Architectural Design

- **Modeling: Chapter 13**

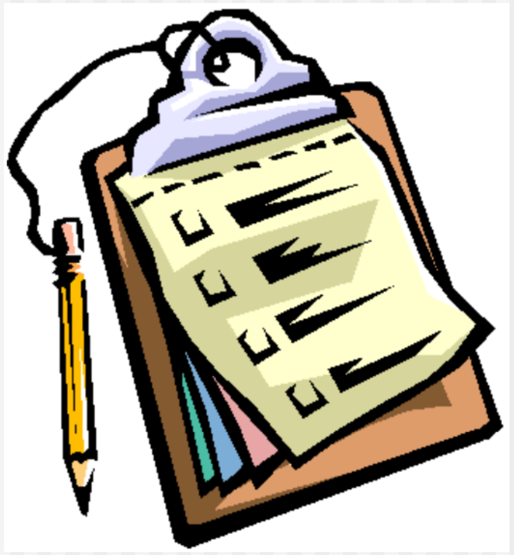
Slide Set to accompany

Software Engineering: A Practitioner’s Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

Agenda

- Software Architecture
- Architecture Genres and Styles
- Architectural Design



Software Architecture

What is Architecture?

- The structure of the system,
  - which comprise software **components**,
  - the externally visible **properties** of those components,
  - and the **relationships** among them
- 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 when making design changes, and
  3. reduce the risks associated with the construction of the software.

Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders)
- 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
- Architecture “constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together” [BAS03].

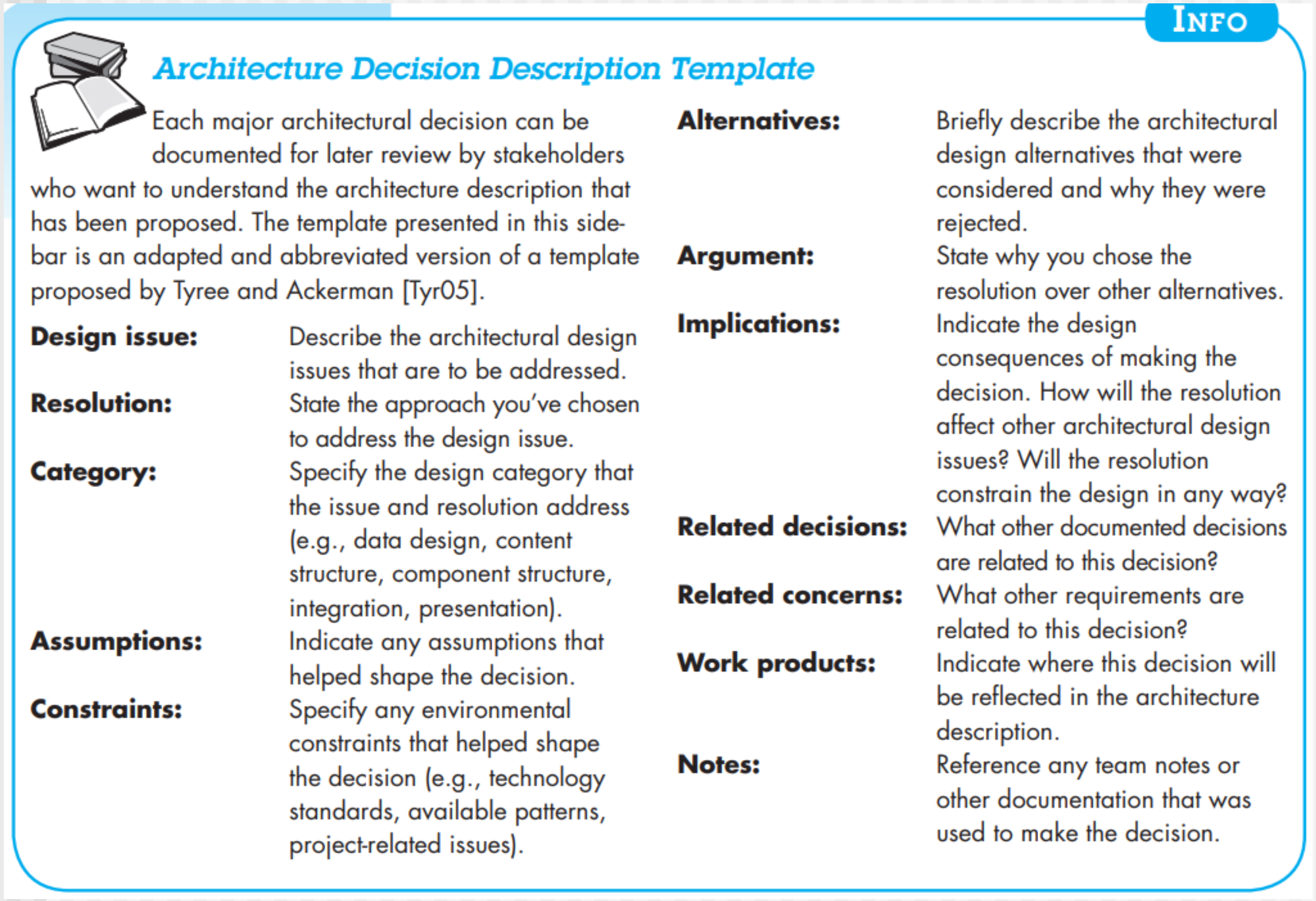
Architectural Descriptions

- Architectural description is a **set of work products** that reflect **different views** of the system
- The IEEE Standard defines an *architectural description* (AD) as a "a collection of products to document an architecture."
  - The description represents multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Decisions

