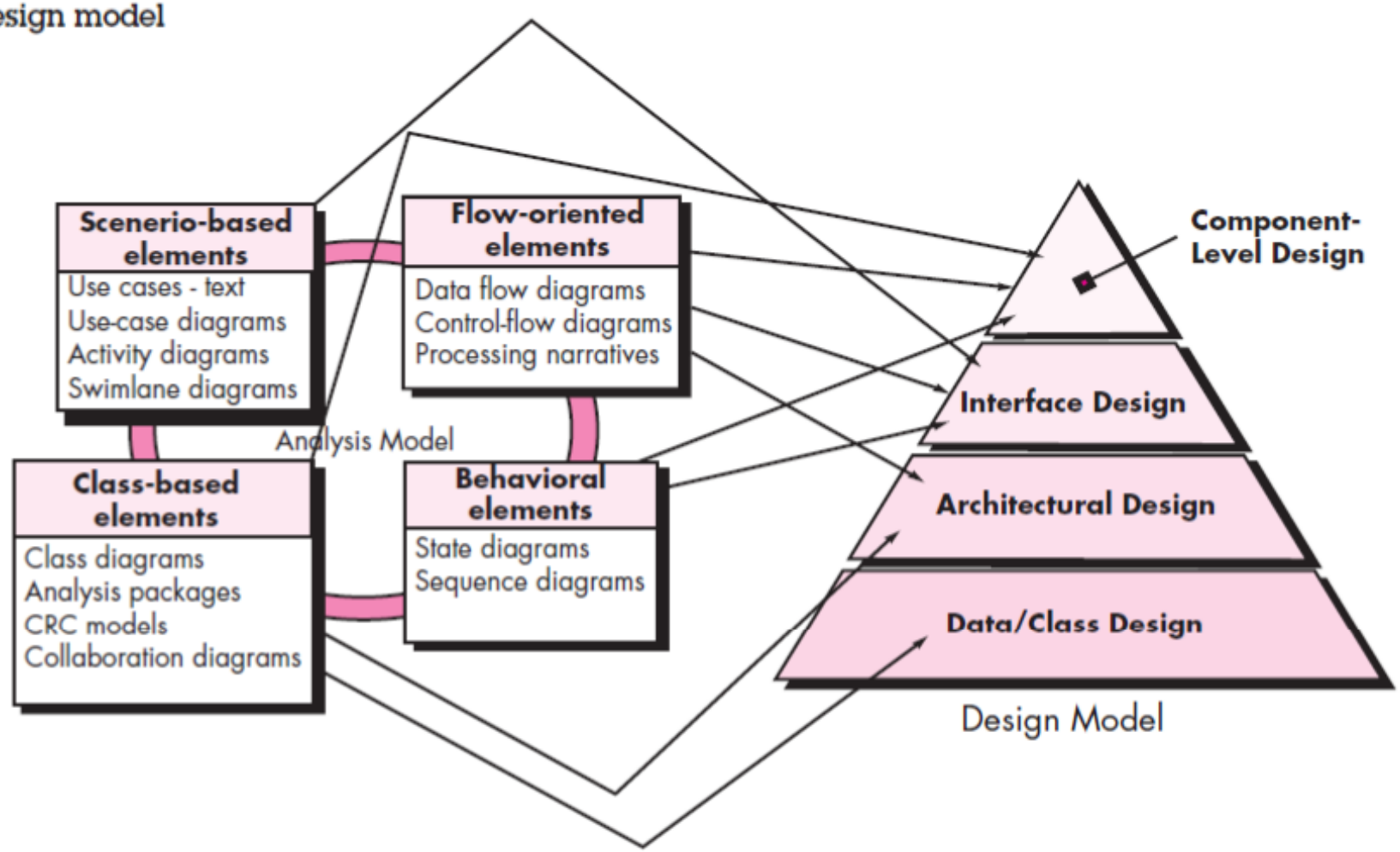
- Firmness: A program should not have any bugs that inhibit its function.
- Commodity: A program should be suitable for the purposes for which it was intended.
- Delight: The experience of using the program should be a pleasurable one.

- Software design changes continually as new methods, better analysis, and broader understanding evolve.
- Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines.
- However, methods for software design do exist, criteria for design quality are available, and design notation can be applied.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

Translating the requirements model into the design model



- Data/Class 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.

- Architectural 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.

- Interface Design:

- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior.
- Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

- Component-Level 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 behavioral models serve as the basis for component design.

Design versus Coding

The scene: Jamie's cubicle, as the team prepares to translate requirements into design.

The players: Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

The conversation:

Jamie: You know, Doug [the team manager] is obsessed with design. I gotta be honest, what I really love doing is coding. Give me C++ or Java, and I'm happy.

Ed: Nah . . . you like to design.

Jamie: You're not listening; coding is where it's at.

Vinod: I think what Ed means is you don't really like coding; you like to design and express it in code. Code is the language you use to represent the design.

Jamie: And what's wrong with that?

Vinod: Level of abstraction.

Jamie: Huh?

Ed: A programming language is good for representing details like data structures and algorithms, but it's not so good for representing architecture or component-to-component collaboration . . . stuff like that.

Vinod: And a screwed-up architecture can ruin even the best code.

Jamie (thinking for a minute): So, you're saying that I can't represent architecture in code . . . that's not true.

Vinod: You can certainly imply architecture in code, but in most programming languages, it's pretty difficult to get a quick, big-picture read on architecture by examining the code.

Ed: And that's what we want before we begin coding.

Jamie: Okay, maybe design and coding are different, but I still like coding better.

DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.
- Initially, the blueprint depicts a holistic view of software.
- That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

- Software Quality Guidelines and Attributes:

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

➢ The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

➢ The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

➢ The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

- Quality Guidelines:

- In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

- Quality Attributes:

- Hewlett-Packard 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.

The Evolution of Software Design:

- The evolution of software design is a continuing process that has now spanned almost six decades.
- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner.
- Procedural aspects of design definition evolved into a philosophy called structured programming.
- Later work proposed methods for the translation of data flow or data structure into a design definition.
- Newer design approaches proposed an object-oriented approach to design derivation.
- More 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 aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

- A number of design methods, growing out of the work just noted, are being applied throughout the industry.
- Like the analysis methods in which each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality.
- Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

