Unit 3

Design Engineering

Contents:

- Chapter 1: Design Concepts
- Chapter 2: Architectural Design

Design Engineering

- Design is the representation of something **what is to be built.**
  - Ex1**:** **Blue print** for a building architecture.
  - Ex2: A **plan** for conducting workshop – event is executed as per the plan.
  - Ex3: **Course syllabus** – course has to be taught accordingly.
- Design engineering covers the set of **principles, concepts, and best practices** that lead to the development of a high quality system or product.

- **Goal of design engineering** is to produce a model or representation that depict:

  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 product should be **pleasurable** one.

Design Concepts

Design within the context of Software Engineering

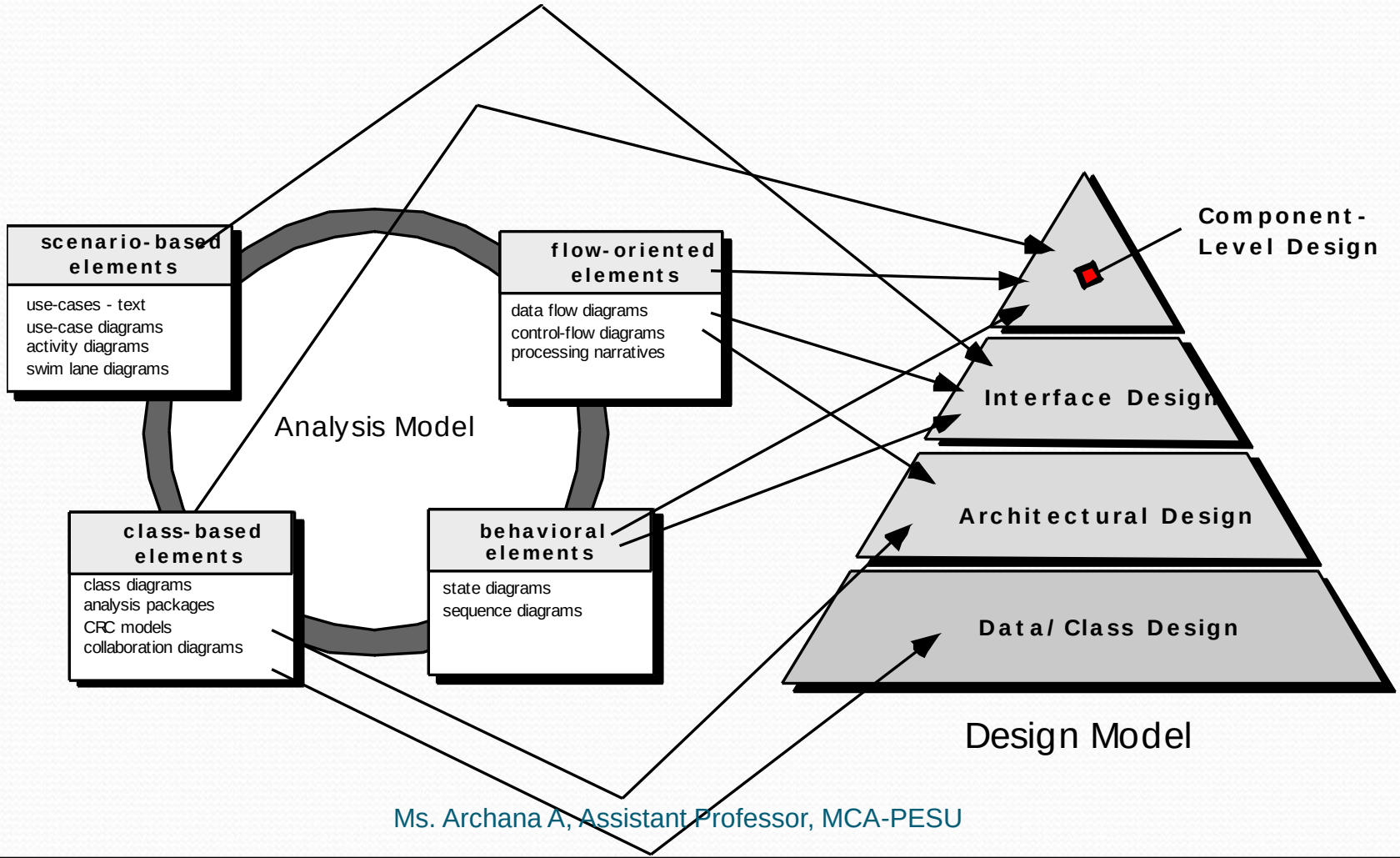
Design model

- The **design model provides** details about software **data structures, architecture, interfaces, and components** that are necessary to implement the system.

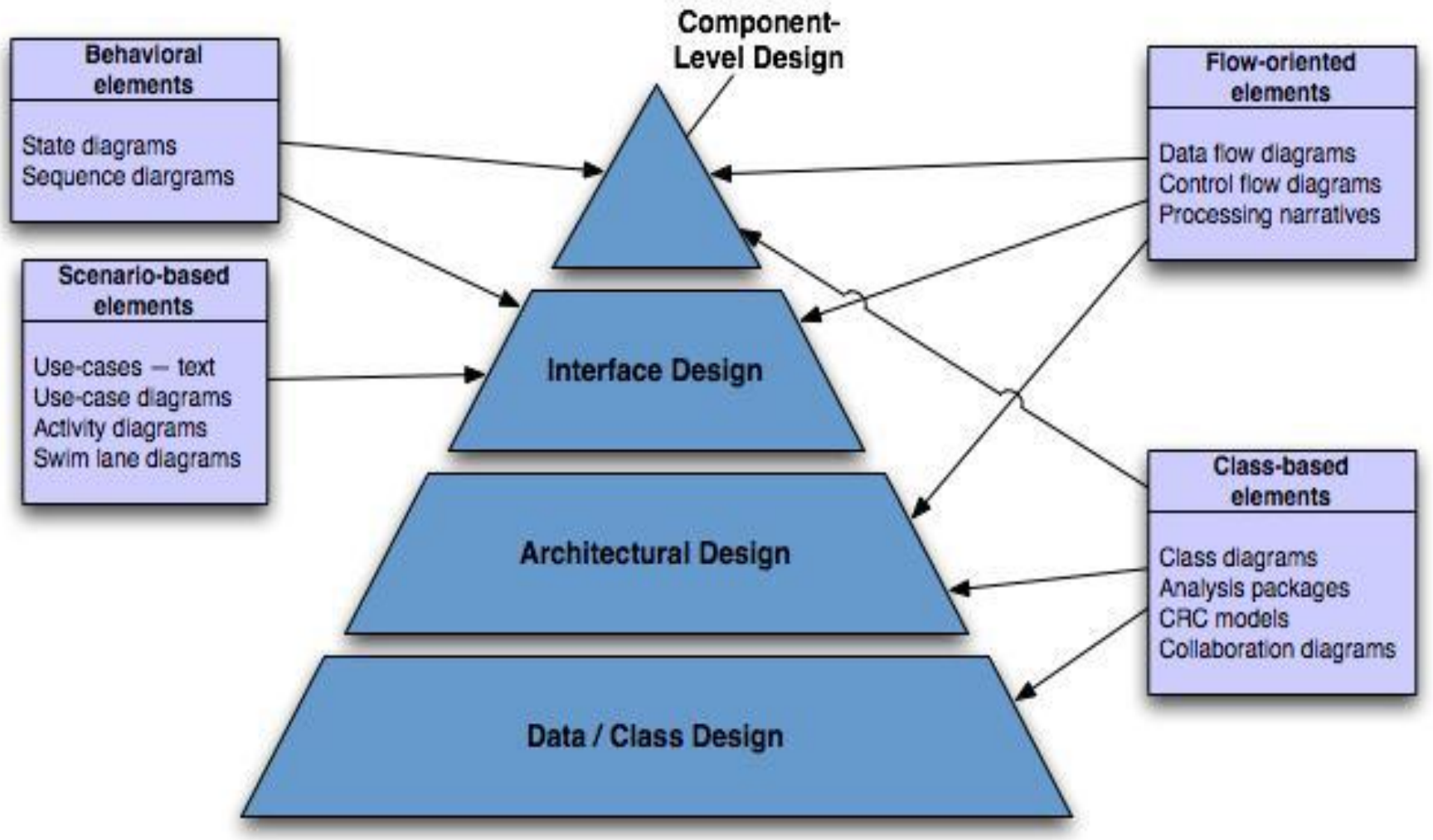
Software Design model…

- Software design model consists of 4 designs:
  - **Data/class Design**
  - **Architectural Design**
  - **Interface Design**
  - **Component Design**

Analysis Model → Design Model



Translating Analysis → Design



Design Model

- **Data/class design** - Created by transforming the analysis model **class-based elements into classes and data structures required** to implement the software.

- **Architectural design** - defines **the relationships among the major structural elements** of the software, it is derived from the **class-based elements and flow-oriented elements** of the analysis model.

Design Model…

- **Interface design** - describes how the software elements, hardware elements, and end-users **communicate** with one another, it is derived from the analysis model **scenario-based elements, flow-oriented elements, and behavioral elements.**

- **Component-level design** - created by transforming the structural elements defined by the software architecture into a procedural description of the **software components** using information obtained from the analysis model **class-based elements, flow-oriented elements, and behavioral elements.**

The Design Process

- Software design is an **iterative process** through which **requirements** are **translated** into a "**blueprint**" for constructing the software.

- As design iteration occur, subsequent **refinement** leads to design representation at much **lower levels of abstraction.**

Goal of Design process

- 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.

Design Quality Attributes

- Hewlett-Packard (**HP**) **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.**

Design Quality Attributes - FURPS

- **Functionality** – is assessed by evaluating the **feature set and capabilities** of the program.
  - Functions that are **delivered and security** of the overall system.