- Each view of an architectural description addresses a specific stakeholder concern
- To develop each view,
  - the architect considers a **variety of alternatives**
  - and ultimately decides on the specific architectural features that **best meet** the concern

A Sample Architecture Decision Template



INFO

Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue:	Describe the architectural design issues that are to be addressed.
Resolution:	State the approach you've chosen to address the design issue.
Category:	Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).
Assumptions:	Indicate any assumptions that helped shape the decision.
Constraints:	Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).
Alternatives:	Briefly describe the architectural design alternatives that were considered and why they were rejected.
Argument:	State why you chose the resolution over other alternatives.
Implications:	Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?
Related decisions:	What other documented decisions are related to this decision?
Related concerns:	What other requirements are related to this decision?
Work products:	Indicate where this decision will be reflected in the architecture description.
Notes:	Reference any team notes or other documentation that was used to make the decision.

Architectural Genres & Styles

Architectural Genres

- *Genre* implies a **specific category** within the overall software domain
  - Also called an application domain
  - The architectural genre will often dictate the specific architectural approach to the structure
- Grady Booch suggests the following architectural genres for software systems:
  - artificial intelligence, communications, financial, games, industrial, medical, military, transportation, …
- A number of different **architectural styles** may be applicable to a specific genre

Architectural Styles

- An architectural style describes:
  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.

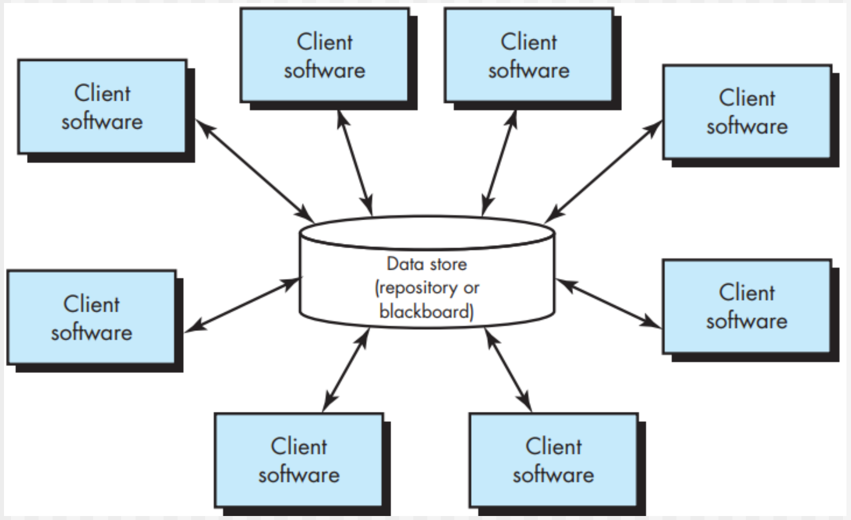
A Brief Taxonomy of Architectural Styles

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

Data-Centered Architecture-I

- A data store (e.g., a file or database) resides at the center
- The data store is accessed frequently by other components
  - To update, add, delete, or modify data within the store

1. Passive repository
   - Client software accesses the data independent of any changes to the data or the actions of other clients
2. Blackboard repository
   - sends notifications to client when data changes
   - blackboard component coordinates the transfer of information between clients