TASK SET

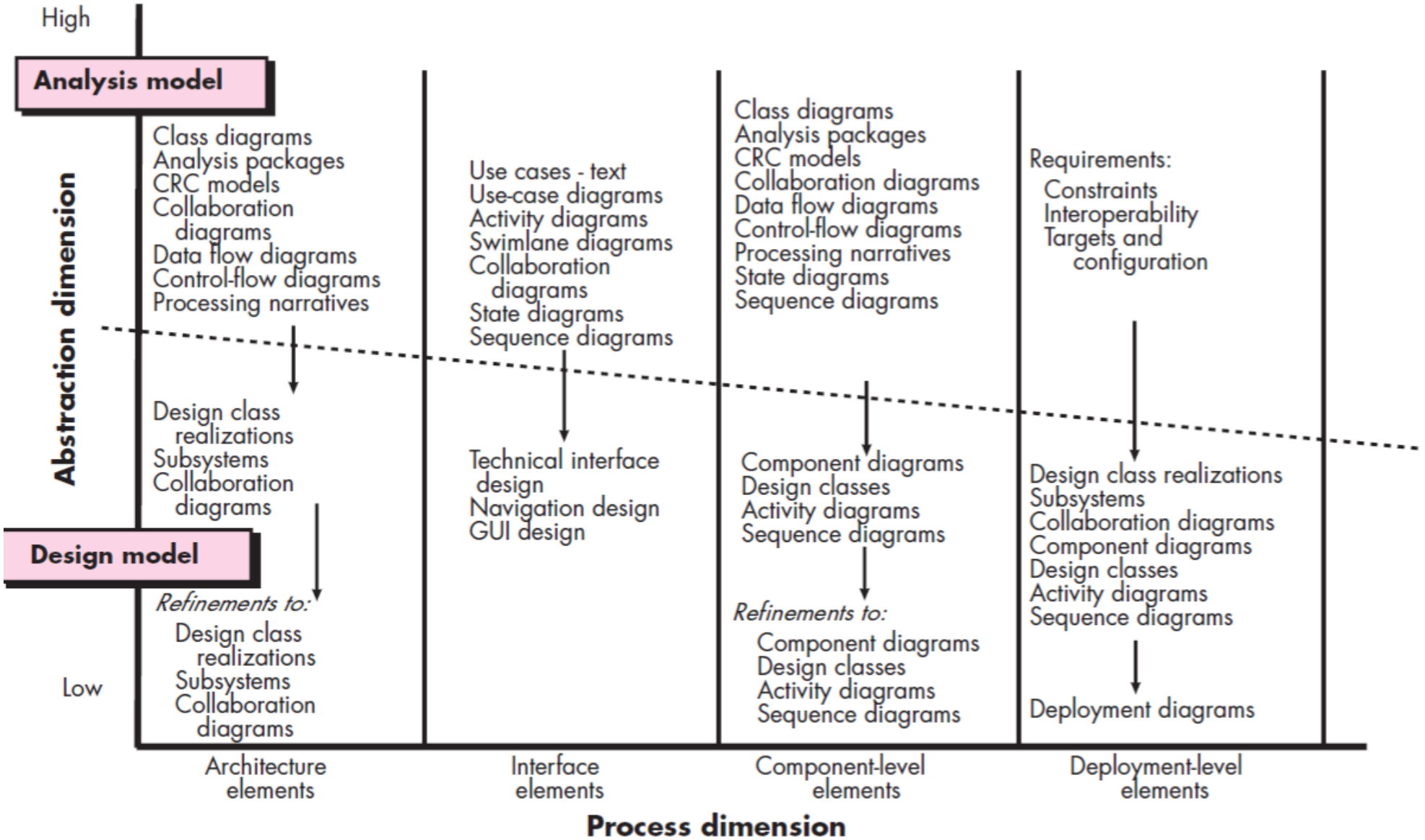
Generic Task Set for Design

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 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.

DESIGN MODEL

- Develop overall software architecture.
- Structure system into sub-systems.
- Design object-oriented software architecture.
- The design model can be viewed in two different dimensions as illustrated in below 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.
- Referring to Figure, the dashed line indicates the boundary between the analysis and design models.
- In some cases, a clear distinction between the analysis and design models is possible.
- In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.
- 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.

Dimensions of the design model



- Data Design Elements:

- Like other software engineering activities, 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).
- This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.
- The structure of data has always been an important part of software design.
- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the application level, the translation of a data model (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.
- In every case, data design plays an important role.

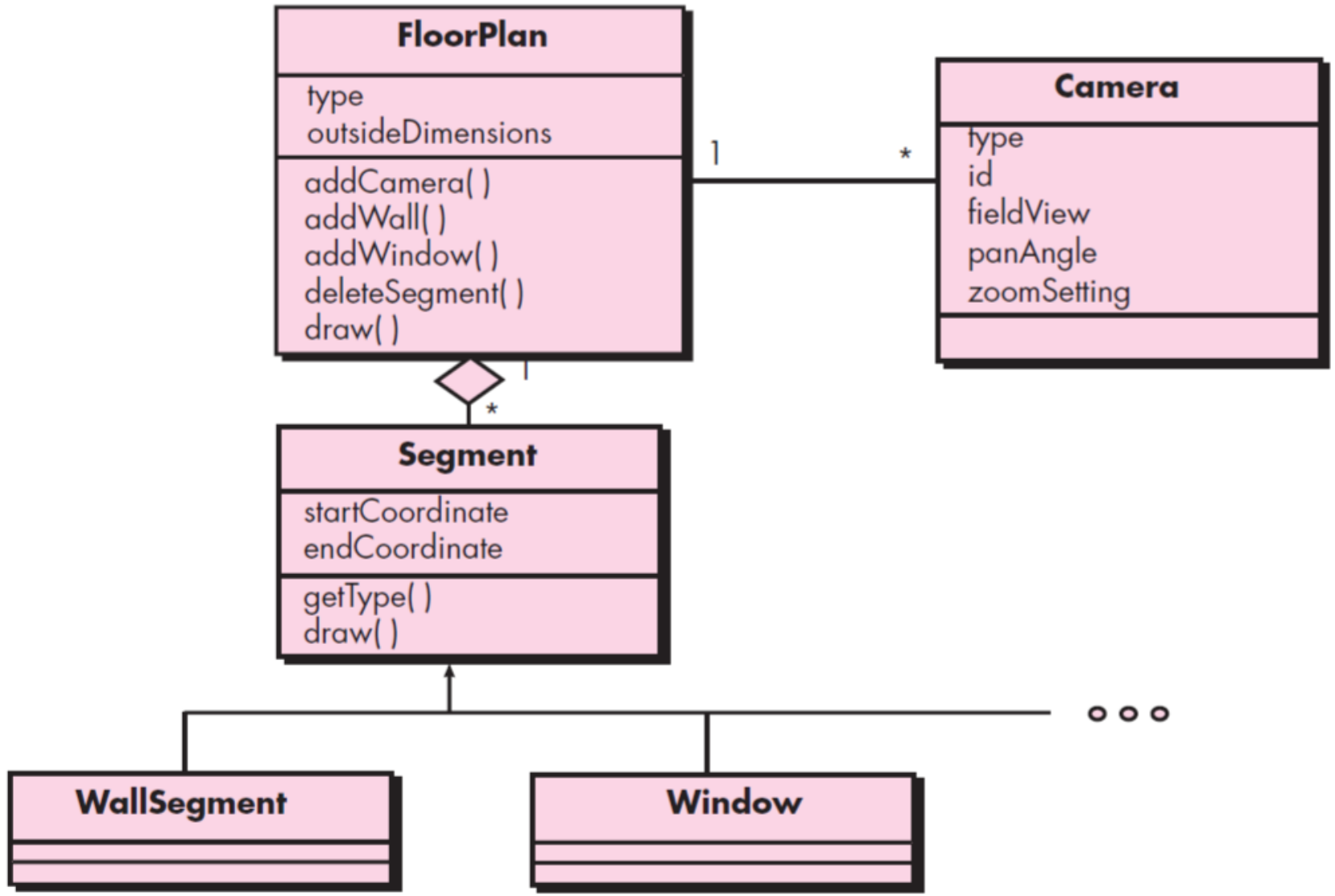
- Architectural Design Elements:

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

1. Information about the application domain for the software to be built;
2. Specific 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.

- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
- Each subsystem may have it's own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces).

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



- Interface Design Elements:

- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house.
- These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.
- They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed.
- In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan.
- The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

- 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.

- UI design (increasingly called usability design) is a major software engineering action and is considered.
- Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).
- In general, the UI is a unique subsystem within the overall application architecture.

