Ch.13 Architectural Design

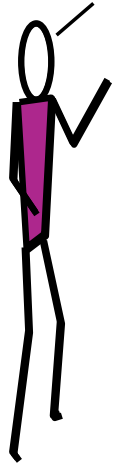
Lecturer: Yusheng Liu (刘玉生)

April 8, 2024

• Why Architecture?

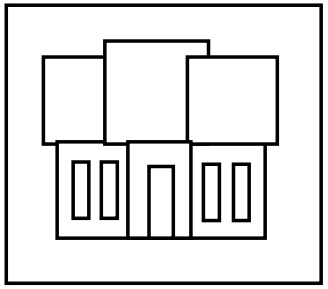
customer requirements

- The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:
  - (1) ***analyze the effectiveness of the design*** in meeting its stated requirements,
  - (2) ***consider architectural alternatives*** at a stage when making design changes is still relatively easy, and
  - (3) ***reduce the risks*** associated with the construction of the software.



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

Ads---2 bed 1.5 bath



architectural design

Architecture is the structure or structures of a program or computing system, which comprise software components, the externally visible properties of those components, and the relationships among them [BAS03].

Why is Architecture Important?

- *Representations of software architecture are an enabler* for communication between all parties (stakeholders) interested in the development of a computer-based system.

- *The architecture highlights early design decisions* that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

- *Architecture "constitutes a relatively small, intellectually graspable mode* of how the system is structured and how its components work together" [BAS03].

Architectural Descriptions

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

Architectural Genres

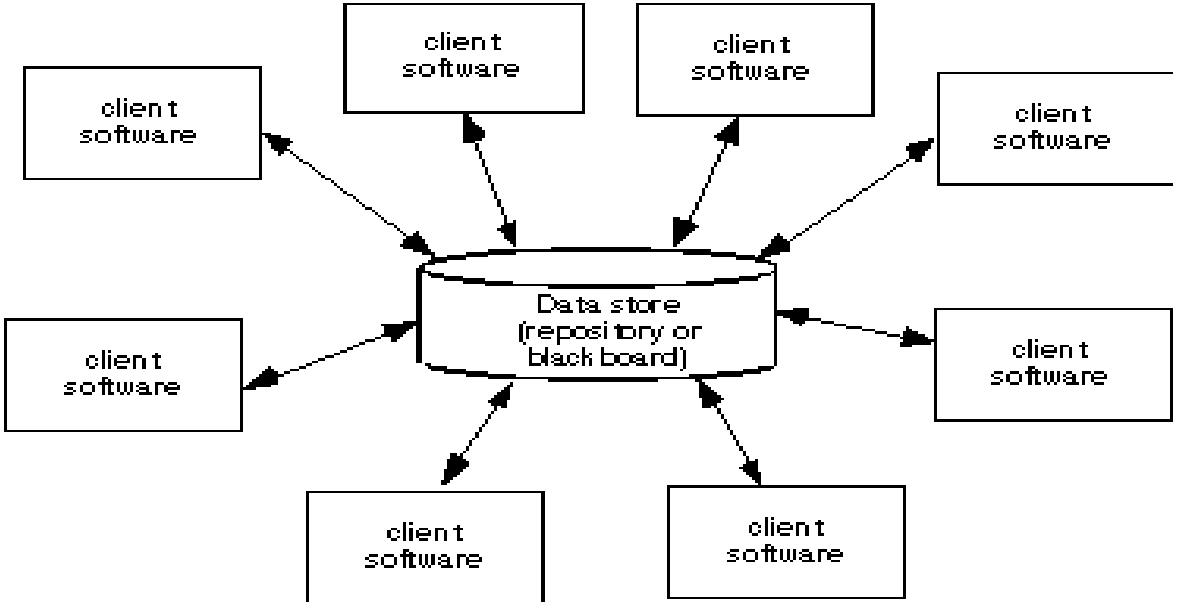
- *Genre(类型,样式)* implies a specific category within the overall software domain.

- Within each category, you encounter a number of subcategories.
  - For example, within the genre of buildings, you would encounter the following general ***styles***: Houses(别墅), condos(有独立产权公寓), Townhouse(联排别墅),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.

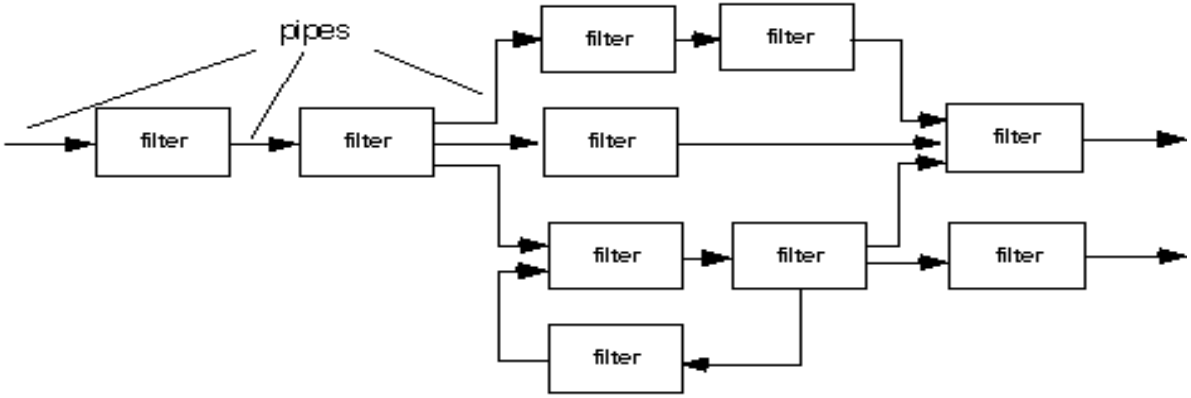
Architectural Styles

- Each style describes a system category that encompasses:
- (1) a *set of components* (e.g., a database, computational modules) that perform a function required by a system,
- (2) a *set of connectors* that enable “communication, coordination and cooperation” among components,
- (3) *constraints* that define how components can be integrated to form the system,
- (4) *semantic models* that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.
  - Data-centered architectures
  - Data flow architectures
  - Call and return architectures
  - Object-oriented architectures (构件封装了数据和操作，构件间通过信息通信合作)
  - Layered architectures