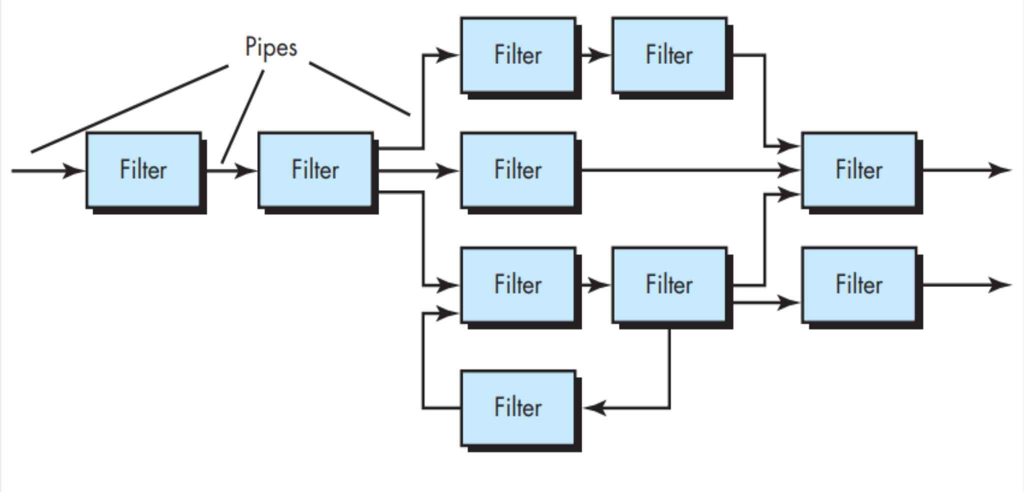


Data-Centered Architecture-II

- Data-centered architectures promote *integrability* [Bas03].
- 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
- Client components independently execute processes

Data-Flow Architecture-I

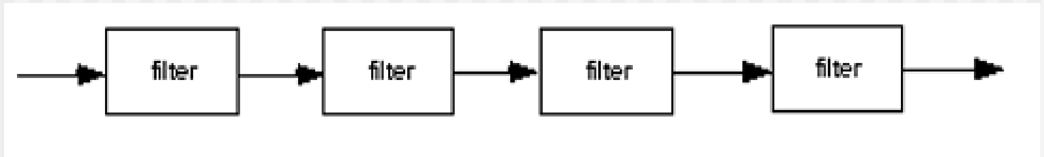
- Applied when input data are to be transformed into output data through a series of computational or manipulative components
- A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next
- Each filter works independently of those components upstream and downstream
  - expect data input of a certain form
  - produces data output (to the next filter) of a specified form



Pipes and filters

Data-Flow Architecture-II

- Batch sequential
  - The data flow degenerates into a single line of transforms
  - This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it