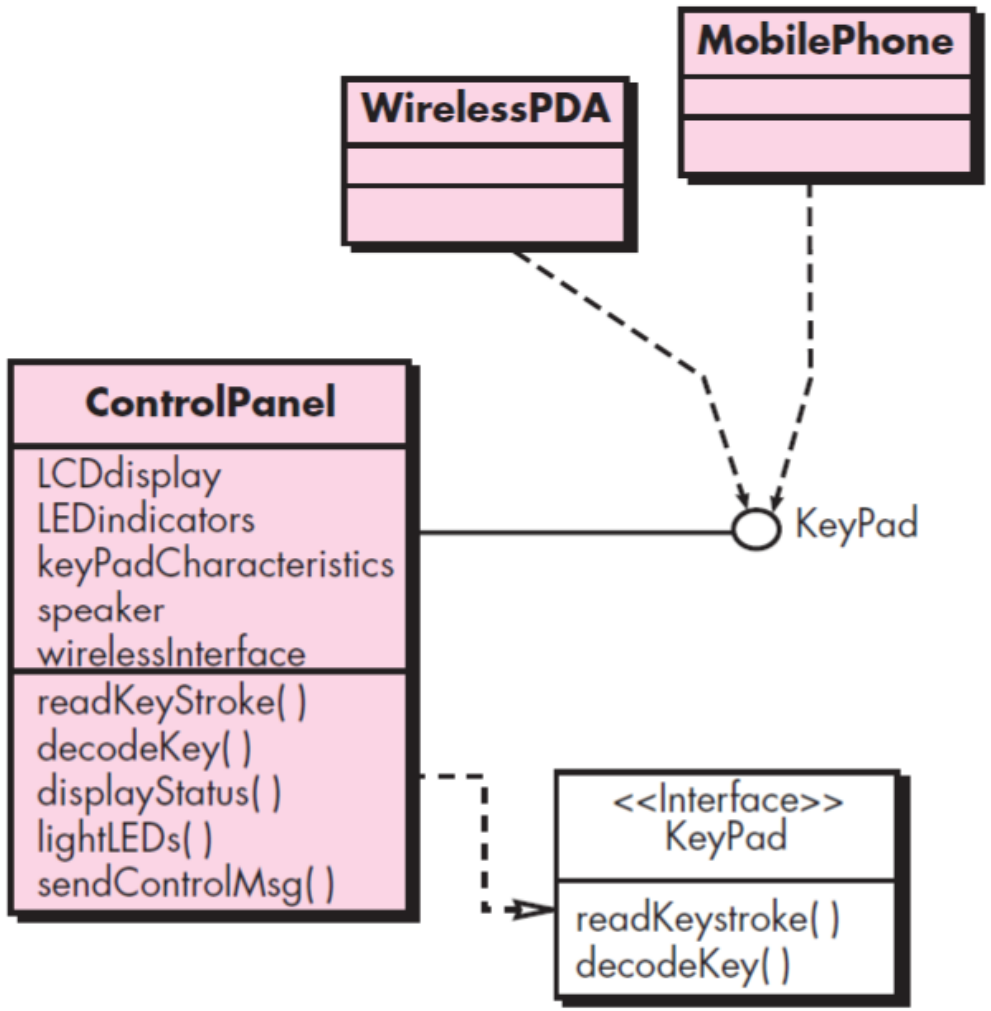
- The design of external interfaces requires definitive information about the entity to which information is sent or received.
- In every case, this information should be collected during requirements engineering and verified once the interface design commences.
- The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

- The design of internal interfaces is closely aligned with component-level design.
- Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes.
- Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested.
- 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 some cases, an interface is modeled in much the same way as a class.

- 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.

Interface representation for Control-Panel



- The ControlPanel class Figure ahown above provides the behavior associated with a keypad, and therefore, it must implement the operations readKeyStroke () and decodeKey ().
- If these operations are to be provided to other classes (in this case, WirelessPDA and MobilePhone), it is useful to define an interface as shown in the above figure.
- The interface, named KeyPad, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line.
- The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.
- The dashed line with an open triangle at its end indicates that the ControlPanel class provides KeyPad operations as part of its behavior.

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

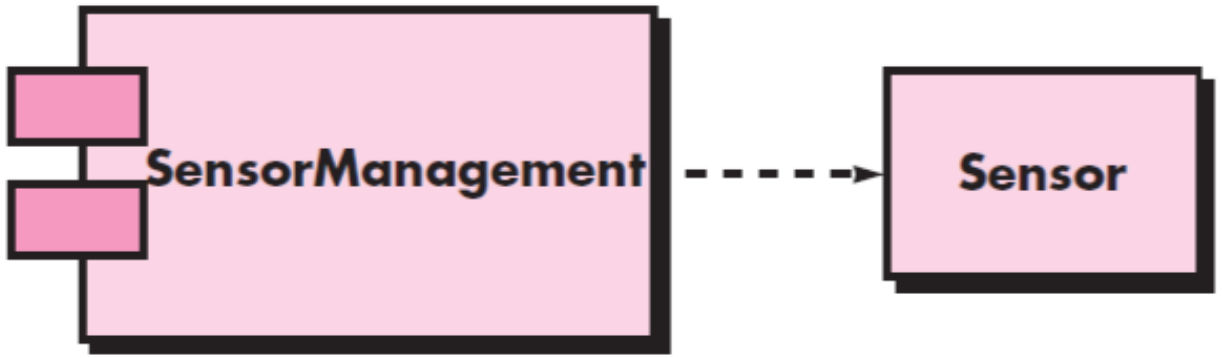
- Component-Level Design Elements:

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

- Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in below Figure.
- In this figure, 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.

- The design details of a component can be modeled at many different levels of abstraction.
- 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).