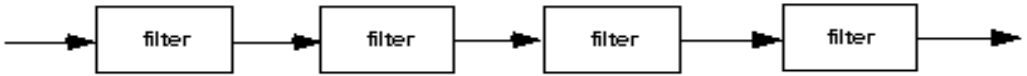
- **Data-Centered Architecture**



- **Data Flow Architecture**

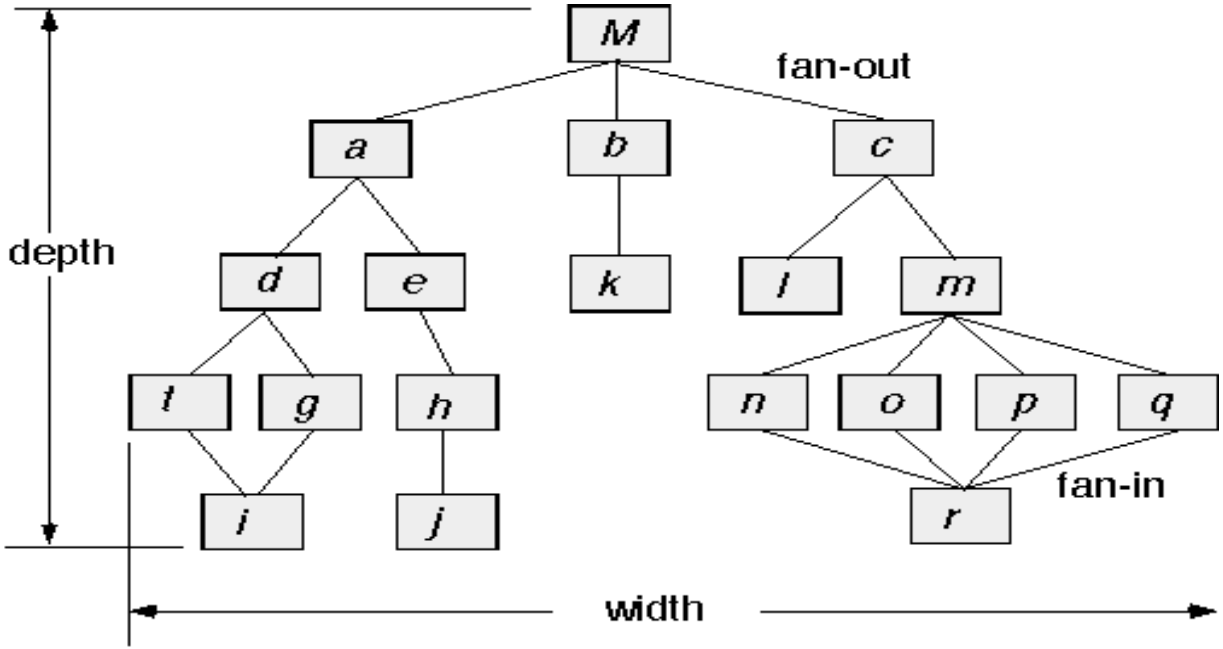


(a) pipes and filters



(b) batch sequential

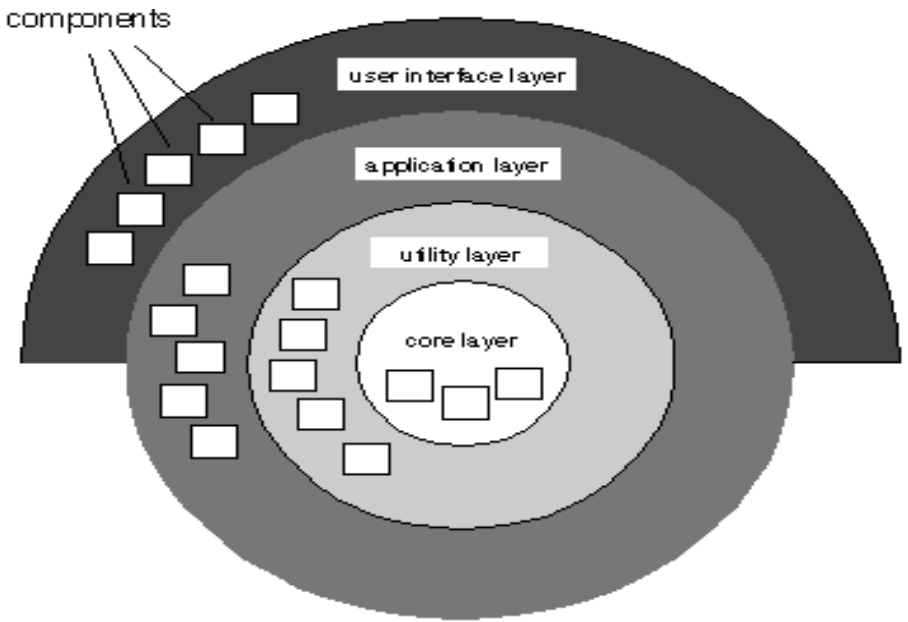
- **Call and Return(调用与召回) Architecture**



- **Layered Architecture**

e.g. C-S (Client-Server) model:

---WebApp, MobileApp）

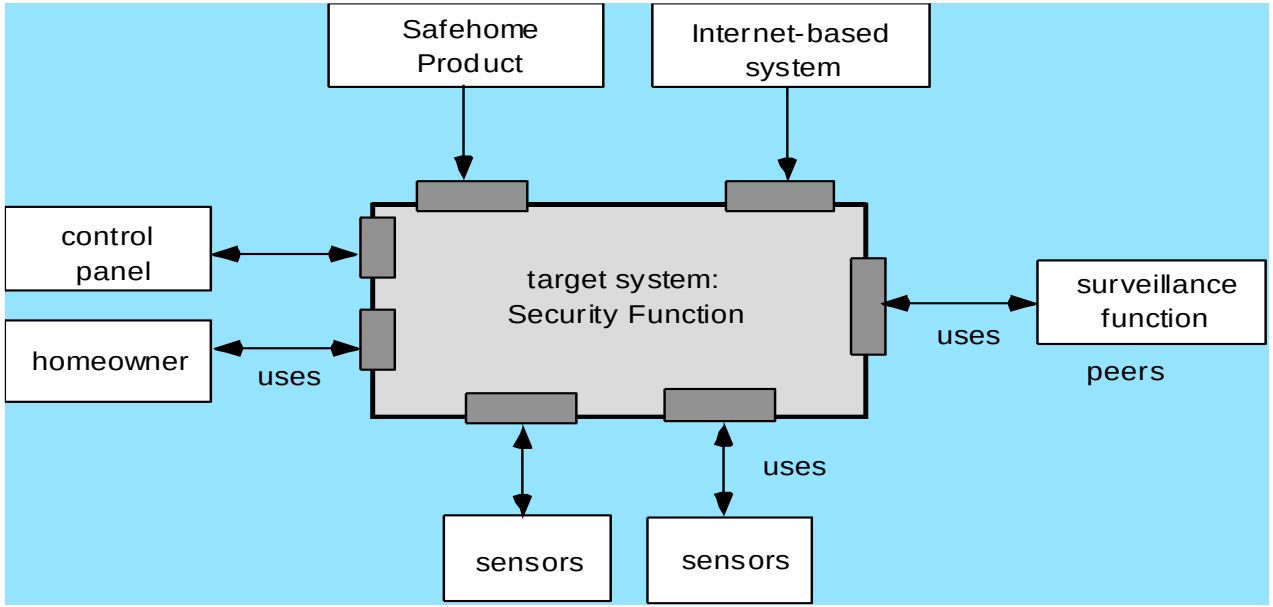


Architectural Patterns

- *Concurrency*（*并发性*）—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(***代理***)*** acts as a ‘middle-man’ between the client component and a server component.

- **Architectural Design**
  - The software must be placed into context
    - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
  - A set of architectural archetypes should be identified
    - An ***archetype(原型)*** is an abstraction (similar to a class) that represents one element of system behavior
  - The designer specifies the structure of the system by defining and refining software components that implement each ***archetype***

- **Architectural Context**



- **Archetypes**

（原型）

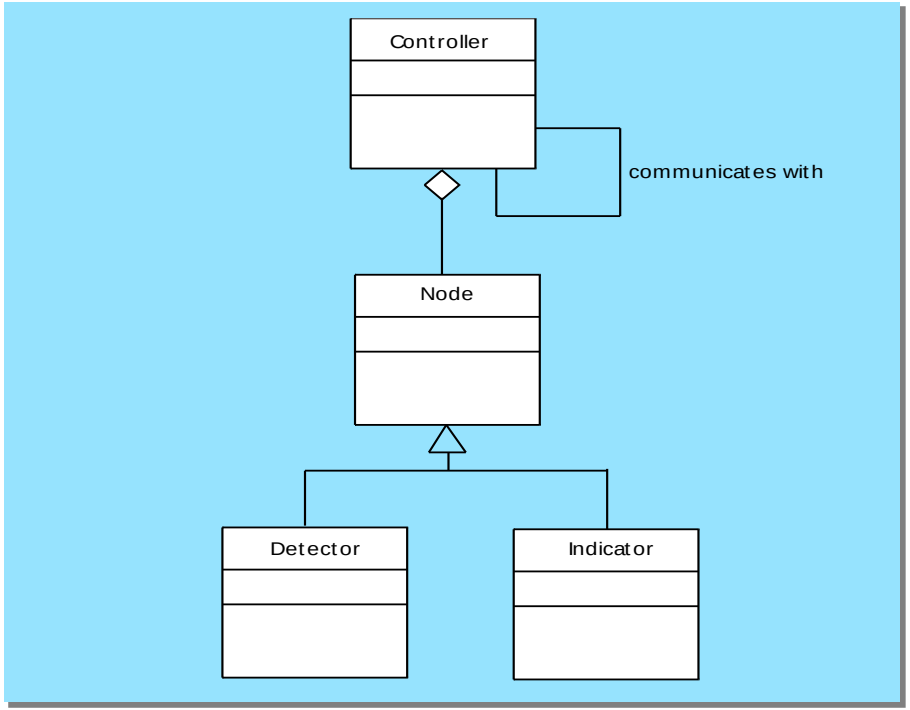
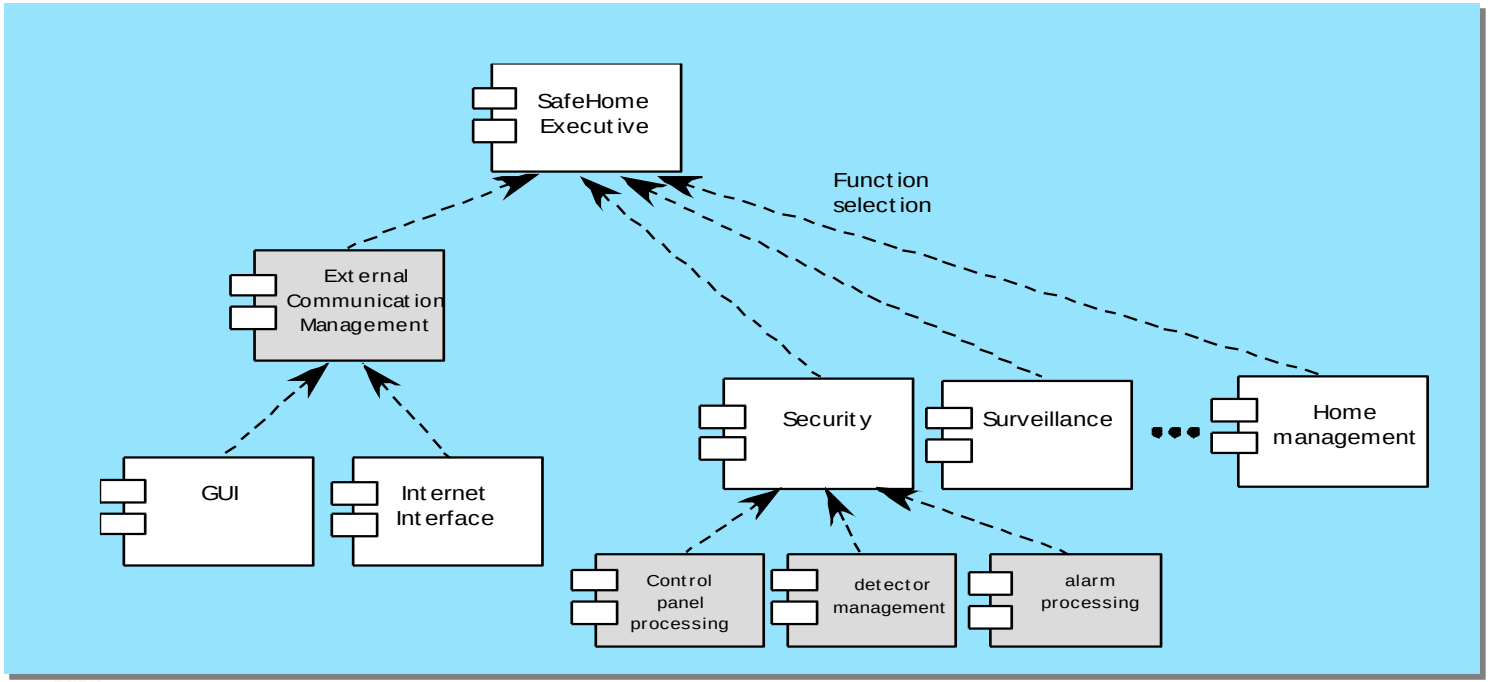
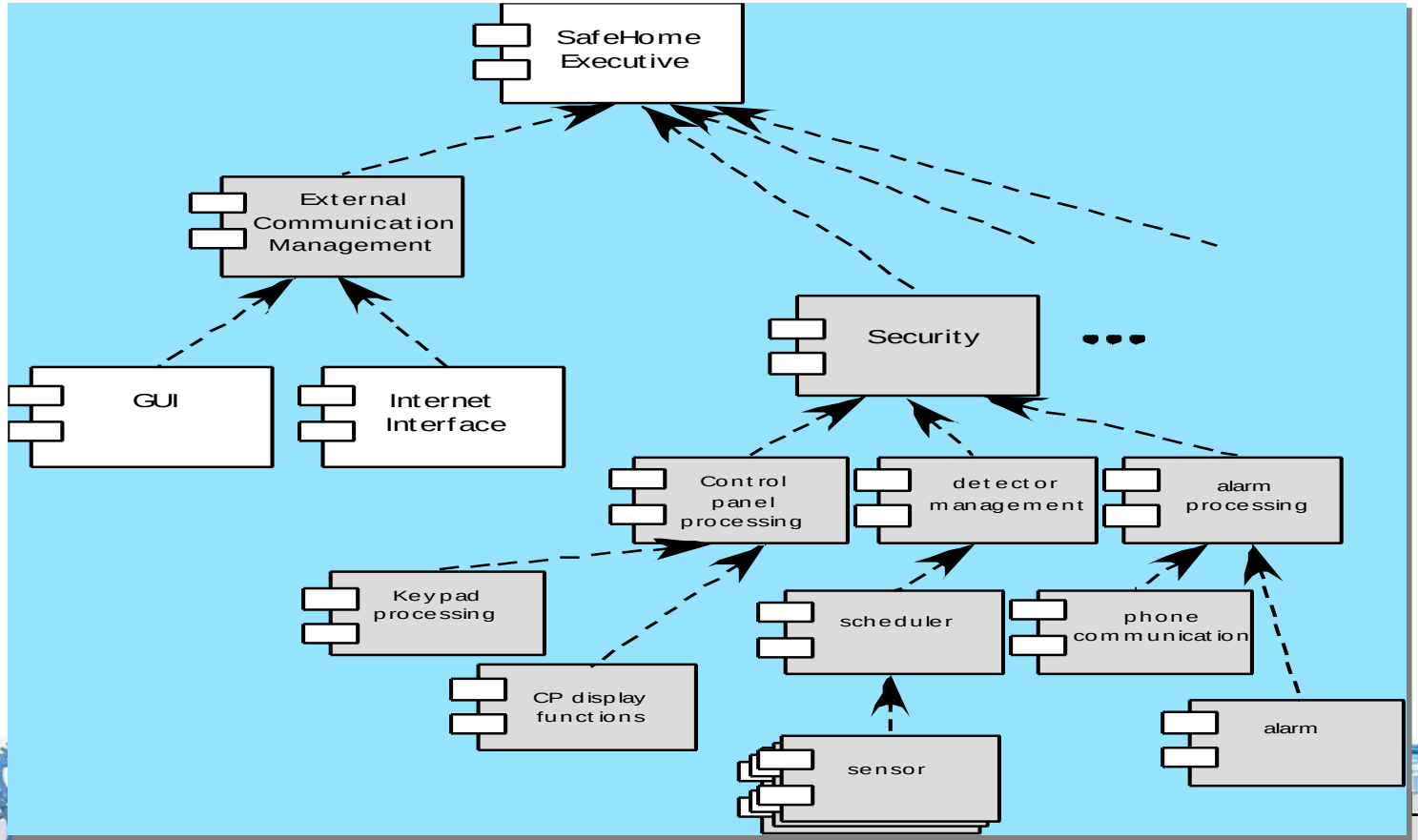