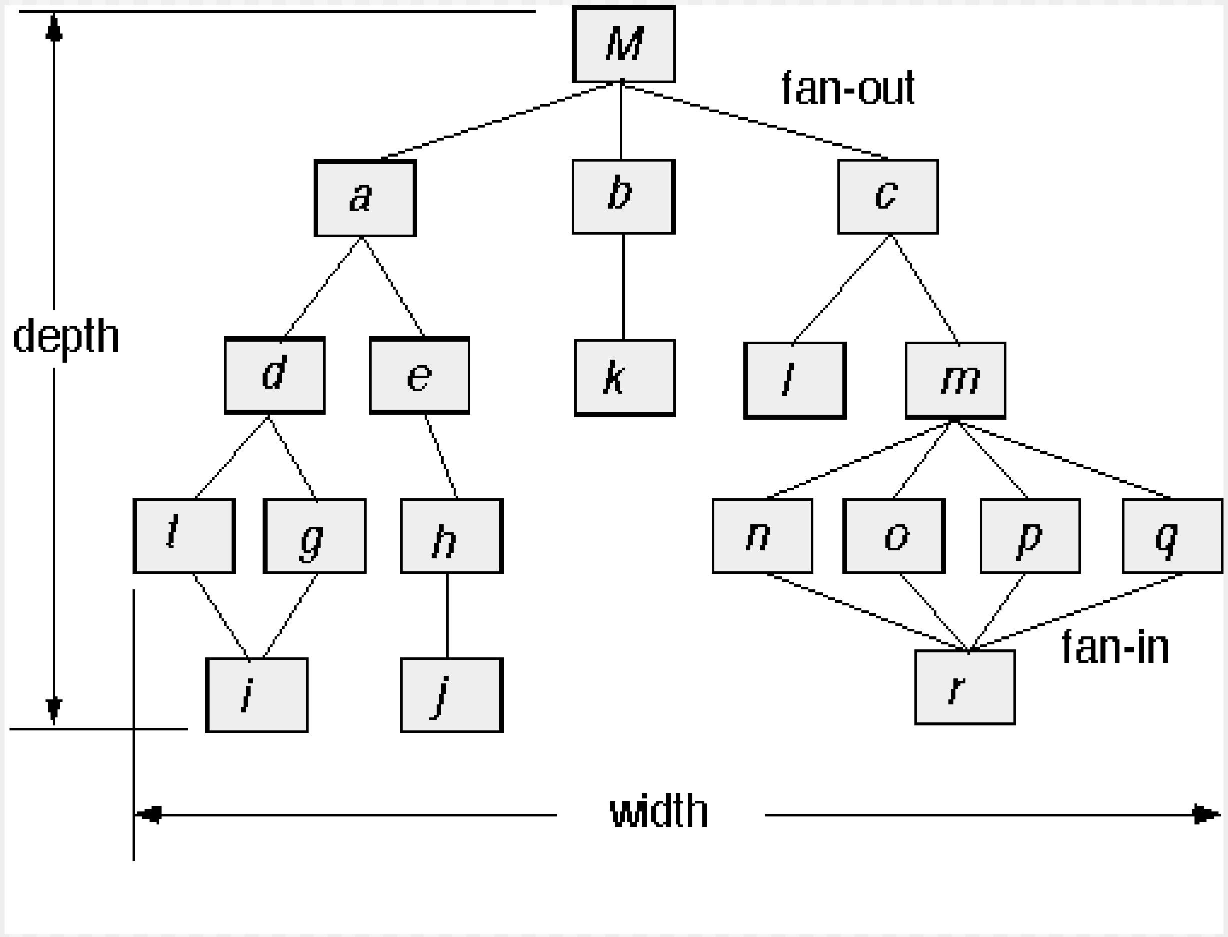


(b) batch sequential

Call and Return Architecture-I

- Enables you to achieve a program structure that is relatively easy to modify and scale
- A number of substyles within this category:
  1. Main program/subprogram architectures
     - This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components
  2. Remote procedure call architectures
     - The components of a main program/subprogram architecture are distributed across multiple computers on a network

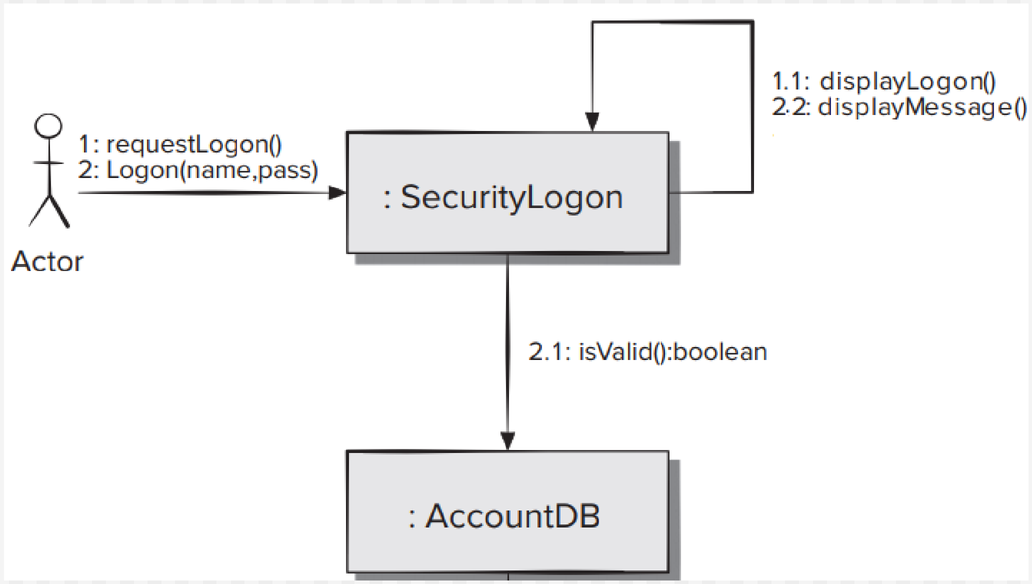
Call and Return Architecture-II



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

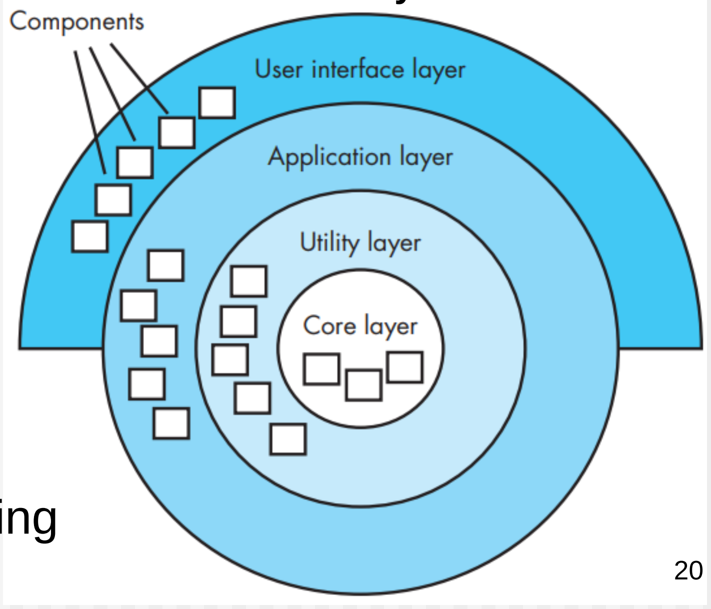
A UML communication diagram that shows the message passing for the login portion of a system using an object-oriented architecture



Layered Architecture

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set
- User interface layer
  - At the outer layer, components service user interface operations
- Application layer
  - provides application software functions
- Utility layer
  - provides utility services
- Core layer
  - At the inner layer, components perform operating system interfacing