A UML component diagram

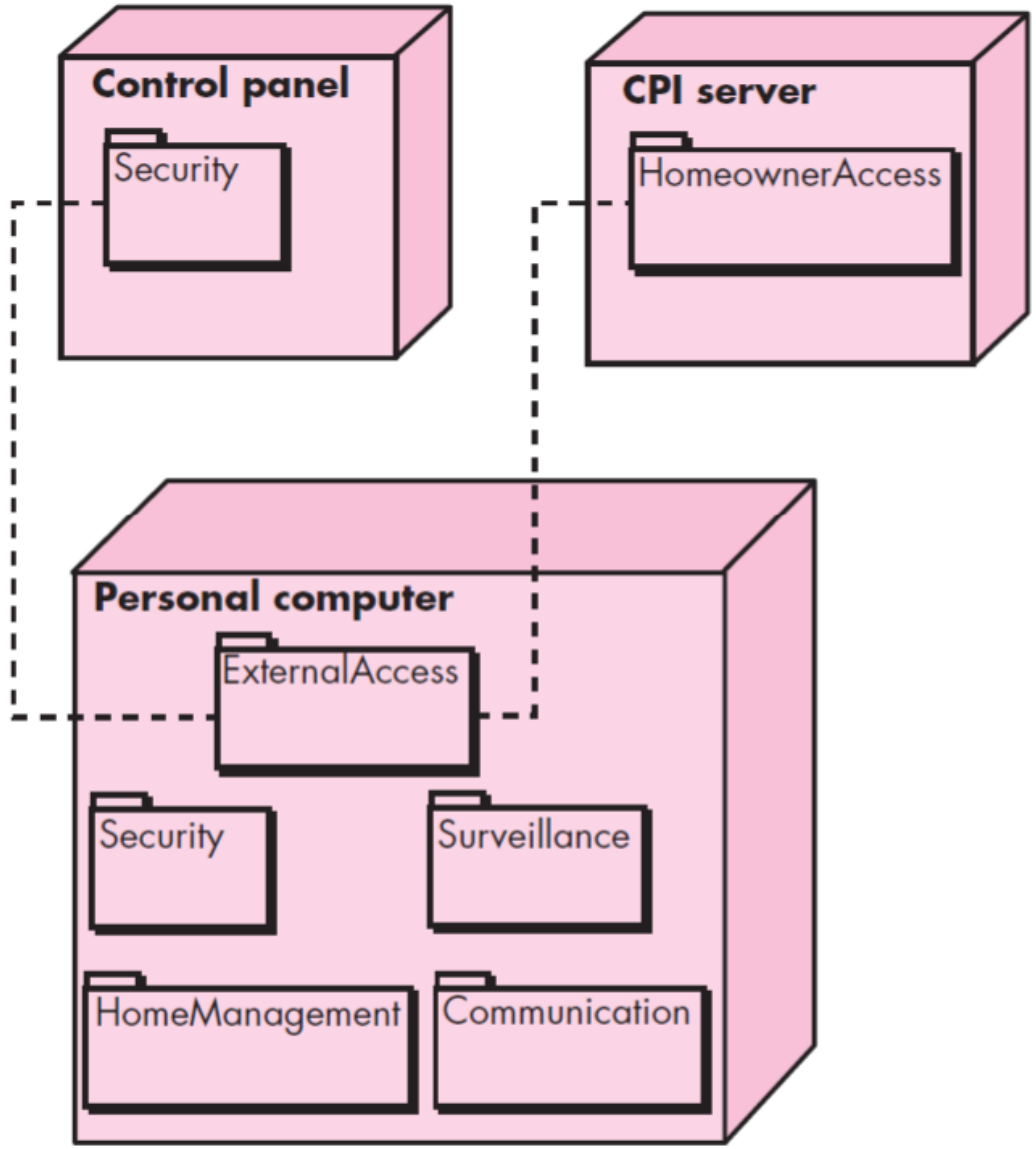


- # Deployment-Level Design Elements:

- 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).

- During design, a UML deployment diagram is developed and then refined as shown in below Figure.
- In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and others).
- 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.
- In addition, 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.

A UML deployment diagram



ARCHITECTURAL DESIGN

- It is the structure of software system like blue prints in building architecture.
- Software architecture consists of:

- ❑ Software components
- ❑ Details about data structures & algorithms
- ❑ Relationship among components
- ❑ Relationship like Data flow, control flow , dependencies etc.

- Three key reasons that software architecture is important:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
2. 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.
3. Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

- 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, 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.

- Grady Booch suggests the following architectural genres for software-based systems:

- Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- Commercial and nonprofit—Systems that are fundamental to the operation of a business enterprise.
- Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts.

- Devices—Systems that interact with the physical world to provide some point service for an individual.
- Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience.
- Financial—Systems that provide the infrastructure for transferring and managing money and other securities.
- Games—Systems that provide an entertainment experience for individuals or groups.
- Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- Industrial—Systems that simulate or control physical processes.
- Medical—Systems that diagnose or heal or that contribute to medical research etc.

• Architectural Styles:

- The software that is built for computer-based systems also exhibits one of many architectural styles.
- Each style describes a system category that encompasses

❑ A set of Components : (e.g., a database, computational modules) that perform a function required by a system.

❑ A set of Connectors : that enable “communication, coordination and cooperation” among components.

- Constraints : that define how components can be integrated to form the system; and
- Semantic models : that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

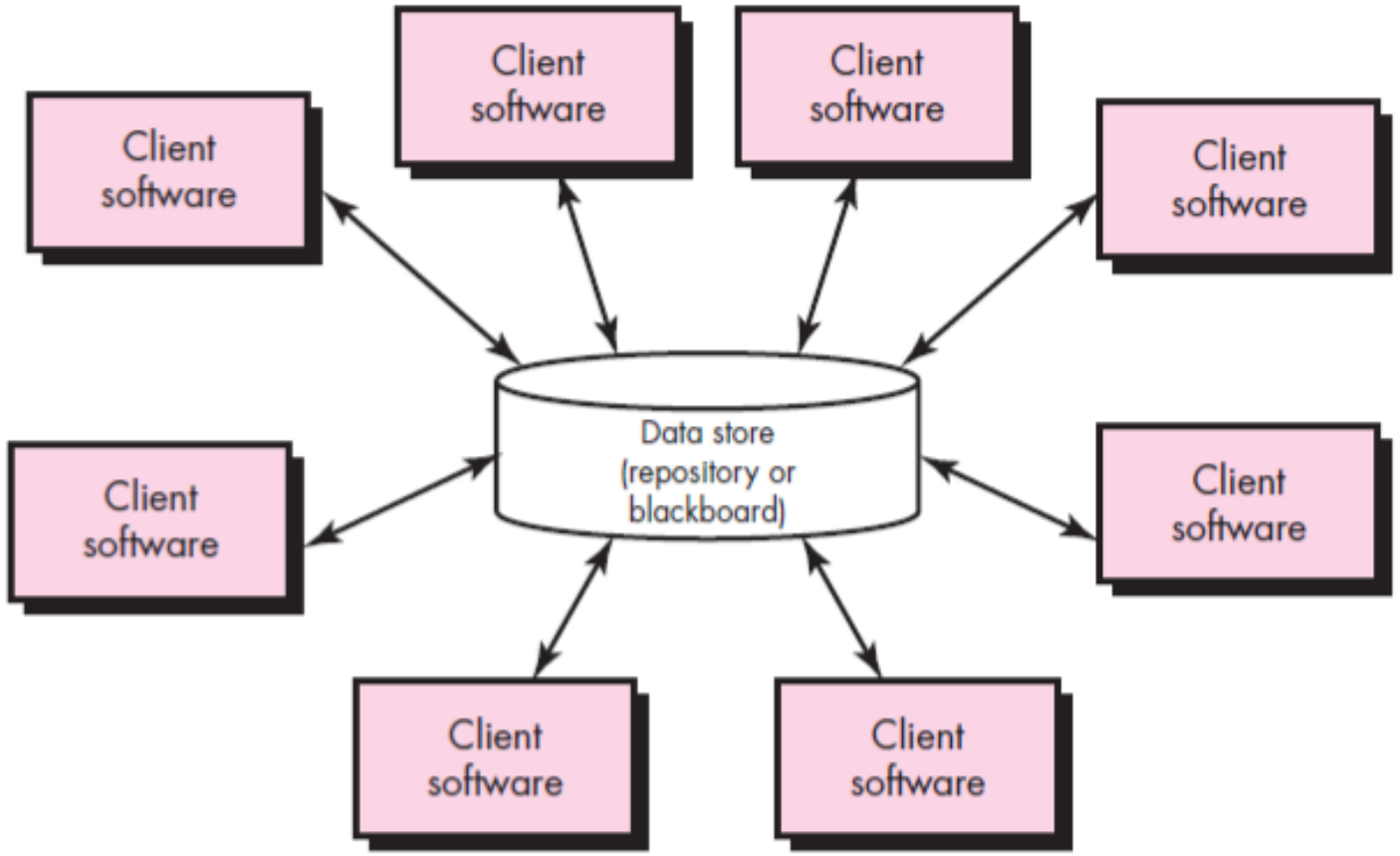
- Types of Architectural Styles:

- Data-Centered architecture
- Data Flow architecture
- Call & Return architecture
- Object-oriented architecture
- Layered architecture

❖Data-Centered architecture:

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

Data-centered architecture



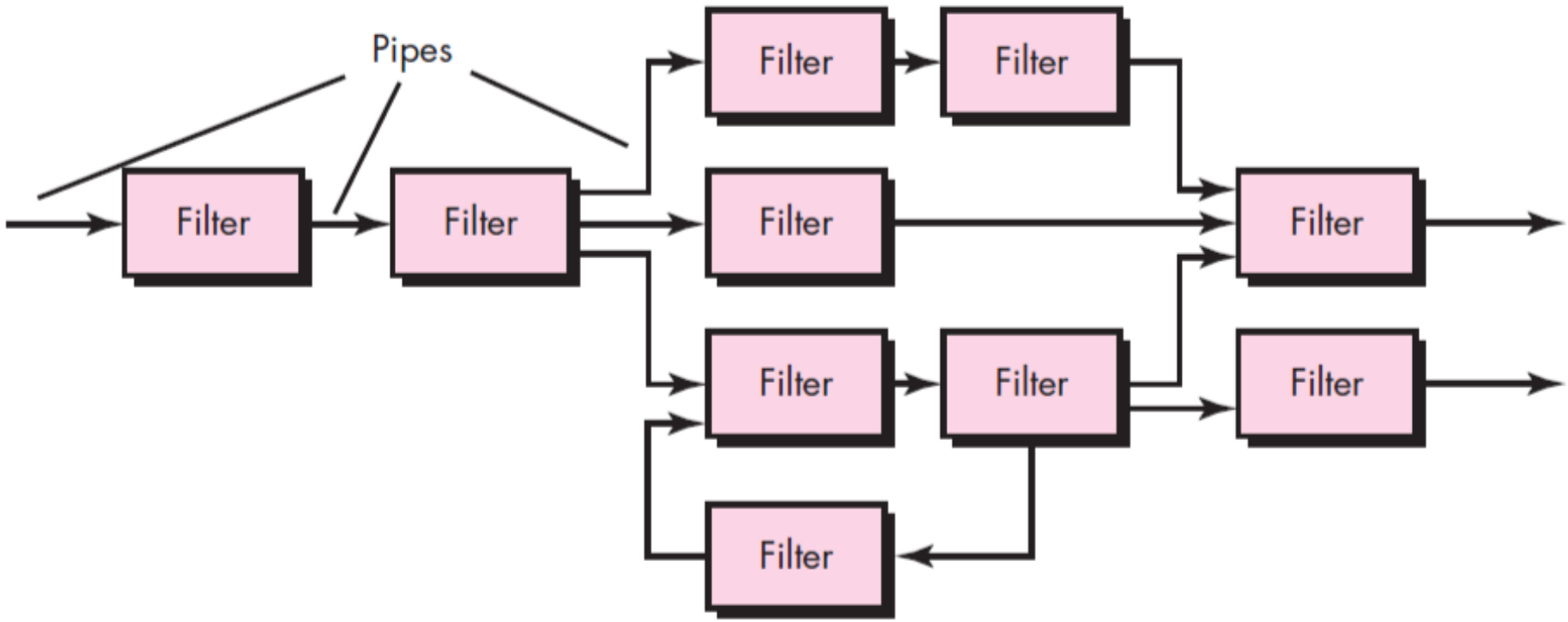
- Figure above illustrates a typical data-centered style.
- Client software accesses a central repository.
- That is, client software accesses the data independent of any changes to the data or the actions of other client software.
- A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.

- Data-centered 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).
- In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients).
- Client components independently execute processes.

❖Data-flow architectures:

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe-and-filter pattern in below Figure 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.
- However, the filter does not require knowledge of the workings of its neighboring filters.

Data-flow architecture



Pipes and filters

❖Call and return architectures:

- 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:

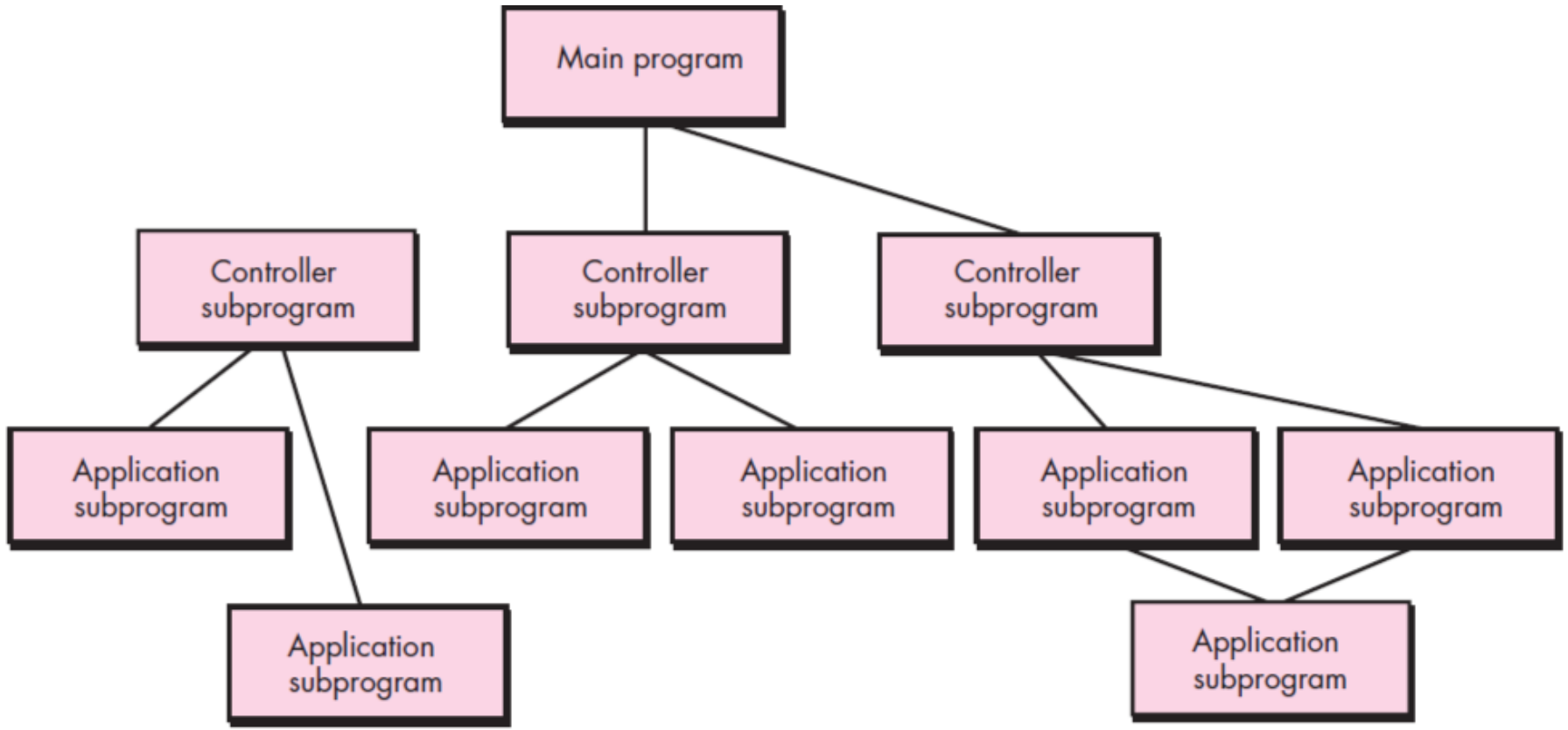
❖Main program/subprogram architectures:

- 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.
- Below Figure 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.