- **Usability** – assessed by considering **human factors, consistency & documentation.**

- **Reliability** – evaluated by
  - measuring the **frequency and severity of failure**.
  - **Accuracy** of output results.
  - Ability to **recover from failure** and **predictability** of the program.

- **Performance** - measured by processing **speed, response time, resource consumption, efficiency**.

- **Supportability** – combines the ability to extend the program (**extensibility), adaptability and serviceability**

Design concepts

Design concepts provide the **necessary framework** for "**to get the thing on right way**".

1. Abstraction.
2. Architecture.
3. Patterns.
4. Separation of concerns.
5. Modularity .
6. Information Hiding.
7. Functional independence.
8. Refinement .
9. Aspects .
10. Refactoring.
11. Object Oriented design concepts .
12. Design Classes.

Fundamental Concepts

- **Abstraction**—data, procedure.
- **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.
- **Information Hiding**—controlled interfaces.

Fundamental Concepts

- **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.
- **Object Oriented design concepts.**
- **Design Classes** —provide design detail that will enable analysis classes to be implemented.

1.Abstraction

- For modular solution there are many level of abstraction
- At the **highest level** of abstraction – a solution is stated in broad terms
- At **lower level** of abstraction – a more detailed description of the solution is provided.

Two types of abstraction:

- ***Procedural Abstraction:*** Sequence of instructions that have a specific and limited function.

  Ex. ***Open*** a door

  open implies long sequence of activities i.e.,walk to the door, grasp knob, turn knob and pull the door, etc.

1.Abstraction

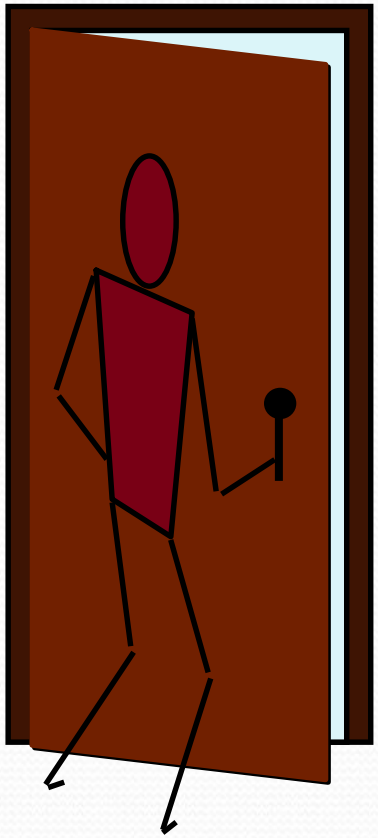
- ***Data Abstraction***: Collection of data that describes a data object.

Ex. Open a door. – ***door*** is data object.

- Data abstraction for ***door*** would encompass a set of attributes that describe the door.

Ex: door type, swing direction, opening mechanism, etc.

Procedural Abstraction

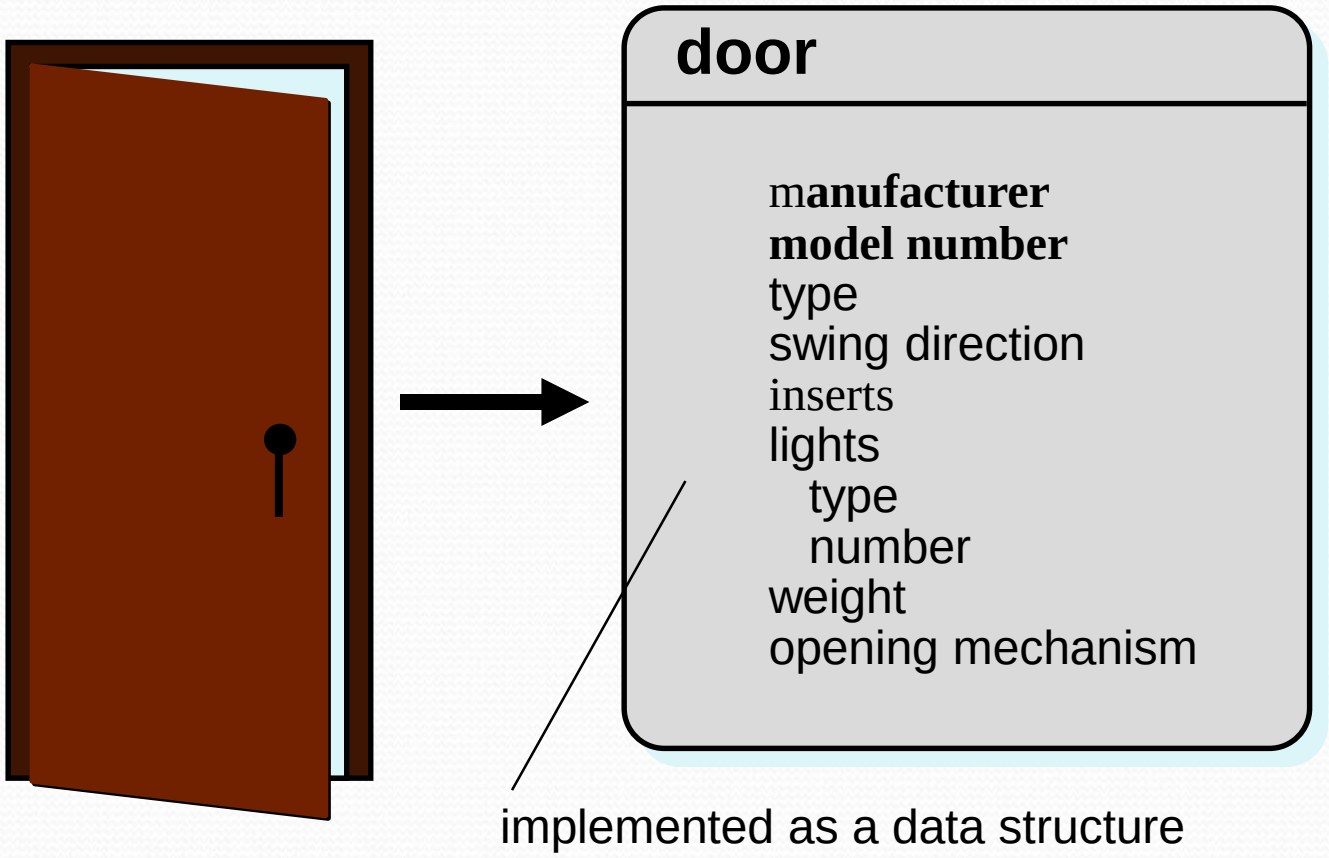


open

details of enter algorithm

implemented with a "knowledge" of the object that is associated with enter

Data Abstraction



2.Architecture

- Software architecture suggest **“ the overall structure of the software** and the ways in which that structure provides **conceptual integrity for a system.”**

- There are **different models,** which can be used to represent architecture.

1. **Structural Model-** represent architecture as an ***organized collection of components*.**
2. **Framework model** – Increase level of design abstraction by identifying ***repeatable architectural design framework.***

2.Architecture…

3. **Dynamic model** – addresses the behavior aspects of the program architecture and indicating *how structure or system configuration may change as a function of* ***external event.***

4. **Process Model** – focus on **design of the business** or **technical process** that the system must accommodate.

5. **Functional models** – used to represent the *functional hierarchy* of a system.

3. Design Pattern

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

- The **intent of each design pattern** is to provide a description that enables a designer to determine

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

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

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

4. 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.

4.Separation of Concerns…

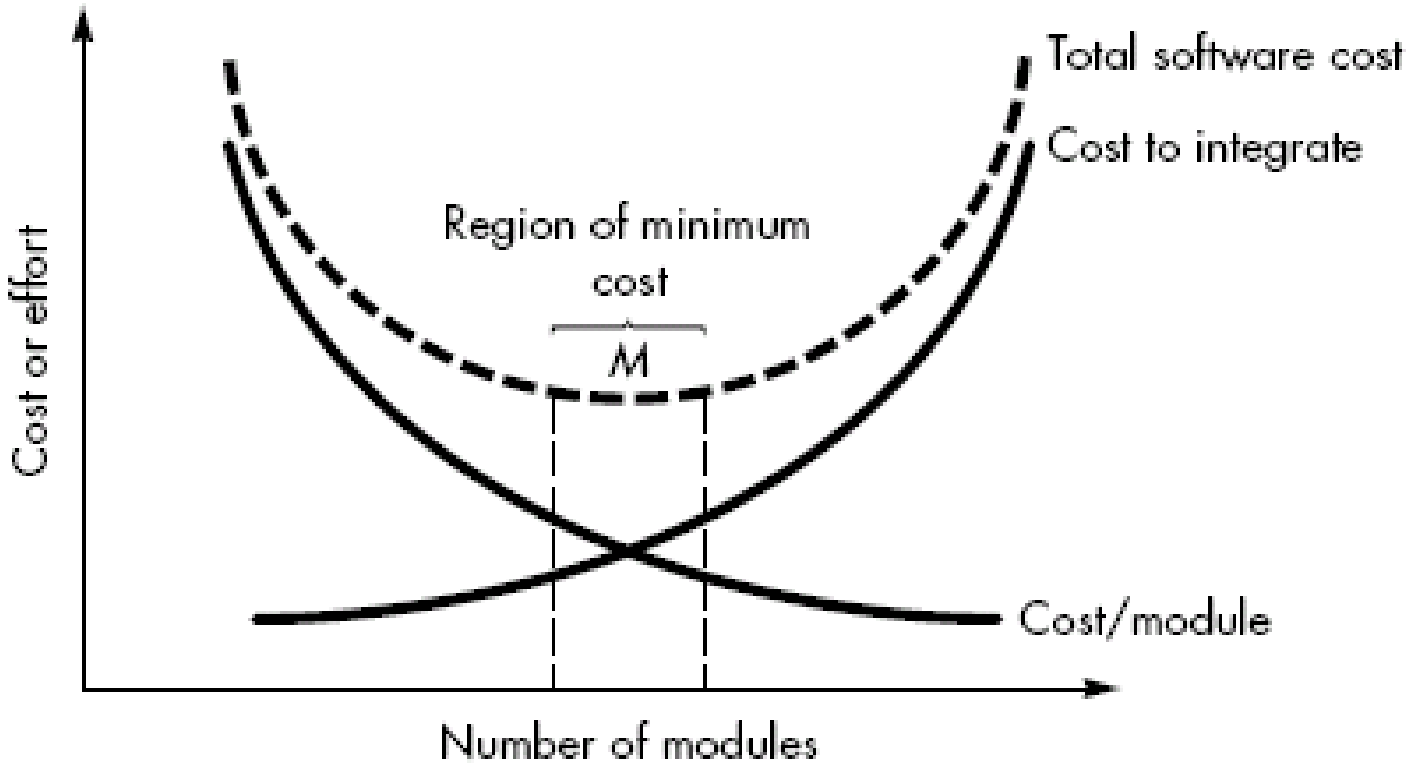
- This leads to **divide-and-conquer** strategy—it's easier to solve a **complex problem** when you **break it into manageable pieces.** This has important implications with regard to software modularity.