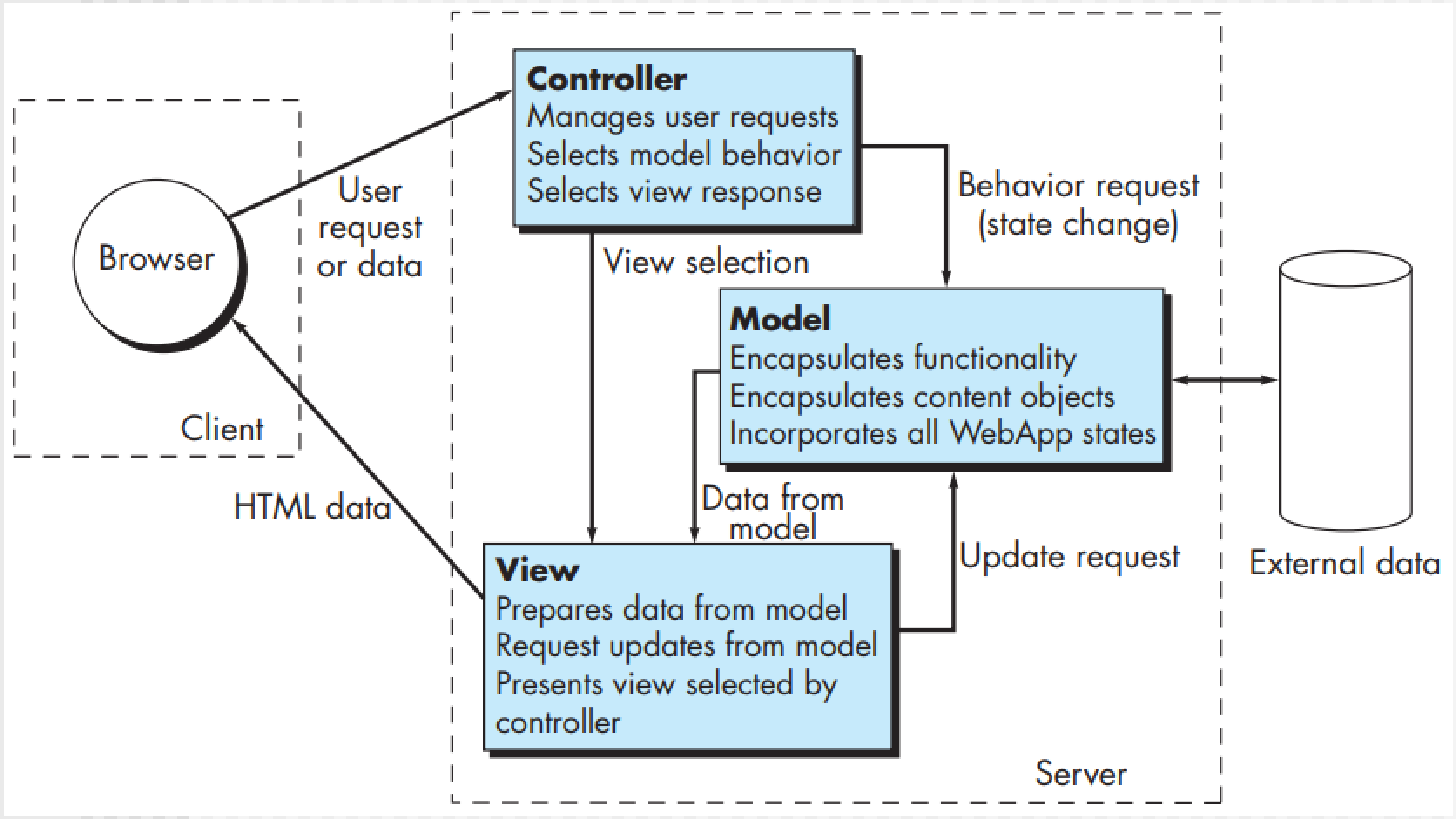
The basic structure of a layered architecture



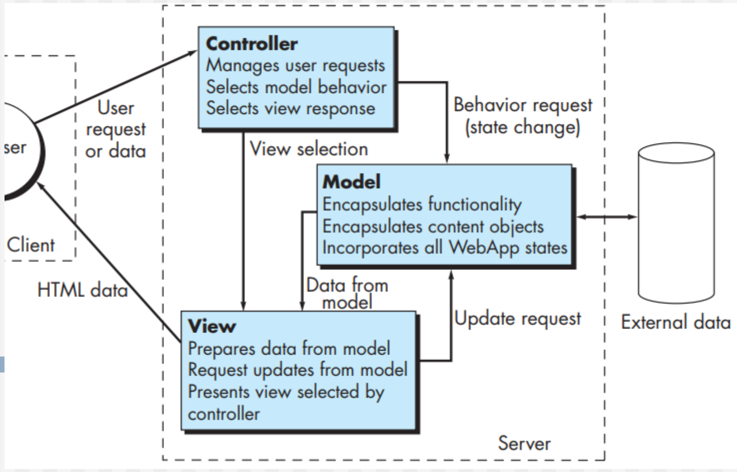
The MVC Architecture-I

- The model-view-controller (MVC) architecture
- Often used in WebApp design
- A three-layer design architecture that decouples the user interface from the WebApp functionality & content
- Model
  - contains all **application-specific content** and processing **logic**
- View
  - contains all **interface-specific** functions and enables the presentation of content and processing logic to the user
- Controller
  - **manages access** to the model and the view and **coordinates** the flow of data between them
- In a WebApp, “the view is updated by the controller with data from the model based on user input” [WMT02]