Main program/subprogram architecture



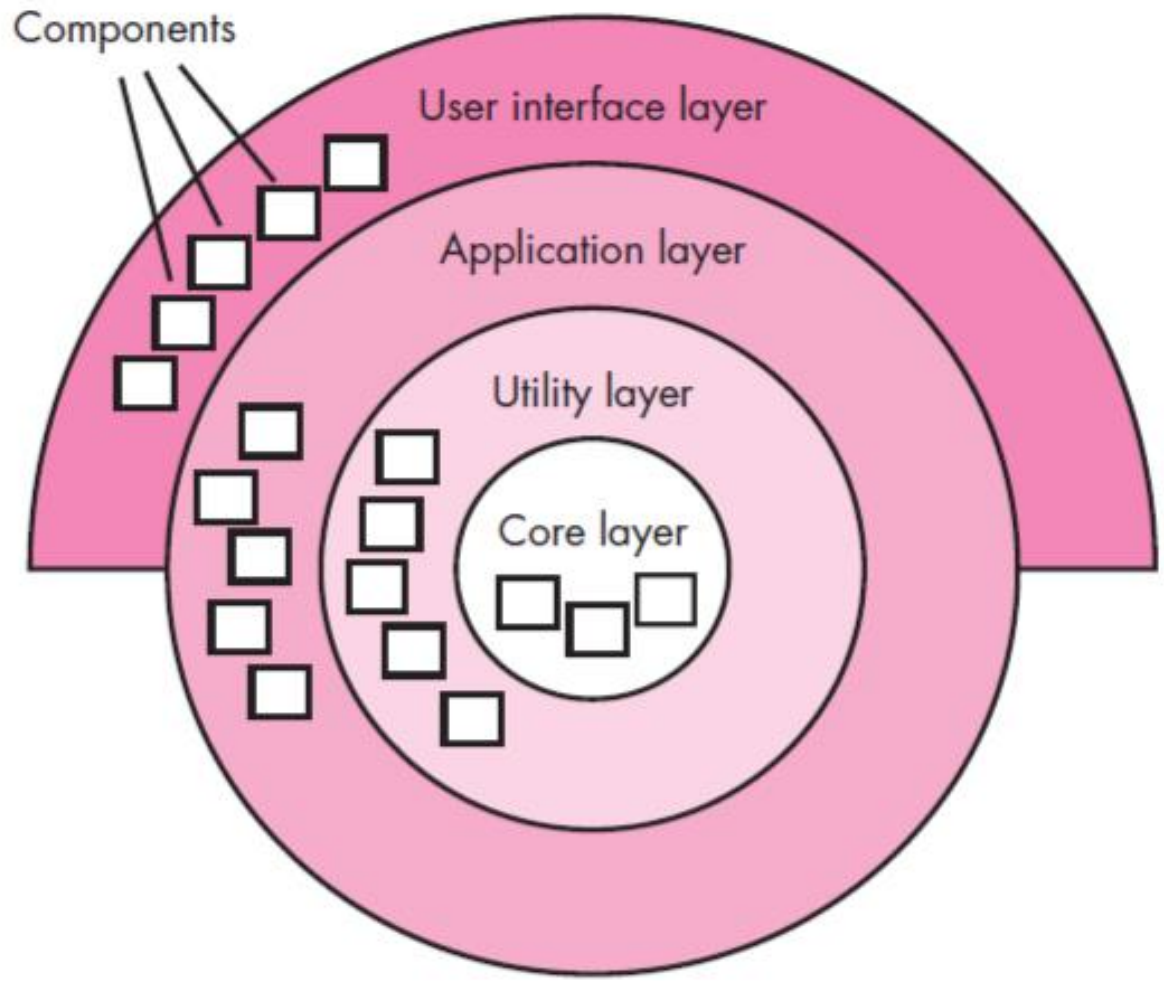
❖Object-oriented architectures:

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing.

❖Layered architectures:

- 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.

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.

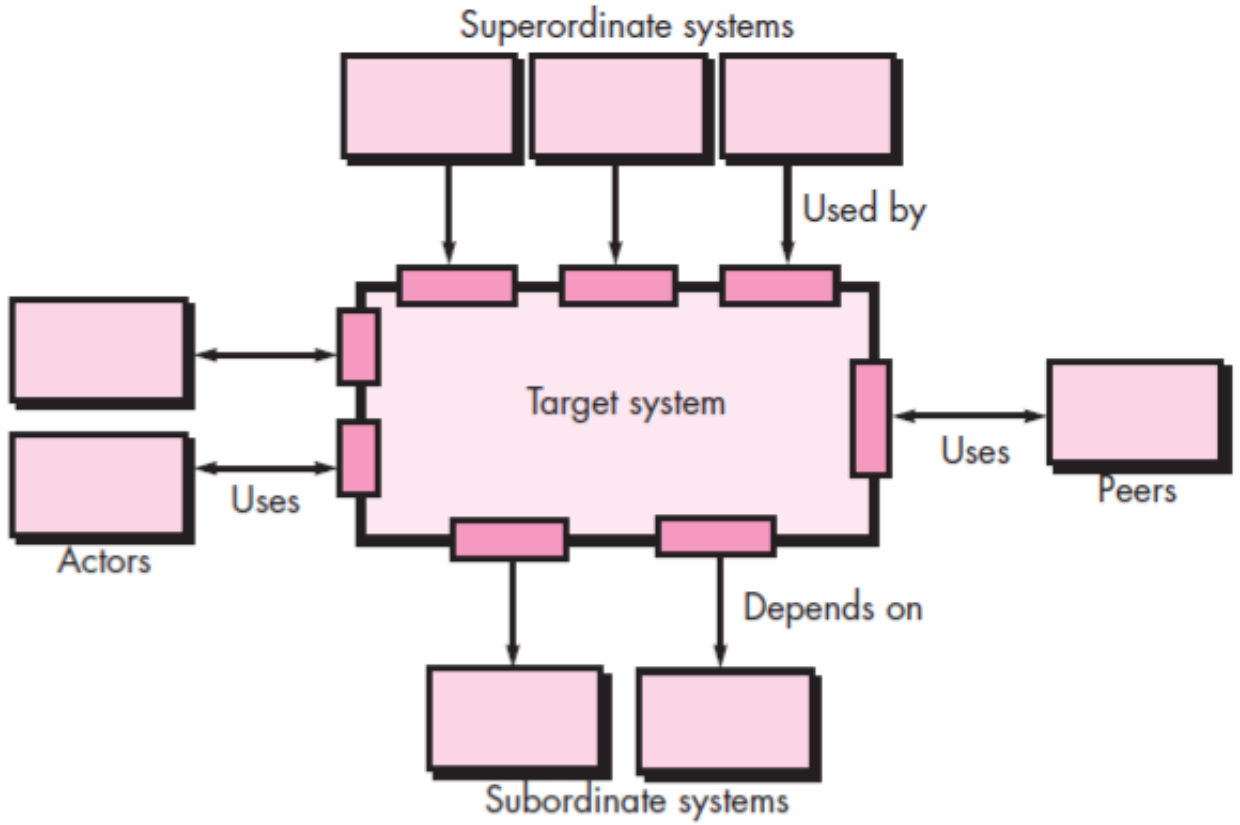
- ARCHITECTURAL DESIGN:

- As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.
- This information can generally be acquired from the requirements model and all other information gathered during requirements engineering.
- Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.
- An archetype is an abstraction (similar to a class) that represents one element of system behavior.
- The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.
- Therefore, 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.

- Representing the System in Context:

- 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 below Figure.