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

- **Component Structure**



Refined Component Structure



• Architectural Tradeoff Analysis

1）Collect scenarios.

2）Elicit requirements, constraints, and environment description.

3）Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:

- module view →assignments with components
- process view →system performance
- data flow view →functional requirements

4）Evaluate quality attributes by considered each attribute in isolation(reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability).

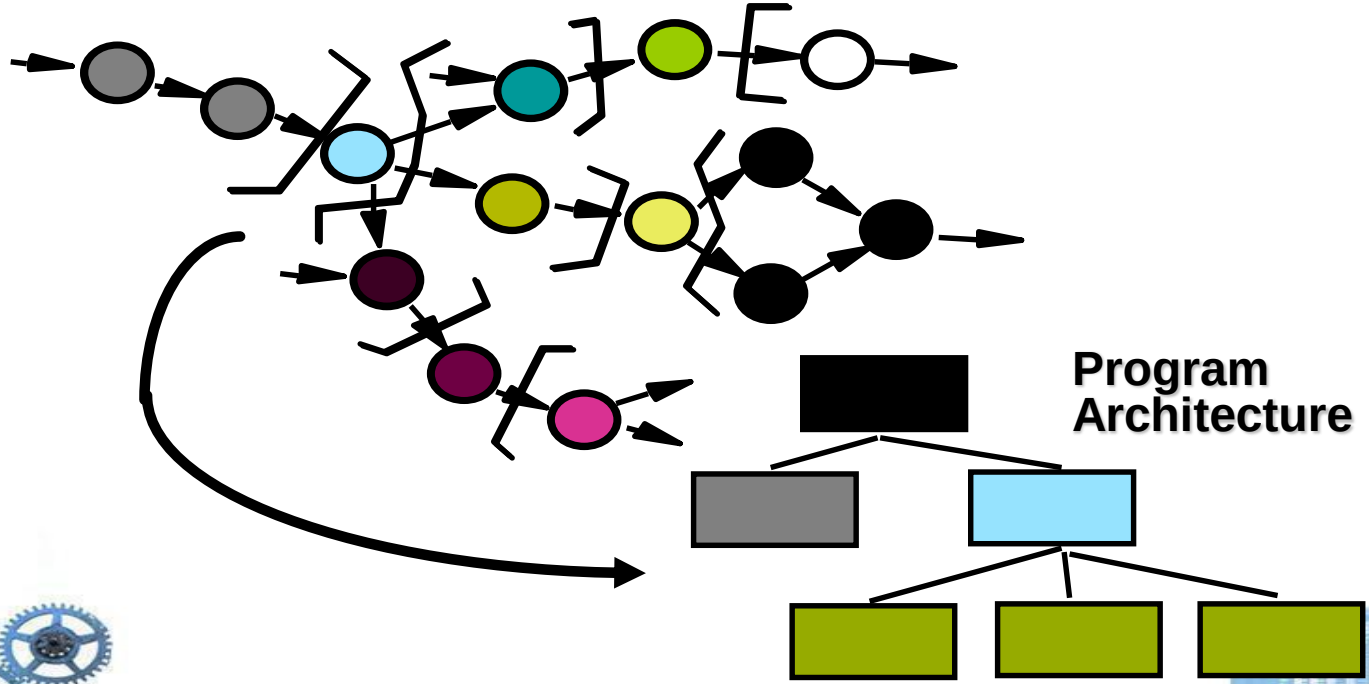
5）Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.（e.g. **C-S model →server number**）