- Separation of concerns is manifested in other related design concepts: **modularity, aspects, functional independence, and refinement**.

5. Modularity

- Software is divided into separately named and **addressable components,** sometimes called **modules,** which are **integrated** to satisfy problem requirement.
- modularity is a single attribute of software that allows a program to be **intellectually manageable**.
- Refer fig. that state that effort (cost) to develop an individual software module does decrease if total number of modules increase.
- However as the no. of modules grows, the effort (cost) associated with integrating the modules also grows.

Modularity and software cost



5. Modularity…

- **Undermodularity and overmodularity should be avoided.**
  - We modularize a design so that development can be more **easily planned**.
  - **Software increments** can be defined and delivered.
  - **Changes** can be more easily accommodated.
  - **Testing and debugging** can be conducted more **efficiently** and long-term maintained can be conducted without serious side effects.

6. Information Hiding

- The principle of *information hiding* suggests that "**modules be characterized by design decisions that (each) hides from all others**."

- **i.e., Modules** should be **specified** and **designed** so that information (algorithm and data) contained within a **module is inaccessible** to other modules that have no need for such information.

Information Hiding…

- The intent of information hiding is to **hide the details of data structure and procedural** processing behind a module interface. Knowledge of the details need not be known by the users of module
- It gives benefits when **modifications are required during testing and maintenance,** because data and procedure are hiding from other parts of software, **unintentional errors** introduced during modification, **are less**.

7. Functional Independence

- The concept of functional independence is a direct outgrowth of **separation of concerns, modularity, and the concepts of abstraction and information hiding.**
- Functional independence is achieved by developing modules with "**single-minded**" function and an "**aversion**“ (dislike) to excessive interaction with other modules.

why independence is important?

- Independent modules are **easier to maintain** (and test) because
  - **code modification are limited**,
  - **error propagation is reduced, and**
  - **reusable modules** are possible.
- **Functional independence** is a **key** to **good design**, and good design is the key to software quality.
- Independence is assessed using **two qualitative criteria:**
  - ***Cohesion*** **and** ***Coupling.***

- ***Cohesion*** is an indication of the relative **functional strength of a module.**(functional relatedness of elements within 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.** (High Cohesion)

Functional independence…

- ***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.(Low coupling)

8. Refinement

- 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.**

Refinement…

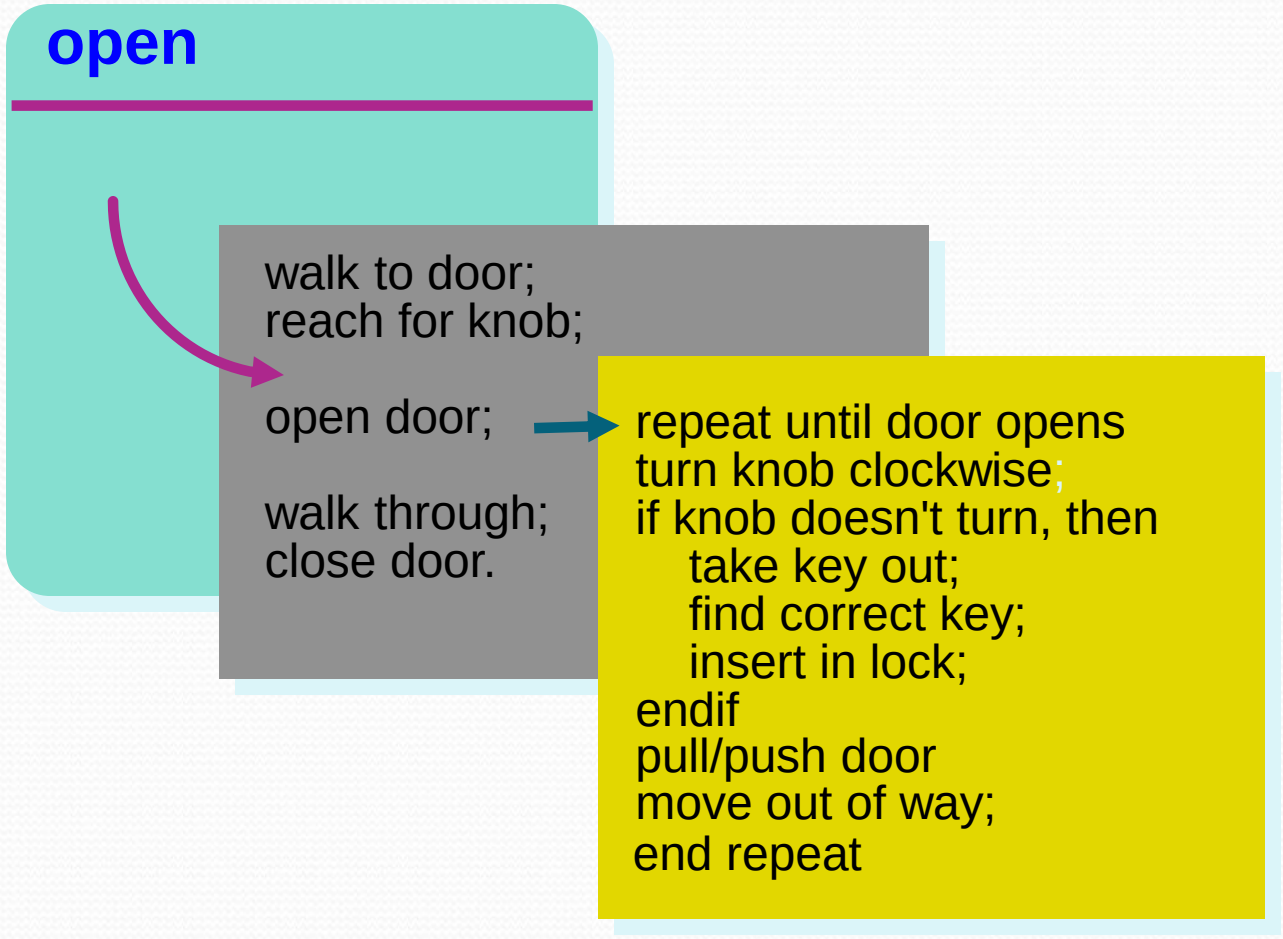
- That is, the **statement describes** function or information **conceptually** but provides **no information about the internal workings** of the function or the internal structure of the information.

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

8. Refinement…

- Abstraction and Refinement are **complementary concepts.**
- **Abstraction** enables a designer to specify procedure and data and yet **suppress low-level details.**
- **Refinement** helps the designer to **expose low-level details** as design progresses.

Stepwise Refinement



9. Aspects

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

- 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".

10. Refactoring

- Refactoring is a **reorganization technique** that simplifies the design (or code) of a component **without changing its function** or **behavior.**

- Refactoring improves its internal structure.

Refactoring…

- When software is refactored, the existing design is examined for :
  - redundancy
  - unused design elements
  - **inefficient** or unnecessary algorithms
  - **poorly constructed** or inappropriate data structures
  - any other design failure that can be corrected to yield a **better design**

11. Object Oriented Design Concepts

The object-oriented (OO) paradigm is **widely used** in modern software engineering.

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

12. Design Classes

- The **requirements model** defines a set of **analysis classes**. Each describes some element of the **problem domain, focusing on aspects** of the problem that are user visible.

- As the **design model evolve,** we will define a set of ***design classes, that refine the*** **analysis classes** by providing design detail that will enable the classes to be implemented,

12. Design Classes…

Five different types of design classes:

1. ***User interface classes*** *define all abstractions that are necessary for Human Computer Interaction* (HCI).

2. ***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

12. Design Classes…

3. ***Process classes*** *implement lower-level business abstractions required to fully* manage the business domain classes.
4. ***Persistent classes*** *represent data stores (e.g., a database) that will* *persist* beyond the execution of the software.
5. ***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.

12. Design Classes…

- Each design class is reviewed to ensure that it is "**well-formed**."
- There are Four characteristics of a well-formed design class:
- **Complete and sufficient**
- **Primitiveness.**
- **High cohesion.**
- **Low coupling**

Architectural Design

Contents

- Software Architecture - Definition
- Architectural Genres
- Architectural Styles
- Architectural Design

Software Architecture -Definition

- The software architecture of a program or computing system is the **structure** or **structures of the system** which comprise
  - The software components,
  - The externally visible properties of those components,
  - The relationships among the components.

- **Software architectural design** represents the **structure of the data** and **program components** that are required **to build a computer-based system**.

- An architectural design model is **transferable**
  - It can be **applied** to the design of other systems

Architectural Genres

Architectural Genres

- ***Genre*** implies a **specific category** within the overall **software domain.**
- Within each category, it 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.