Architectural context diagram

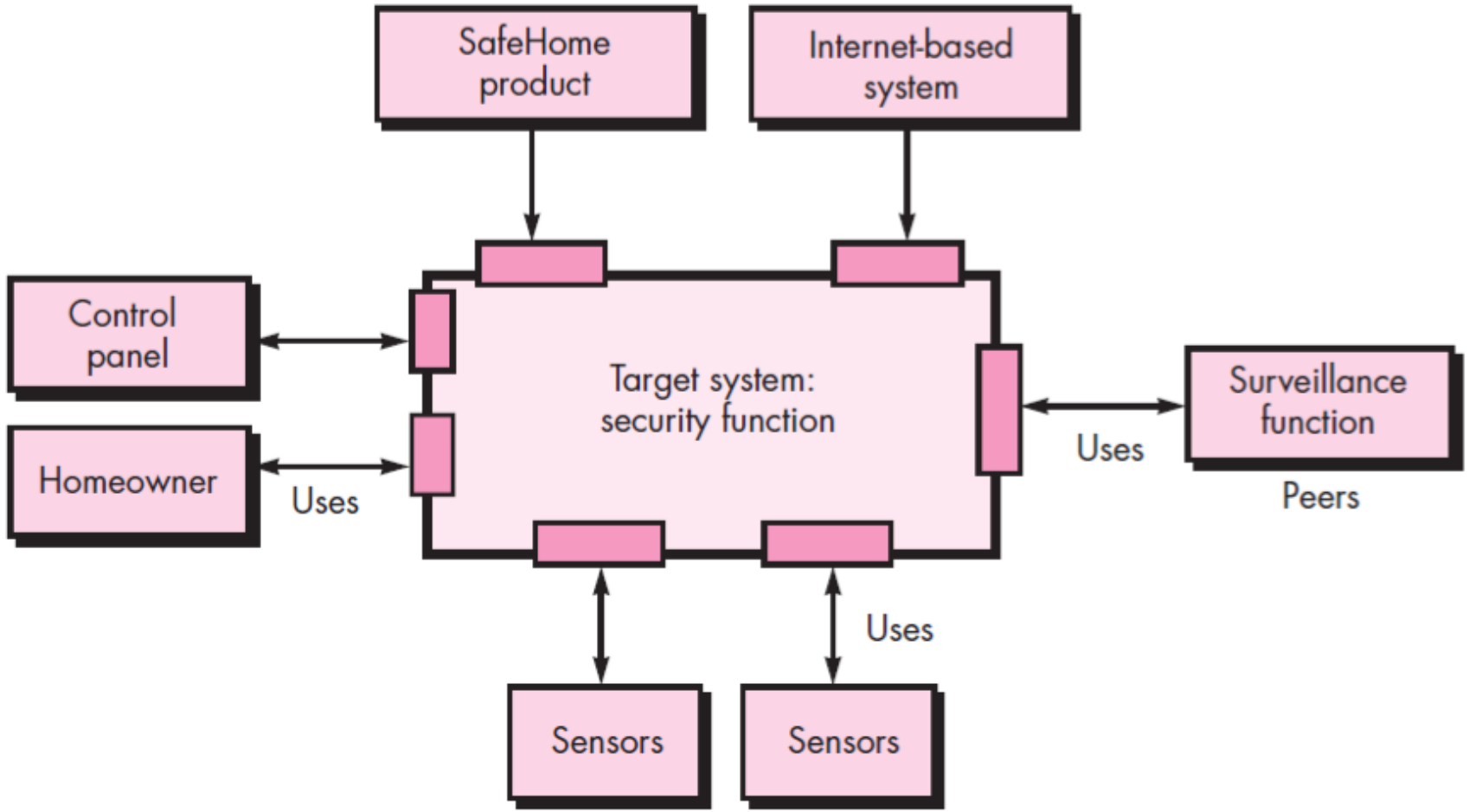


- Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as:

- 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 diagram for the *SafeHome* security function



- 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.
- In general, 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 behavior of the system.
- In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- Continuing the discussion of the SafeHome home security function, you might define the following archetypes:

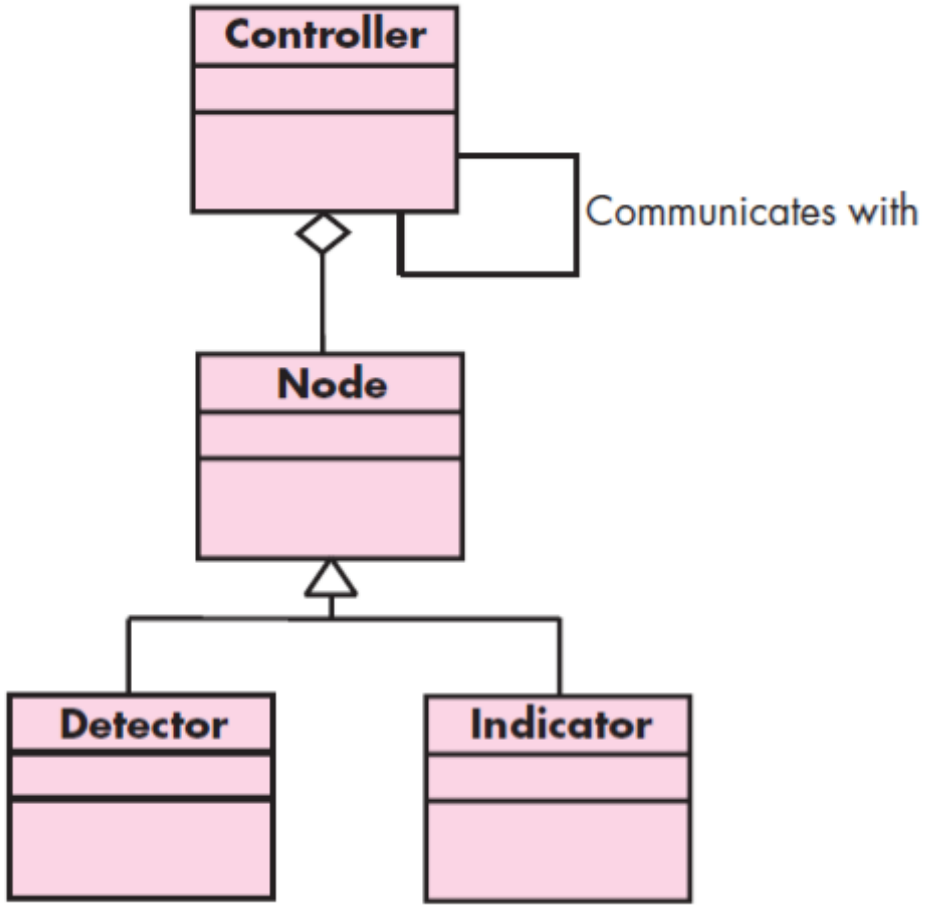
- Node: Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.

- 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.
- Each of these archetypes is depicted using UML notation as shown in below Figure.