6）Critique（评估） candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Mapping Data Flow into a Software Architecture

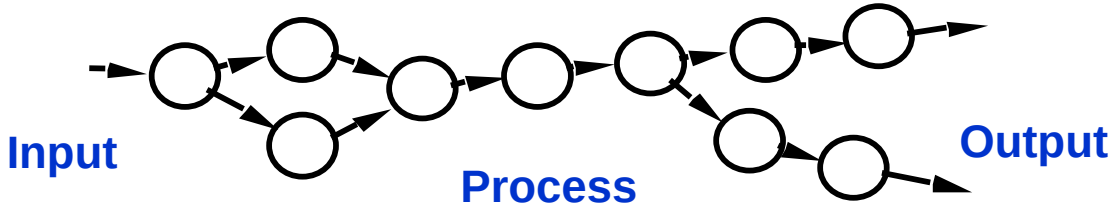
- **Structured Design**

  — an architectural design method, deriving the ***call and return*** architecture

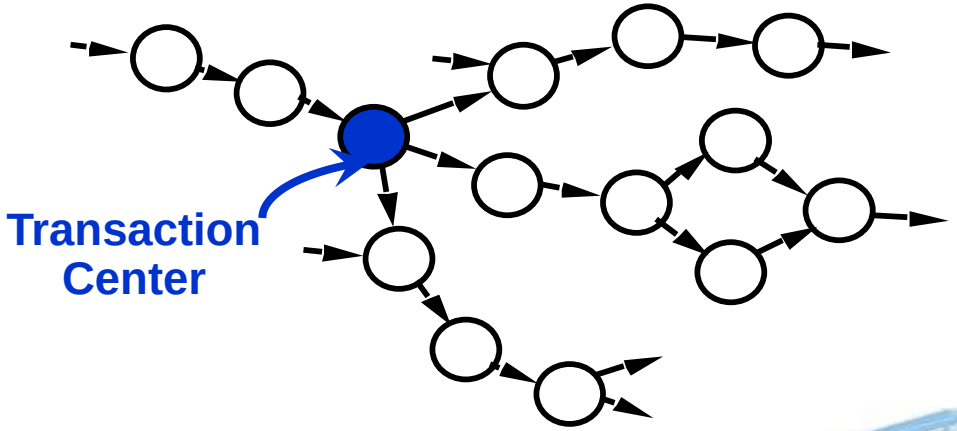


Mapping Data Flow into a Software Architecture

- **Transform Flow**

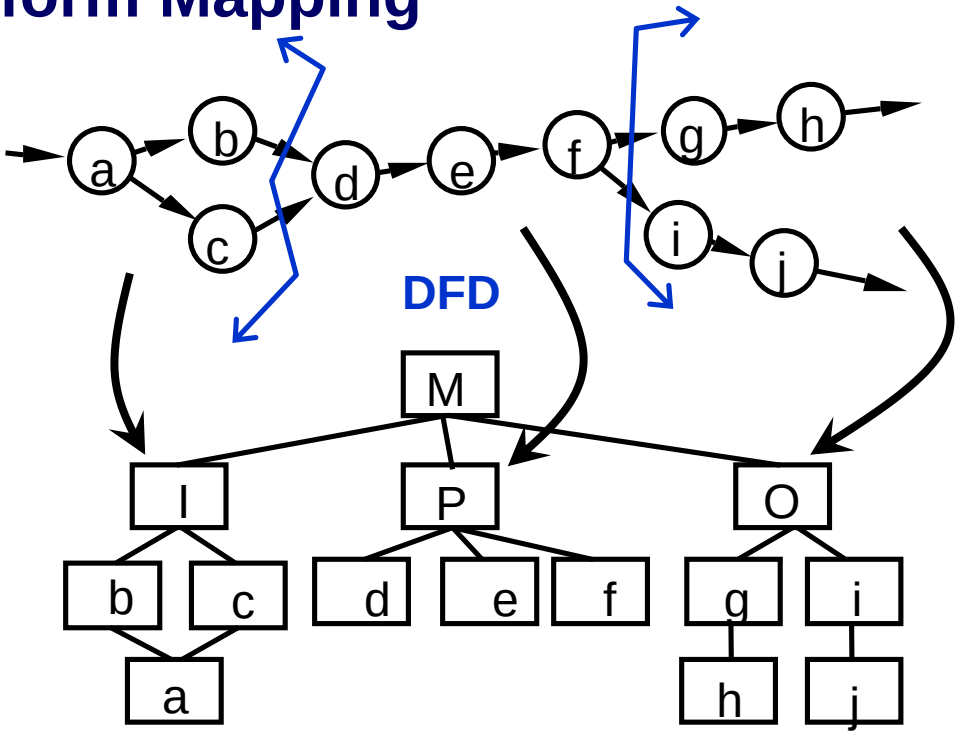


- **Transaction Flow**



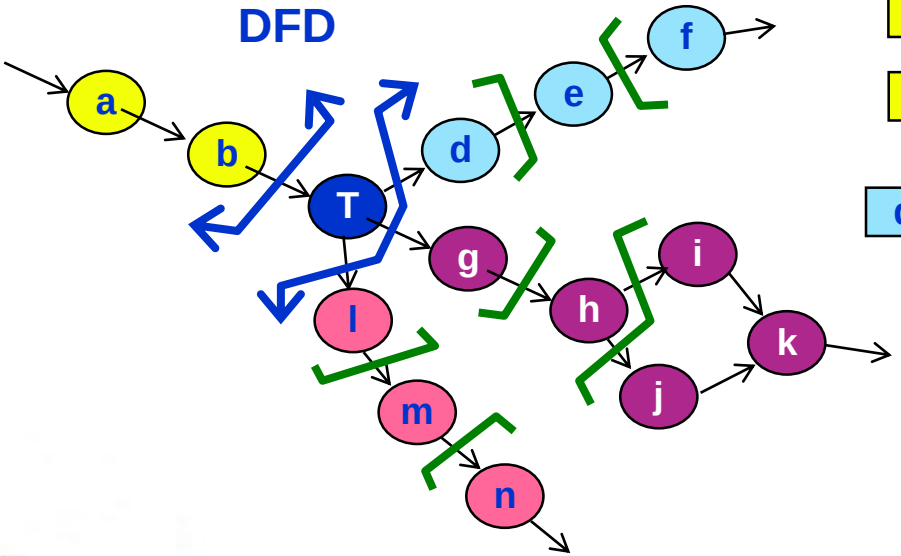
Mapping Data Flow into a Software Architecture

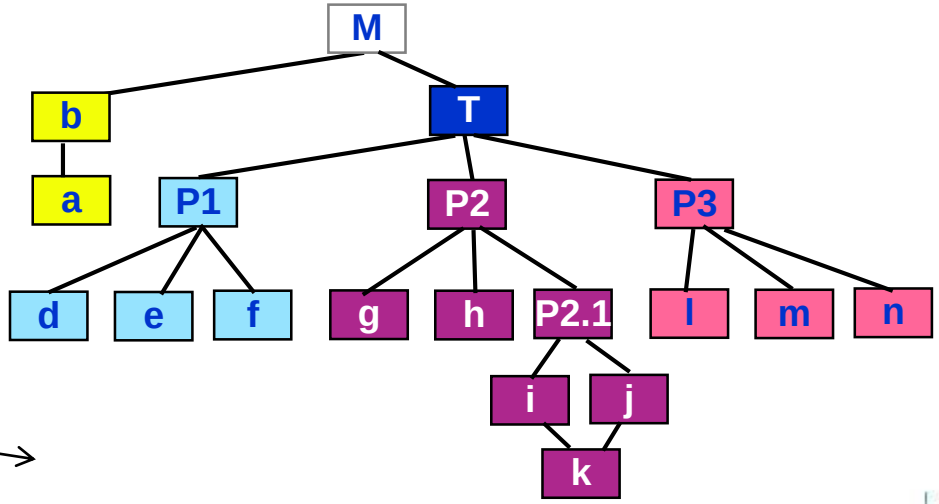
- **Transform Mapping**



Mapping Data Flow into a Software Architecture

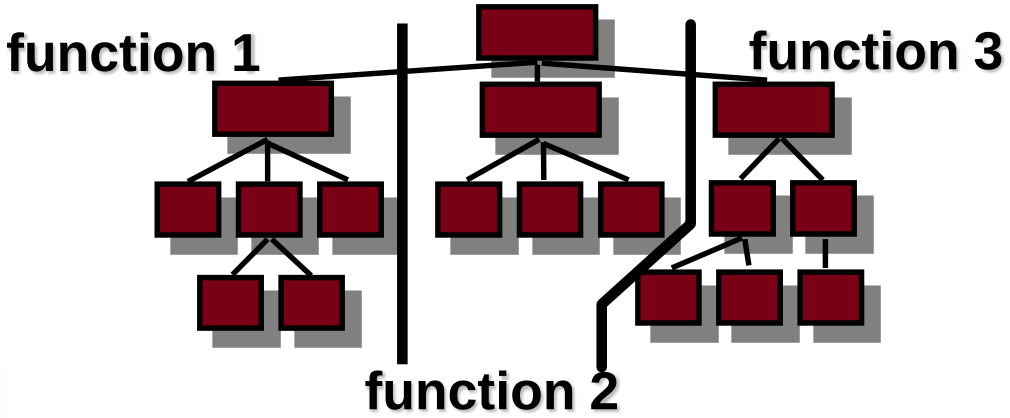
- **Transaction Mapping**





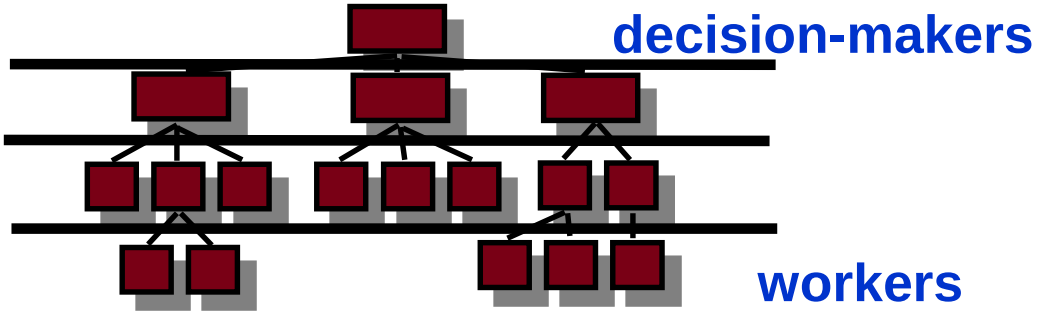
Mapping Data Flow into a Software Architecture

- **Partitioning Program Architecture**
  - **Horizontal Partitioning**
    - — define separate branches of the module hierarchy for each major function