Architectural Styles

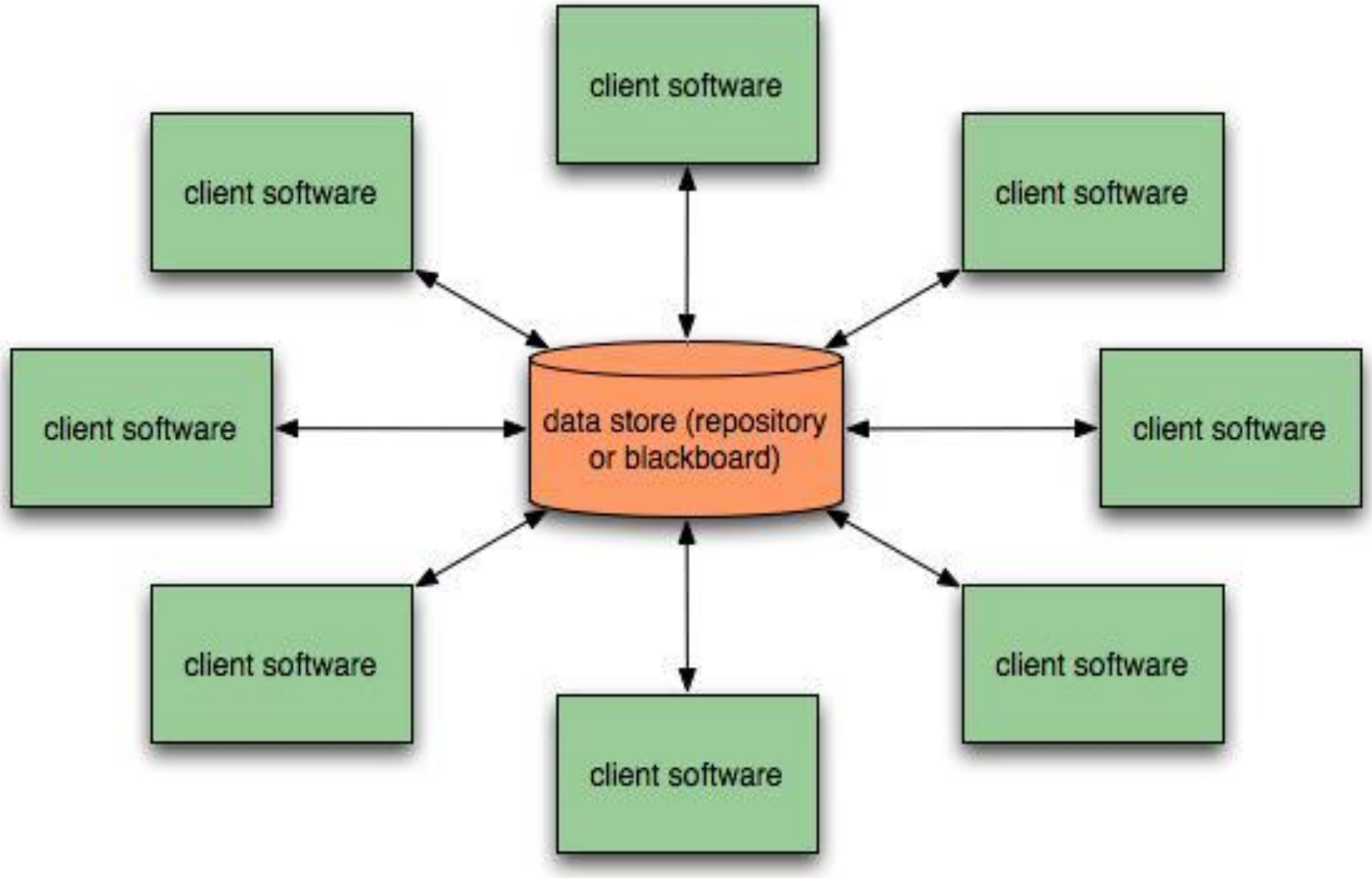
Architectural Styles

- Architectural 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.

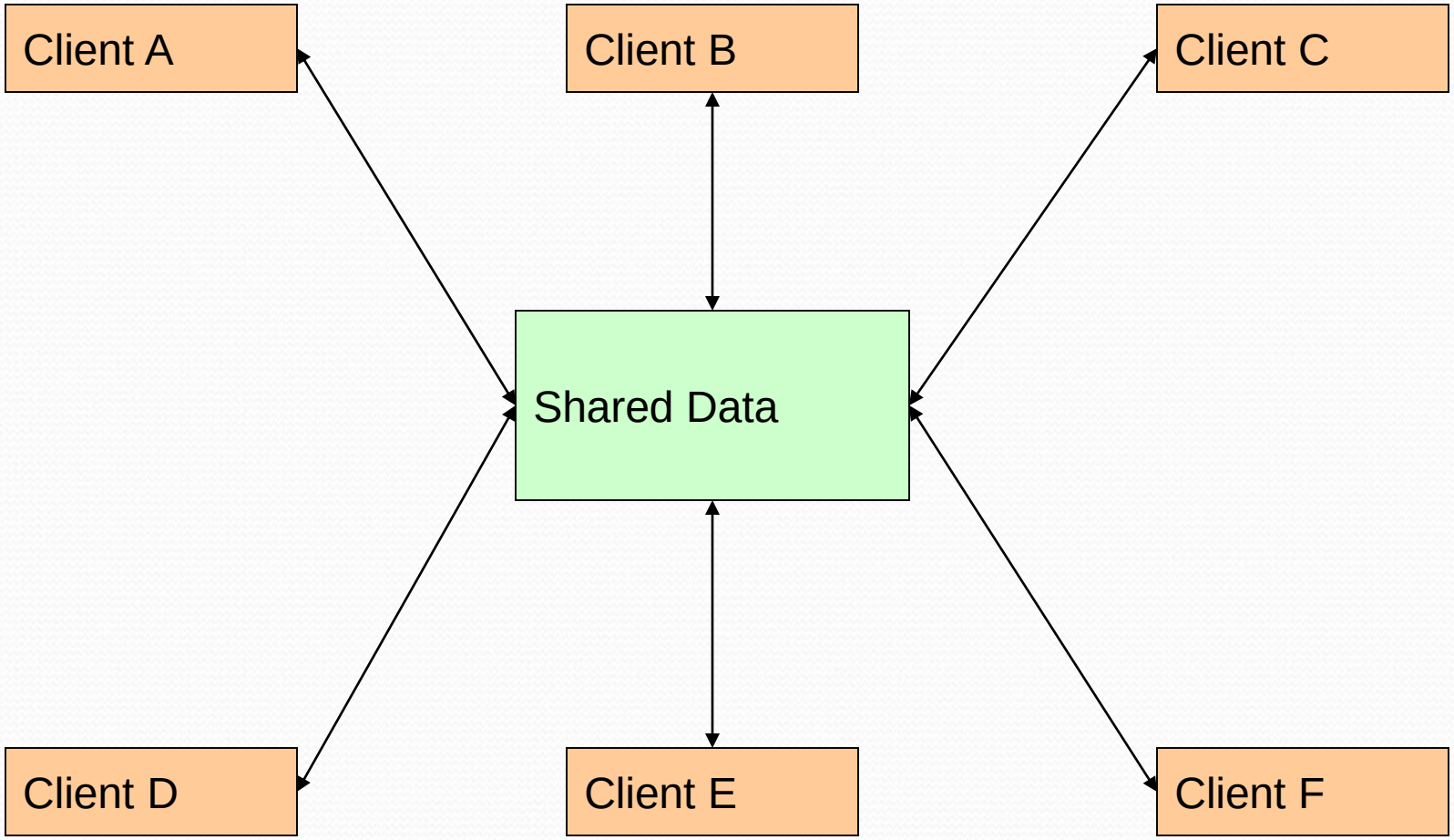
Architectural Styles…

1. **Data-centered architectures**
2. **Data flow architectures**
3. **Call and return architectures**
4. **Object-oriented architectures**
5. **Layered architectures**

1. Data-Centered Architecture



Data-Centered Style…



Data-Centered Style…

- Has the goal of integrating the data
- 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 modify** data within the store.(shown in fig.)
- Client software accesses a central repository.

Data-Centered Style…

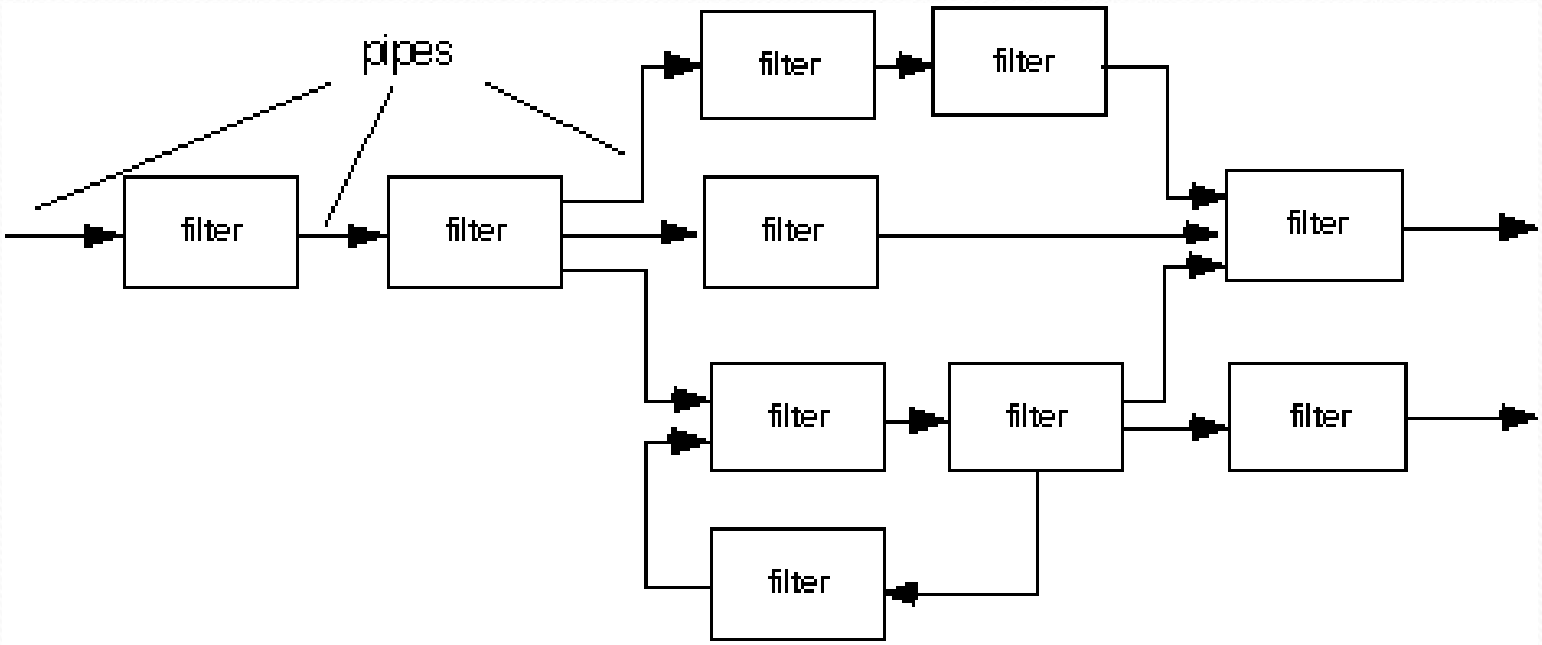
- In some cases data repository is passive. i.e., client software accesses the data independent of any changes to the data.