The MVC Architecture-II



The MVC Architecture-III



- User requests are handled by the controller
- The controller also selects the view object that is applicable based on the user request
- Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request
- The model object can access data stored in a database
- The data developed by the model must be formatted and organized by the appropriate view object
- And then transmitted from the application server back to the client-based browser for display on the customer’s machine

Architectural Patterns

- A pattern is a recurring solution to a recurring problem
- Architectural patterns address an application-specific problem within a specific context
  - The pattern proposes an architectural solution that can serve as the basis for architectural design
- They solve the problems related to the architectural style
  - For example:
  - *“What classes will we have and how will they interact, in order to implement a system with a specific set of layers“*
  - *“What high-level modules will have in our Service-Oriented Architecture and how will they communicate“*
- Examples of Architectural Patterns:
  - 3-tier, Microkernel, MVC, …

Architectural Style vs Architectural Pattern

- Style is a **concept**, theory (and how it's implemented is up to you)
- An architectural pattern describes a **solution** for implementing a style
  - At the level of subsystems or modules and their relationships
- For example, the overall architectural style for an application might be call-and-return or layered
  - But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns
    - Such as: operating system process management pattern, task scheduler pattern, database management system (DBMS) pattern, …

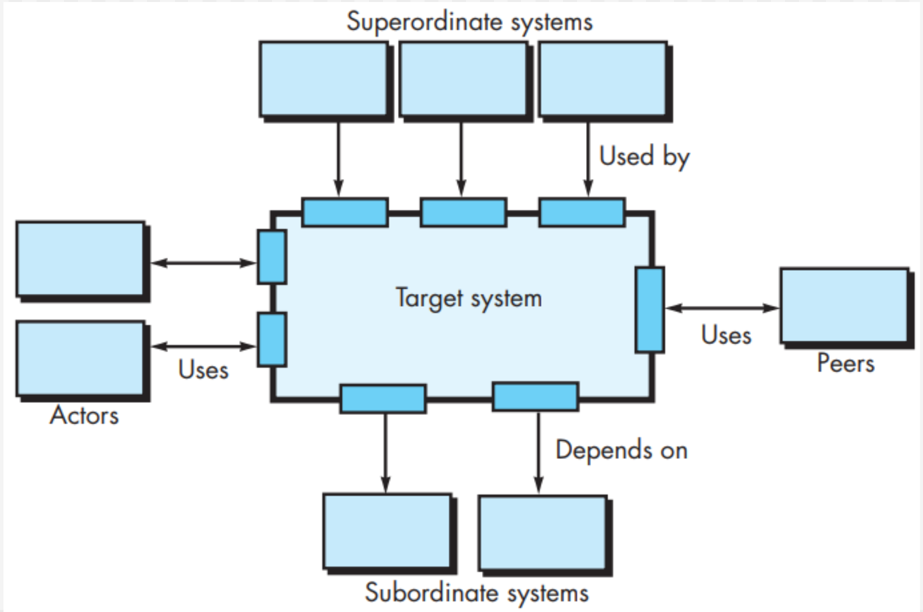
Architectural Design

Architectural Design

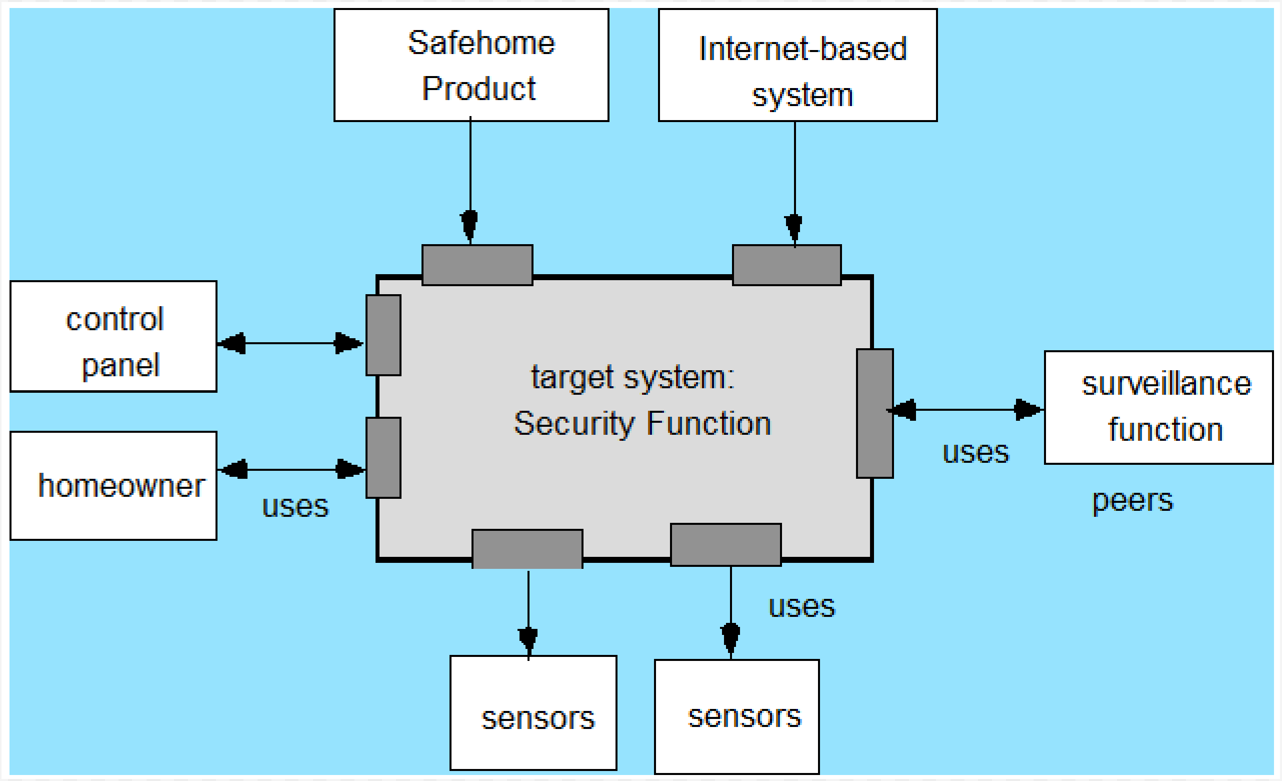