    - — use control modules to coordinate communication between functions



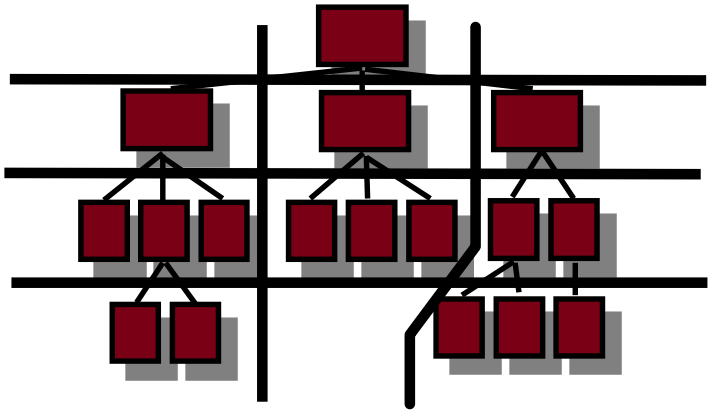
Mapping Data Flow into a Software Architecture

- **Partitioning Program Architecture**
  - **Vertical Partitioning**
    - — design so that decision making and work are stratified
    - — decision making modules should reside at the top of the architecture



Mapping Data Flow into a Software Architecture

- **Partitioning Program Architecture**



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

• Architectural Complexity

- the overall complexity of a proposed architecture is assessed by considering the *dependencies* between components within the architecture [Zha98]

  - ***Sharing dependencies*** represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
  - ***Flow dependencies*** represent dependence relationships between producers and consumers of resources.
  - ***Constrained dependencies*** represent constraints on the relative flow of control among a set of activities (EX. If A Then u Else v).