- 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.

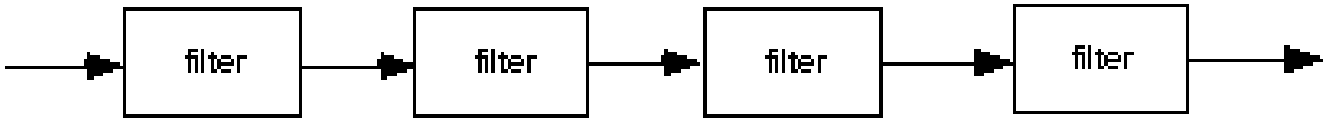
Data-Centered Style…

- Data-centered architectures promote ***integrability***
  - i.e., existing components can be changed and new client components added to the architecture without concern about other clients (since the client component operate independently).
- In addition, data can be passed among clients using **blackboard mechanism**.
- Client components **independently** execute processes

2. Data Flow Architecture



(a) pipes and filters



(b) batch sequential

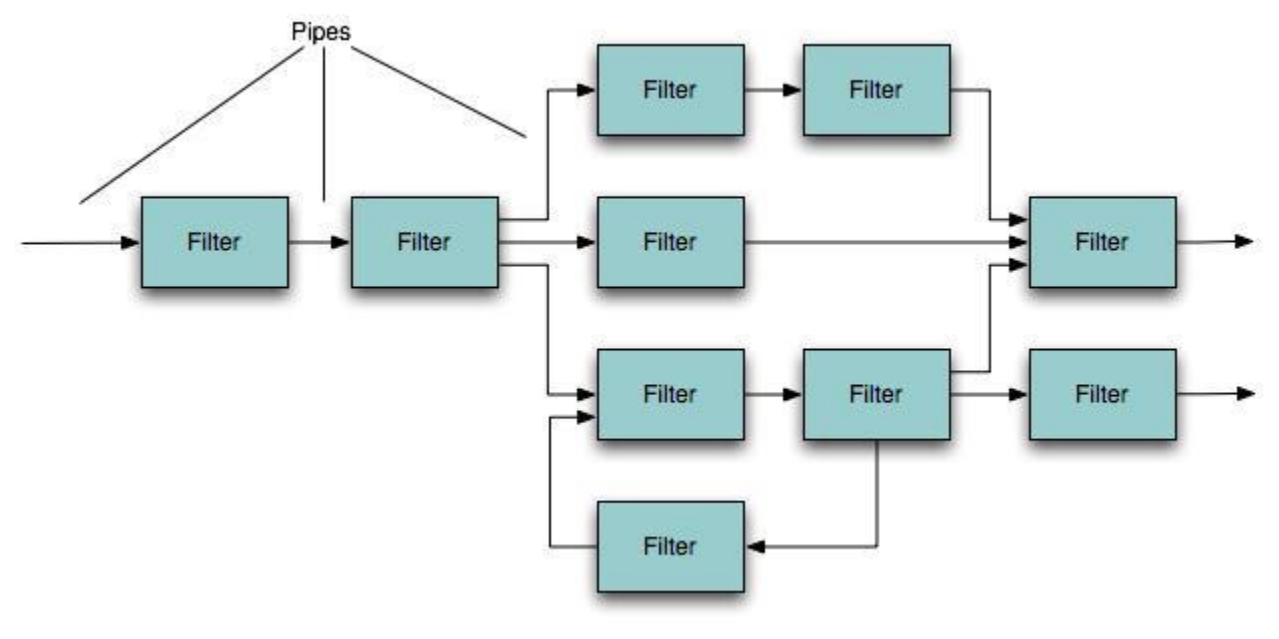
2. Data Flow Style

- Has the goal of modifiability
- Characterized by **viewing the system** as a series of transformations on successive pieces of input data.
- **Data enters the system and then flows through the components one at a time until they are assigned to output or a data store.**

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output.

- **Pipe-and-filter** style
- This style, Pipe-and-filter 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 & downstream, is designed to expect data input of a certain form and produces data output (to the next filter) of a specified form.
- The pipes are stateless and simply exist to move data between filters

Pipes-and-filter style



Data Flow Style…

- **Batch sequential style**
  - If the data flow degenerates into a single line of transforms, it is termed **batch sequential**
  - This structure accepts a **batch of data** & then applies a series of sequential components (filters) to transform it.
  - **Each step runs to completion before the next step begins**

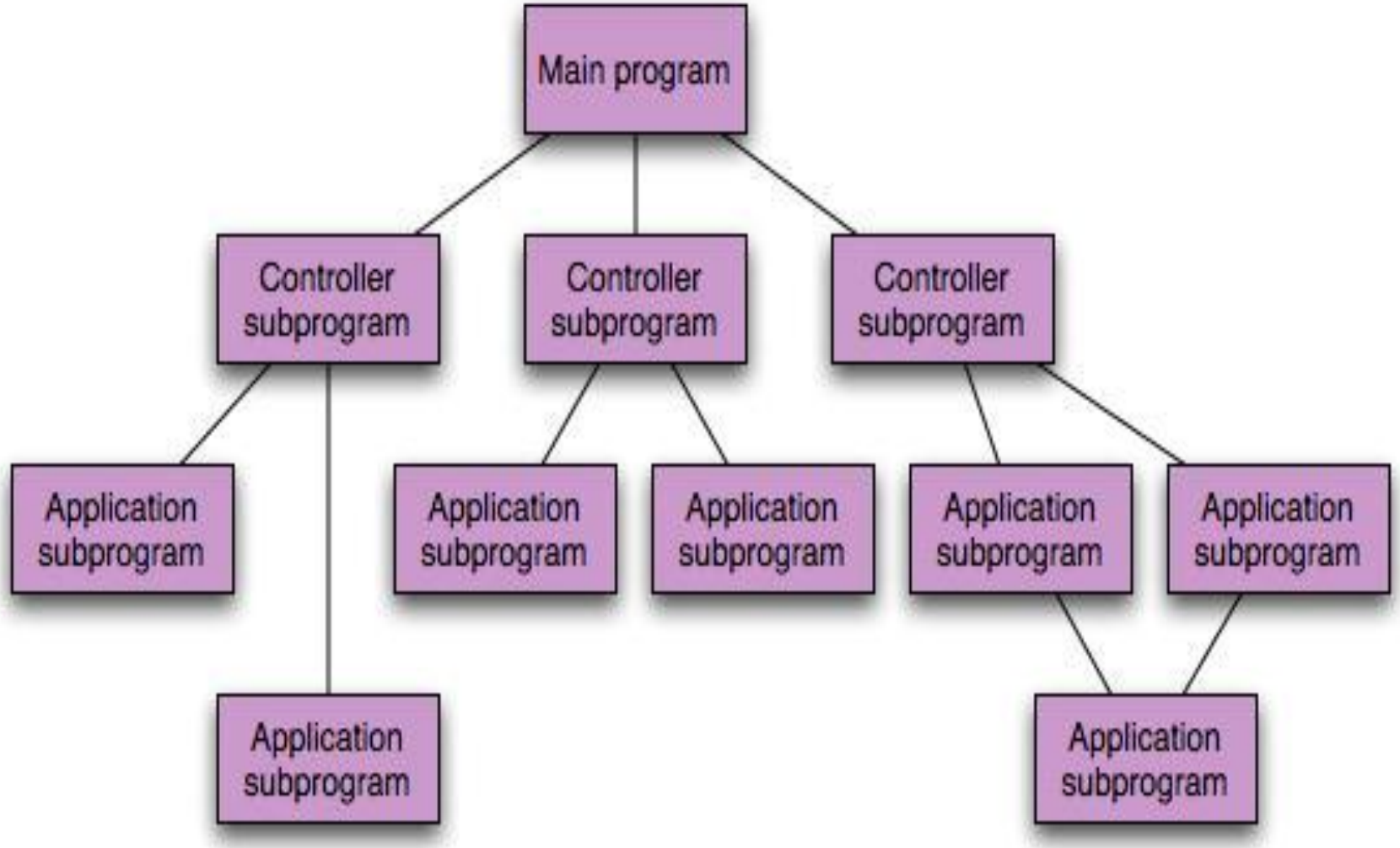
(b) batch sequential

3. Call and return architecture

- This architectural style enables to achieve a program structure that is relatively easy to **modify and scale.**

- The substyles exist within this category are:
- ***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.
- ***Remote procedure call architectures*** – the components of a main program / subprogram architecture are distributed across multiple computers on a network