UML relation-ships for *SafeHome* security function archetypes



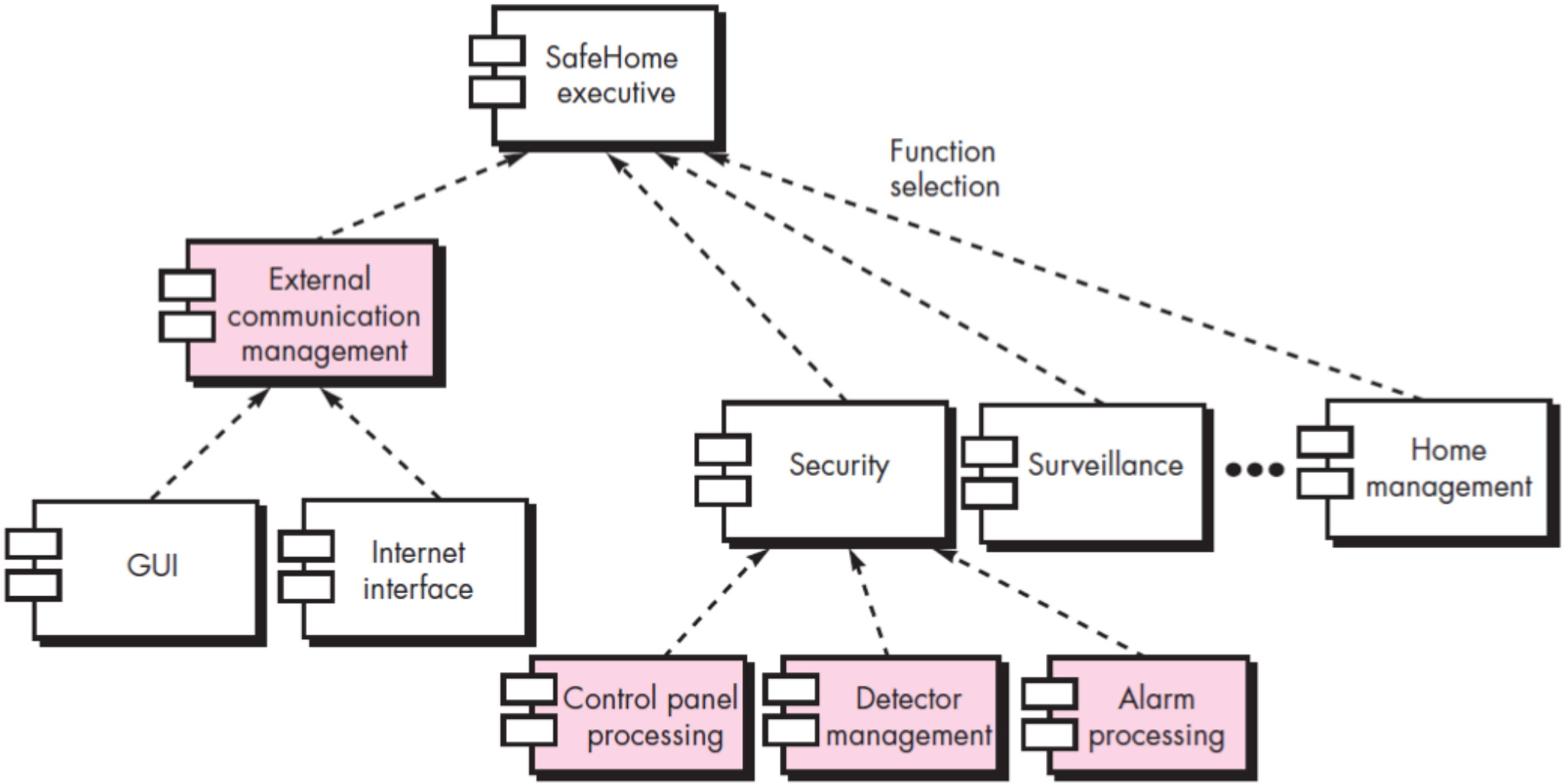
- Refining the Architecture into Components:

- As the software architecture is refined into components, the structure of the system begins to emerge.
- But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model.
- These 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, database components, and task management components are often integrated into the software architecture.

- Continuing the SafeHome home security function example, you might define the set of top-level components that address the following functionality:

- External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- Control panel processing—manages all control panel functionality.
- Detector management—coordinates access to all detectors attached to the system.
- Alarm processing—verifies and acts on all alarm conditions.

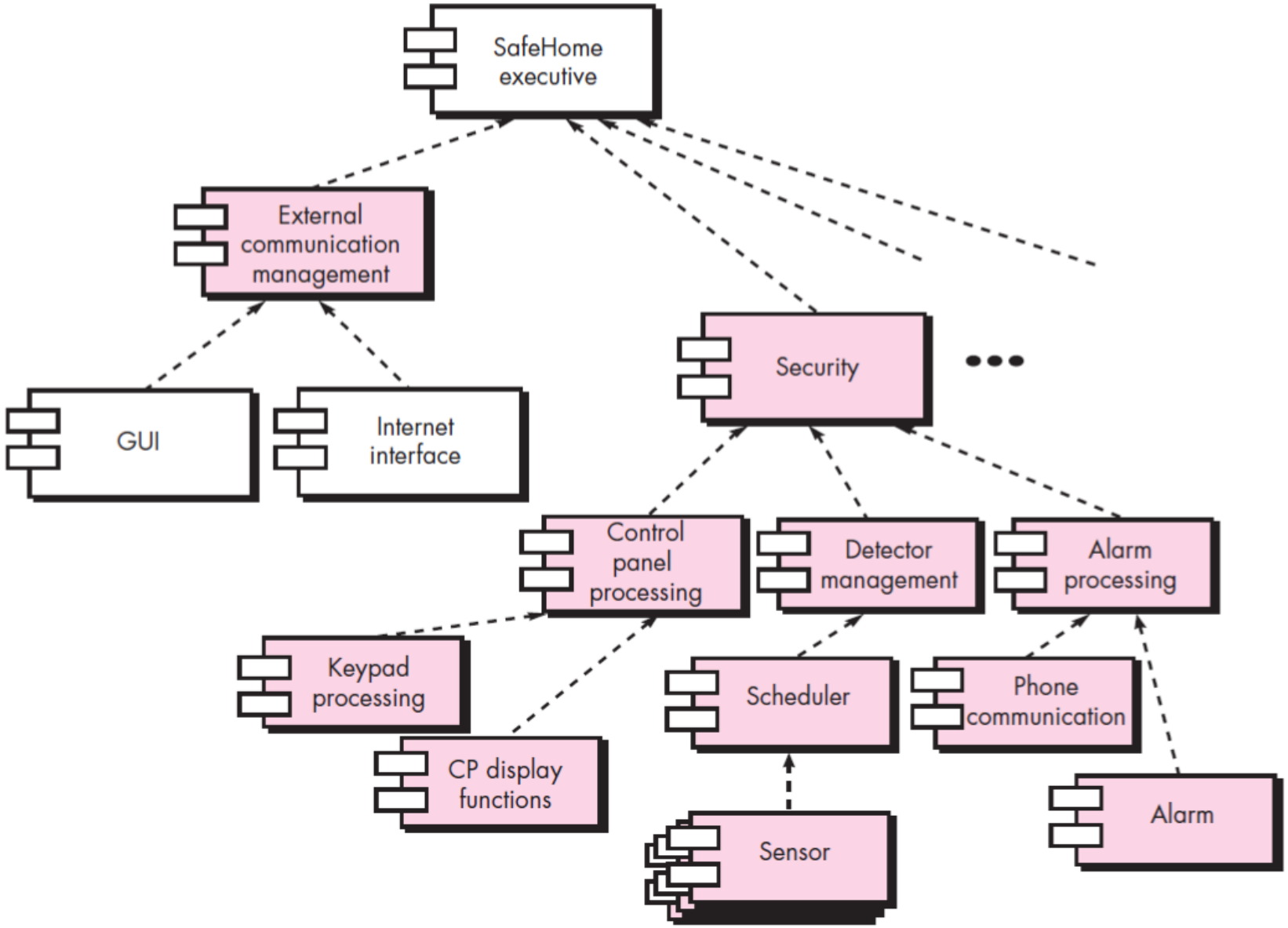
Overall architectural structure for *SafeHome* with top-level components



- Describing Instantiations of the System:

- The architectural design that has been modeled 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.
- However, further refinement (recall that all design is iterative) is still necessary.
- To accomplish this, an actual instantiation of the architecture is developed.
- By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

An instantiation of the security function with component elaboration



Architectural Design

Objective: Architectural design tools model the overall software structure by representing component interface, dependencies and relationships, and interactions.

Mechanics: Tool mechanics vary. In most cases, architectural design capability is part of the functionality provided by automated tools for analysis and design modeling.

Representative Tools:[5]

Adalon, developed by Synthis Corp. (**www.synthis.com**), is a specialized design tool for the design and construction of specific Web-based component architectures.

ObjectiF, developed by microTOOL GmbH (**www.microtool.de/objectiF/en/**), is a UML-based design tool that leads to architectures (e.g., Coldfusion, J2EE, Fusebox) amenable to component-based software engineering (Chapter 29).

Rational Rose, developed by Rational (**www-306.ibm.com/software/rational/**), is a UML-based design tool that supports all aspects of architectural design.

INTERFACE DESIGN

- In his book on interface design, Theo Mandel [Man97] coins three golden rules:

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

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