- The software must be placed into **context**
- As architectural design begins, context must be established
  - Define the **external entities** (other systems, devices, people) that the software interacts with and the nature of the interaction
- This information can generally be acquired from the requirements model
- Architectural Context Diagram (ACD)
  - models the manner in which software interacts with entities external to its boundaries

Architectural Context Diagram

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



ACD For the SafeHome security function



Architectural Design (cont.)

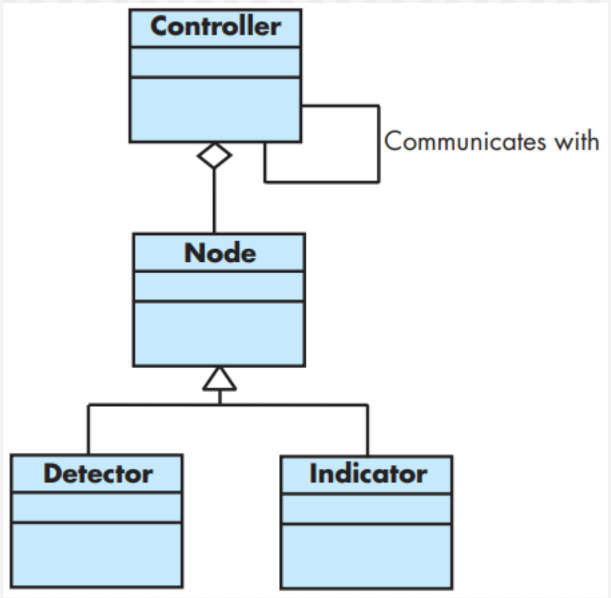
- Once context is modeled and all external software interfaces have been described,
  - a set of architectural archetypes should be identified
- An ***archetype*** is an abstraction (similar to a class) that represents one element of system behavior
- The set of archetypes provides a collection of abstractions to model the architectural structure
  - But the archetypes themselves do not provide enough implementation detail
- 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

Defining Archetypes

- Abstract building blocks of an architectural design
- 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
- Archetypes can be derived by examining the analysis classes defined as part of the requirements model

Archetypes Example:
UML relationships for SafeHome security function archetypes

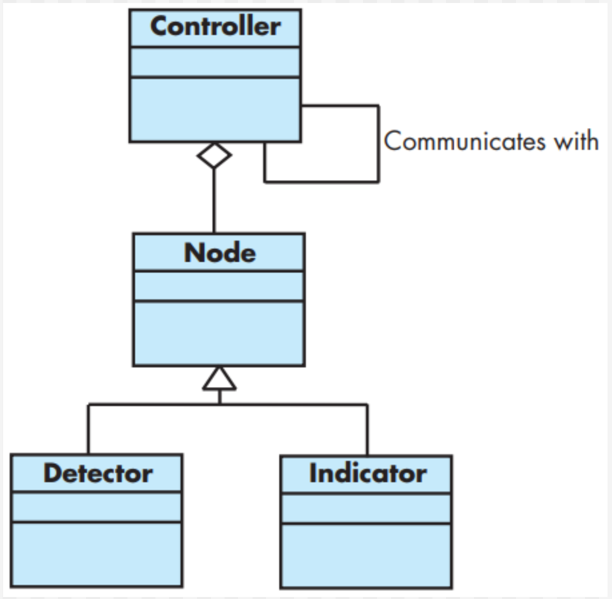
- **Node:** Represents a cohesive collection of input and output elements of the home security function
  - For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators
- **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the target system
- **Indicator:** An abstraction that represents all mechanisms for indicating an alarm condition
  - e.g., alarm siren, flashing lights, bell, ...