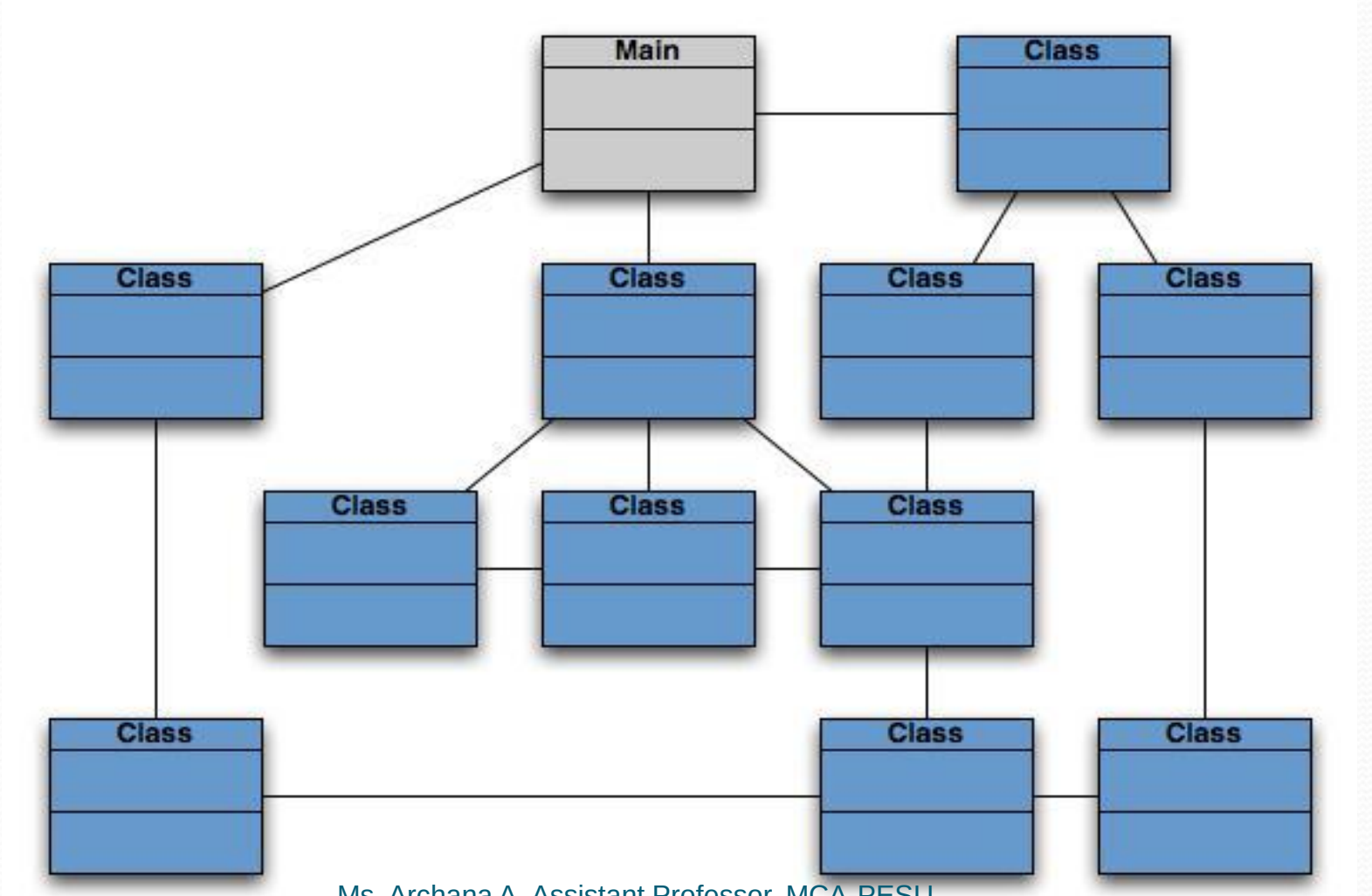
3. Call and Return Architecture



4. Object Oriented Architecture

- 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.**

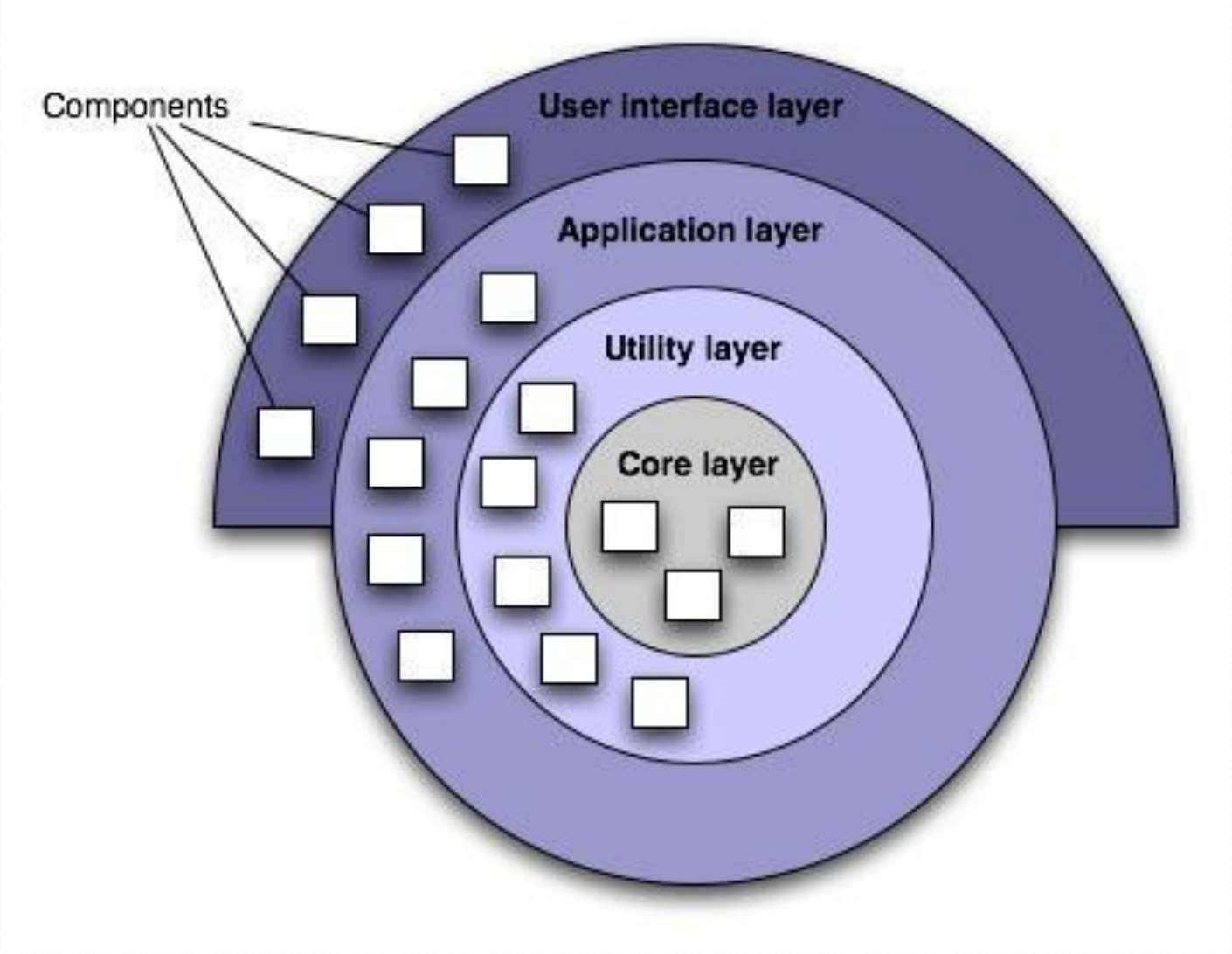
4. Object Oriented Architecture



5. Layered architecture

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
  - At the outer layer, **components perform service user interface operations.**
  - At the inner layer, **components perform operating system interfacing.**
  - Intermediate layers provide **utility services and application software functions**.

5. Layered Architecture



Architectural Styles…

- These architectural styles are only a small subset of those available.
- Once requirements engineering uncovers the 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 can be combined with a data-centered architecture in many database applications.

Architectural Design

Architectural Design

- Architectural design is accomplished using four distinct steps.

1. **The system must be represented in context**. i.e., the designer should **define the external entities** that the software interact with & the nature of interaction.