ADL

- *Architectural Description Language (ADL)* provides a semantics and syntax for describing a software architecture

- Provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks and
  - represent interfaces (connection mechanisms) between components.
  - e.g. xArch, UniCon, Wright, Acme, **UML**, etc. See pp.277

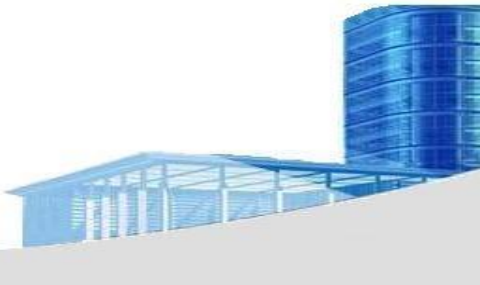
Architecture Reviews

- Assess the ability of the software architecture to meet the systems quality requirements and identify potential risks
- Have the potential to reduce project costs by detecting design problems early
- Often make use of experience-based reviews, prototype evaluation, and scenario reviews, and checklists（清单，检查表）

• Agility and Architecture

- To avoid rework, user stories are used to create and evolve an architectural model (walking skeleton，系统骨架) before coding；
- Hybrid models which allow software architects contributing （起作用的） 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.

Ch.14 Component-Level Design



What is a Component?

"a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

——*OMG Unified Modeling Language Specification* [OMG01]

- *O-O view*: a component contains a set of collaborating classes
- *Conventional view*: a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked(调用) and data to be passed to it.

• Basic design principles

- **The Open-Closed Principle (OCP).** *"A module [component] should be open for extension but closed for modification. Ex. SafeHome's Detector*
- **The Liskov Substitution Principle (LSP).** *"Subclasses should be substitutable for their base classes.*
- **Dependency Inversion Principle (DIP).** *"Depend on abstractions. Do not depend on concretions."*
- **The Interface Segregation Principle (ISP).** *"Many client-specific interfaces are better than one general purpose interface.*
- **The Release Reuse Equivalency Principle (REP).** *"The granule（粗粒度） of reuse is the granule of release."*
- **The Common Closure Principle (CCP).** *"Classes that change together belong together."*
- **The Common Reuse Principle (CRP).** *"Classes that aren't reused together should not be grouped together."*

Design Guidelines

- *Components*
  - Naming conventions(命名约定) should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Ex. FloorPlan
- *Interfaces*
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP)
- *Dependencies and Inheritance*
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

• Cohesion（内聚性）

- Conventional view:
  - the “single-mindedness” (专诚性、单一性)of a module
- O-O view:
  - cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - Utility(功用)

Coupling

- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Common
  - Control
  - Stamp (标记)
  - Data
  - Routine call (程序调入)
  - Type use
  - Inclusion or import
  - External

Component Level Design

1. Identify all design classes that correspond to the ***problem domain***.
2. Identify all design classes that correspond to the ***infrastructure domain***.
3. ***Elaborate*** all design classes that are not acquired as reusable components.
4. Describe ***persistent data sources*** (databases and files) and identify the classes required to ***manage*** them.
5. Develop and elaborate ***behavioral*** representations for a class or component.
6. Elaborate ***deployment diagrams*** (Ch.8)to provide additional implementation detail.
7. ***Factor*** every component-level design representation and always consider ***alternatives***.

Component Level Design

1. Identify all design classes that correspond to the ***problem domain***.
2. Identify all design classes that correspond to the ***infrastructure domain***.
3. ***Elaborate*** all design classes that are not acquired as reusable components.
4. Describe ***persistent*** ***sources*** (databases and files) and identify
5. Develop
compon
6. Elabora
impleme
7. ***Factor*** e
consider ***alternatives***.

3.1 Specify ***message*** details when classes or component collaborate.

3.2 Identify appropriate ***interfaces*** for each component.

3.3 Elaborate ***attributes*** and define data types and data structures required to implement them.

3.4 Describe ***processing flow*** within each operation in detail.

- **Component Design for WebApps**

  - WebApp component is
    - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
    - (2) a cohesive package of content and functionality that provides end-user with some required capability.
  - Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

• Content Design for WebApps

- focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user

---consider a Web-based video surveillance capability within SafeHomeAssured.com

- potential content components can be defined for the video surveillance capability:
  - (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
  - (2) the collection of thumbnail(姆指) video captures (each an separate data object), and
  - (3) the streaming video window for a specific camera.
- Each of these components can be separately named and manipulated as a package.

Functional Design for WebApps

- Modern Web applications deliver increasingly sophisticated processing functions that:
  - (1) perform localized processing to generate content and navigation capability in a dynamic fashion;
  - (2) provide computation or data processing capability that is appropriate for the WebApp's business domain;
  - (3) provide sophisticated database query and access, or
  - (4) establish data interfaces with external corporate systems.
- To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.