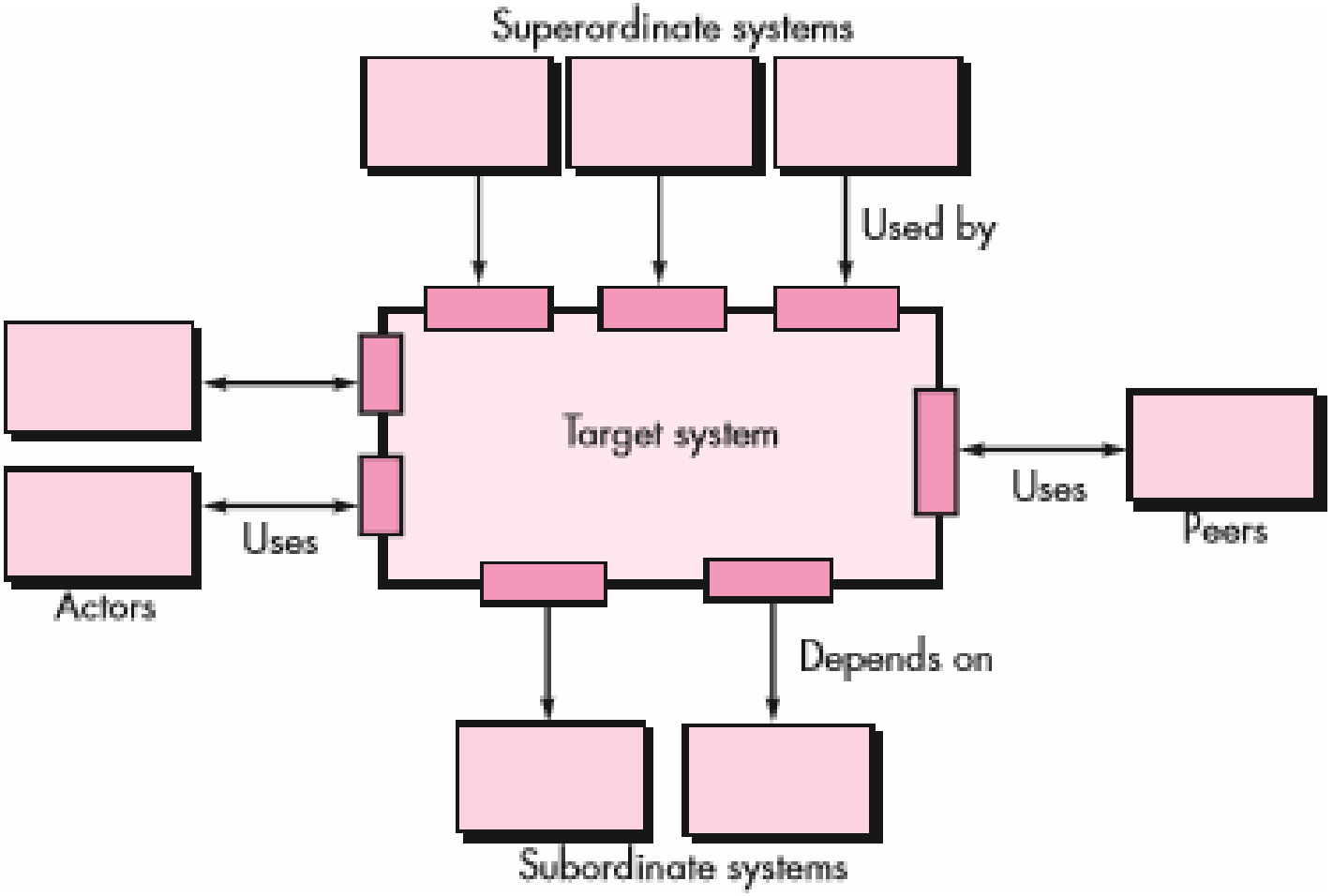
2. Once context specified, the designer should **identify a set of top-level abstractions, called archetypes**, that represent pivotal elements of the system's behavior or function.

Architectural design…

3. After abstractions have been defined, the design begins to move closer to the **implementation domain**. Components are identified & represented with the context of an architecture that supports them.

4. Finally**, specific instantiations of the architecture are developed** to "prove" the design in a real-world context.

1.Representing the system in a context



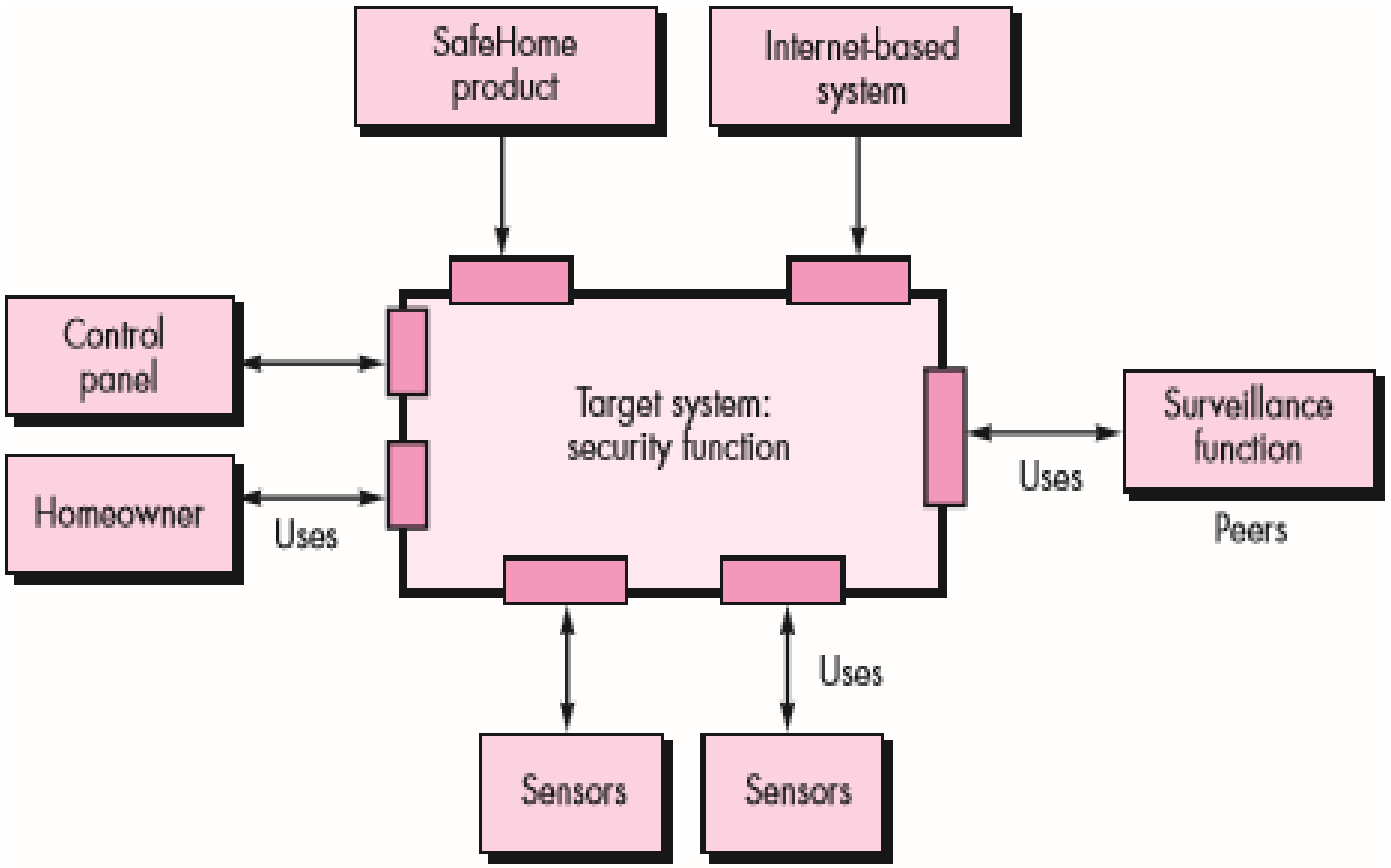
- Referring to fig, **systems that interoperate with the target system** (for which an architectural design is to be developed) are represented as:
- ***Superordinate systems*** – those systems that use the target system as **part of** some higher-level processing scheme.
- ***Subordinate systems*** – 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 system 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 communicate with the **target system** through an **interface** (the small shaded rectangle)

Architectural Context

FIGURE 9.6

Architectural context diagram for the *SafeHome* security function



2. Defining Archetypes

- Archetypes are the **abstract building blocks of an architectural design.**
  - These archetypes are **class or pattern** that represents stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.
- The **SafeHome home security function** might define the following archetypes.

1. Node – represents a ***cohesive*** collection of **input** and **output** elements of the home security function.

- For eg: a node might be comprised of various ***sensors*** & a variety of ***alarm***(output) **indicators**.

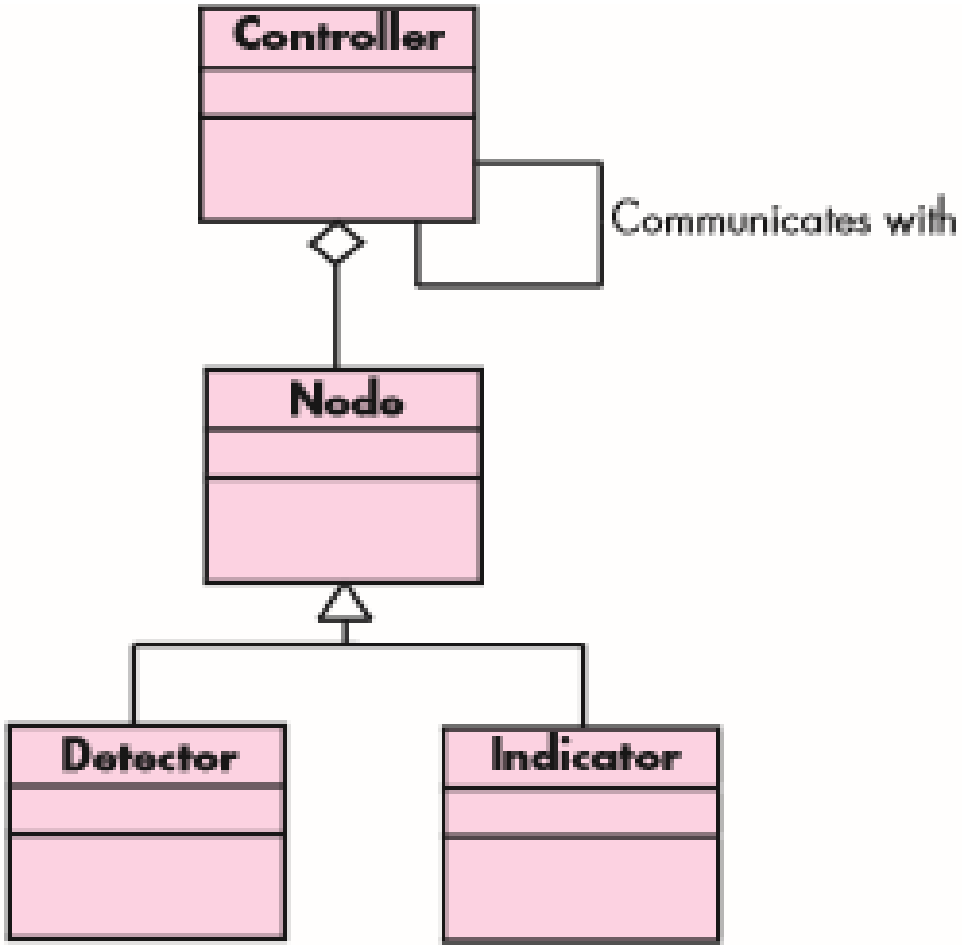
2. Detector – an abstraction that encompasses all ***sensing equipment*** that feeds information into the target system.