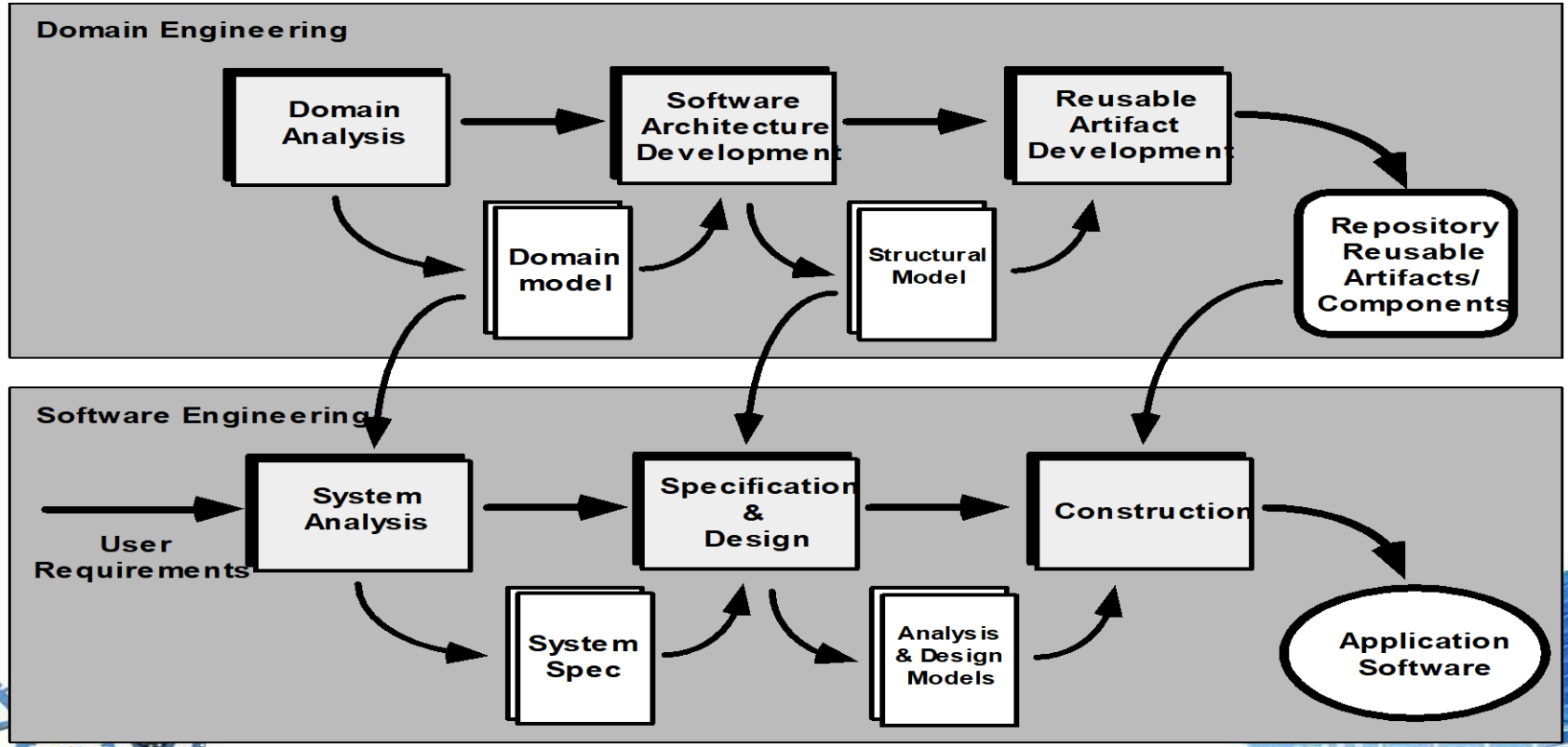
- **Designing Conventional Components**
  - The design of processing logic is governed by the basic principles of algorithm design and structured programming
  - The design of data structures is defined by the data model developed for the system
  - The design of interfaces is governed by the collaborations that a component must effect.

• Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement? Ex. UE (Unreal Engine)
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible(兼容的) within the architecture of the system to be built?

- At the same time, they are faced with some impediments(障碍) to reuse ...

- **The CBSE Process**（Component Based Software Engineering）



- **Domain Engineering**
  - 1. Define the domain to be investigated.
  - 2. Categorize the items extracted from the domain.
  - 3. Collect a representative sample of applications in the domain.
  - 4. Analyze each application in the sample.
  - 5. Develop an analysis model for the objects.

- **Component-Based SE （CBSE）**
  - a library of components must be available
  - components should have a consistent structure
  - a standard should exist, e.g.,
    - OMG/CORBA
    - Microsoft COM
    - Sun JavaBeans

- **CBSE Activities (Read the book for details!!)**

-

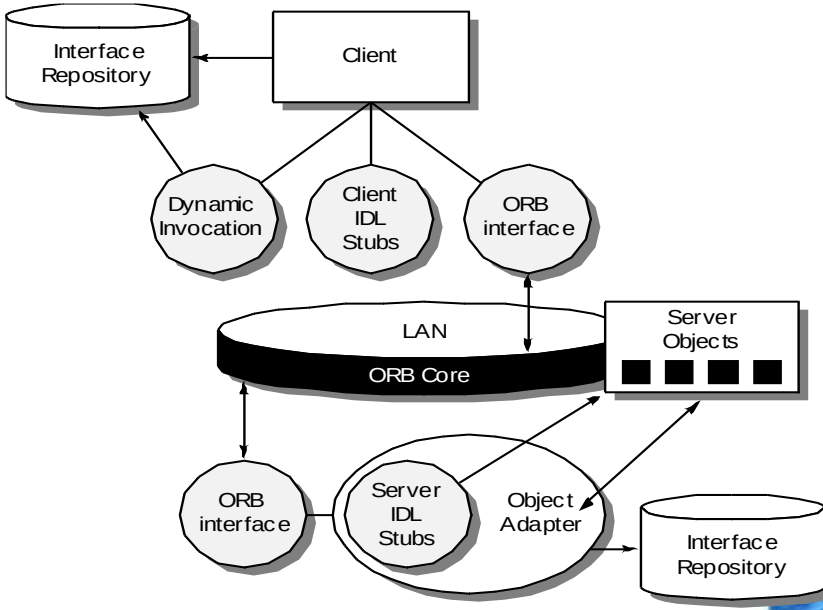
  - Component qualification
  - Component adaptation
  - Component composition
  - Component update

- **OMG/ CORBA**

  - The Object Management Group has published a *common object request broker architecture* (OMG/**CORBA**).

  - An object request broker(经纪人) (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

- **ORB Architecture**



Microsoft COM

- The component object model (COM) provides a specification for using components produced by various vendors(供应商) within a single application running under the Windows operating system.
- COM encompasses two elements:
  - COM interfaces (implemented as COM objects)
  - a set of mechanisms for registering and passing messages between COM interfaces.

- **Sun JavaBeans**
  - The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
  - The JavaBeans component system encompasses a set of tools, called the Bean Development Kit (BDK), that allows developers to
    - analyze how existing Beans (components) work
    - customize their behavior and appearance
    - establish mechanisms for coordination and communication
    - develop custom Beans for use in a specific application
    - test and evaluate Bean behavior.

Tasks

- **Review** Ch.13,14
- **Finish** “Problems and points to ponder” in Ch. 13,14
- **Preview** Ch. 15,16, 17
- Experimental time： Tomorrow afternoon, Room 曹西503;
- **Prepare to submit System Design Report due April 22!**
- **Prepare System Design Speech on April 23!**