Defining Archetypes…

3. **Indicator** – an abstraction that represents all **mechanisms** (eg: alarm siren, flashing lights, bell) for indicating that an ***alarm condition*** is occurring.
4. **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 fig.

UML relationships for SafeHome security function archetypes



3. 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?

- **Ans**: begin with the classes that were described as part of **requirements model**. Classes represents entities within the application domain that must be addressed within software architecture

Refining the architecture into components…

- Hence **application domain** is one source for the **derivation** & **refinement** of components.
- Another source is **infrastructure domain** – for eg: memory management components, communication components, database components, task management component are often integrated into software architecture.
- For SafeHome security function example, we might define the set of top-level components that address the following functionality:
  - ***External communication management*** – coordinates communication of the **security function with external entities** such as other internet based systems and external alarm notifications.

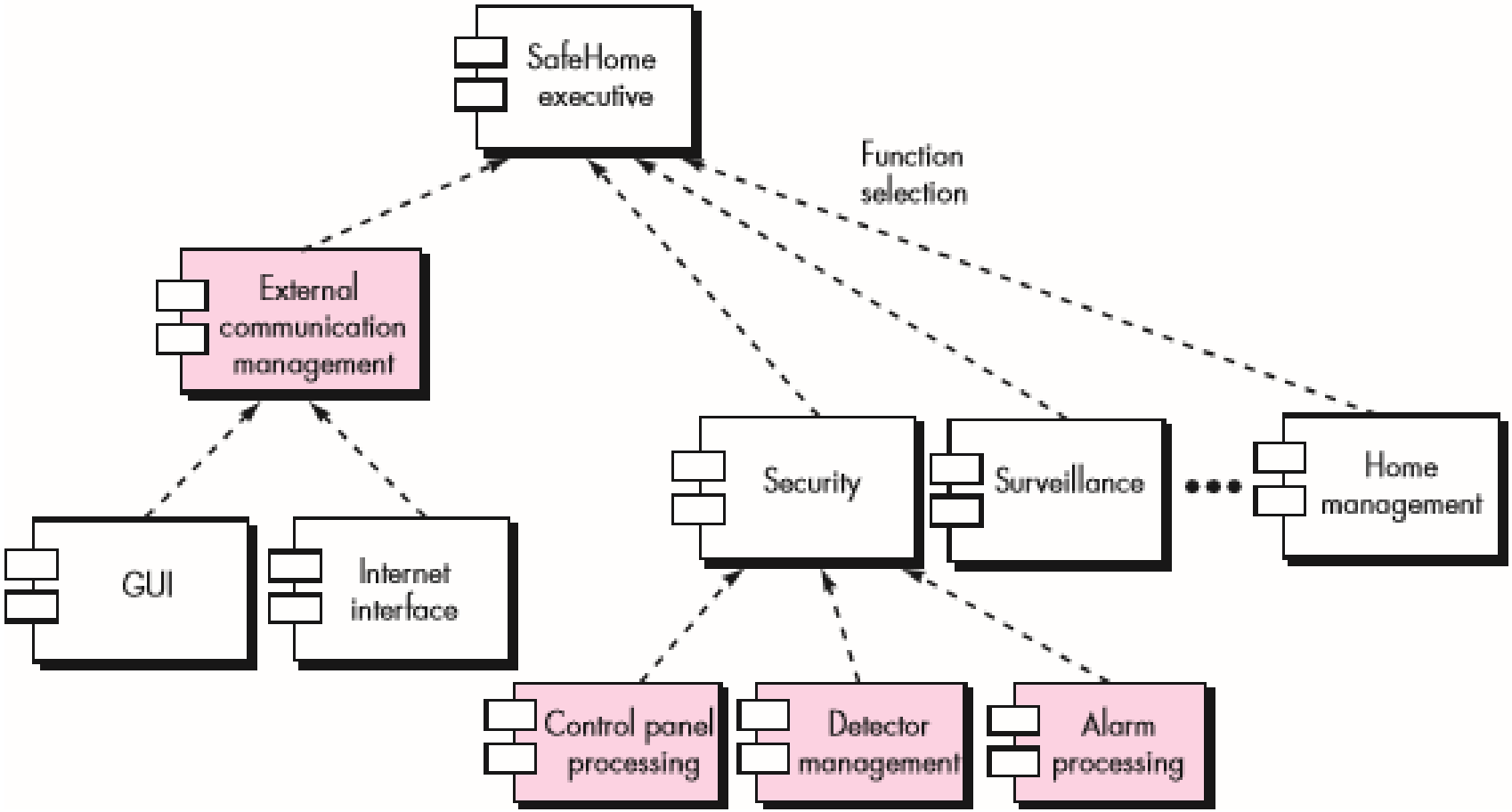
Refining the architecture into components…

- ***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.

- Each of these top-level components would have to be elaborated iteratively & then positioned within the overall SafeHome architecture.

Component Structure

FIGURE 9.8 Overall architectural structure for *SafeHome* with top-level components

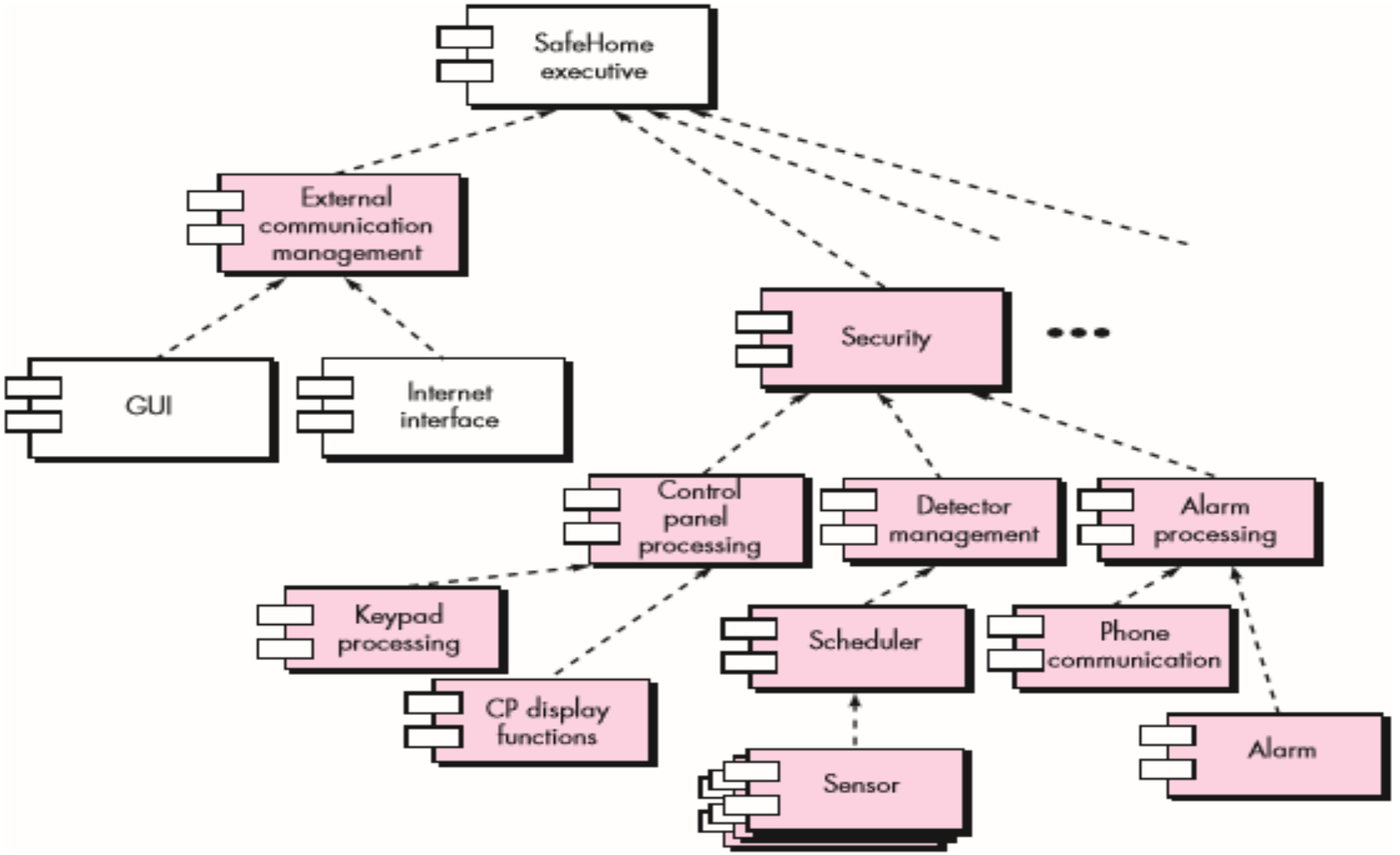


4. Describing instantiations of the system.

- The architectural design that has been modeled to this point is still relatively high level.
- However , **further refinement is still necessary.**
- To accomplish this, an actual instantiation of the architecture is developed. The architecture is applied to a specific problem with the intent of demonstrating that the structure & components are appropriate.
- Fig 9.9 illustrates an instantiation of the SafeHome architecture for the security system.

Refined Component Structure

FIGURE 9.9 An instantiation of the security function with component elaboration



4. Describing instantiations of the system…

- the components shown in fig.9.8 are elaborated to show the additional detail.
- For e.g.: the **detector management** component interacts with a **scheduler** infrastructure component that implements **polling of each sensor** object used by the security system.
- Similarly elaboration is performed for each of the components represented in fig.9.8.

END OF UNIT-3