Archetypes Example:

UML relationships for SafeHome security function archetypes

- **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
- Recall that the archetypes form the basis for the architecture
  - but are abstractions that must be further refined as architectural design proceeds
  - For example, **Detector** might be refined into a class hierarchy of sensors



Refining the Architecture into Components

- How to choose software components?
- Two sources for the derivation and refinement of components:
  1. The application domain (i.e., analysis classes)
  2. The infrastructure domain
     - The architecture must accommodate many infrastructure components that **enable application components** but have <u>no business connection to the application domain</u>
     - For example:
       - memory management components, communication components, database components, and task management components

       are often integrated into the software architecture

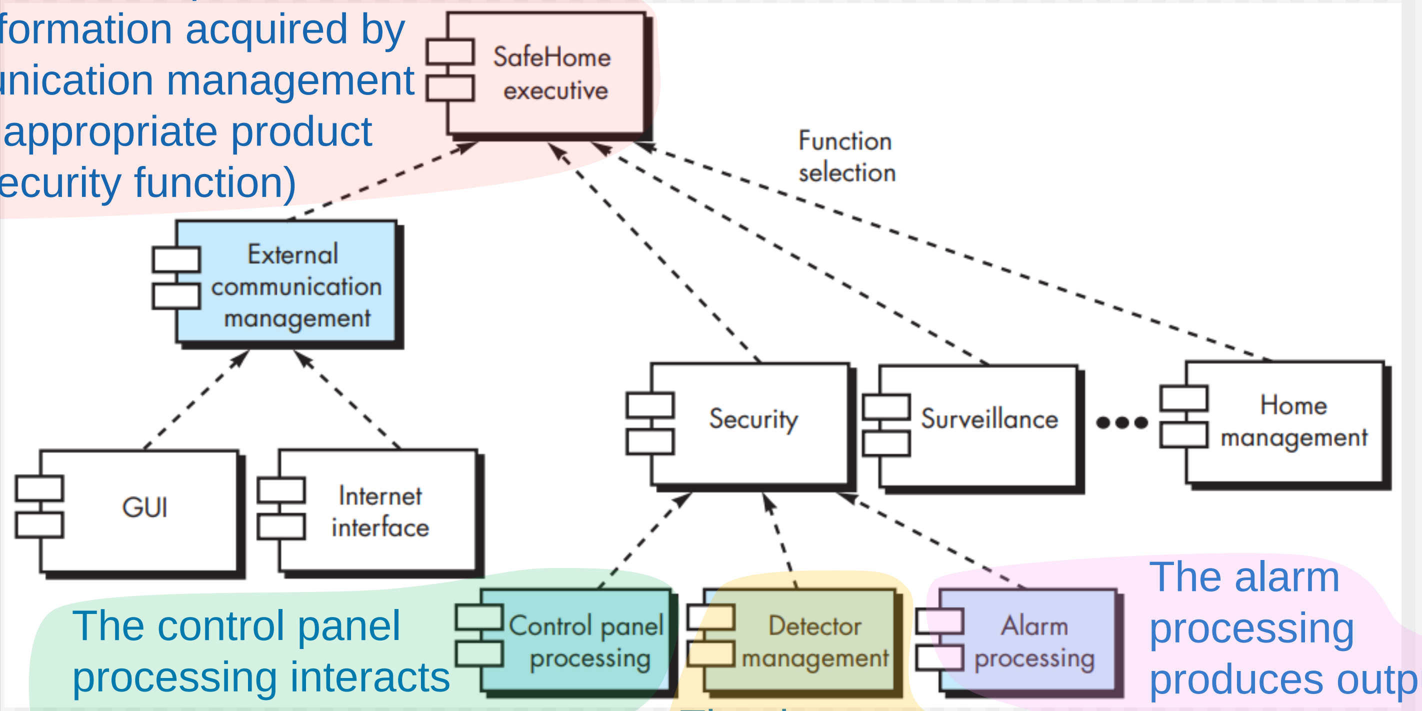
Component Structure: An Example

- Top-level components for the SafeHome home security function:
  - External communication management—coordinates communication of the security function with external entities such as other Internet-based systems
  - 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
- *Note*
  - *Each of these top-level components should be elaborated iteratively and then positioned within the overall architecture*
  - *Design classes should be defined for each*
  - *However the design details of all attributes and operations would not be specified until component-level design*

Component Structure: An Example

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

SafeHome executive component manages the information acquired by external communication management and selects the appropriate product function (e.g., security function)



The control panel processing interacts with the homeowner to arm/disarm the security function

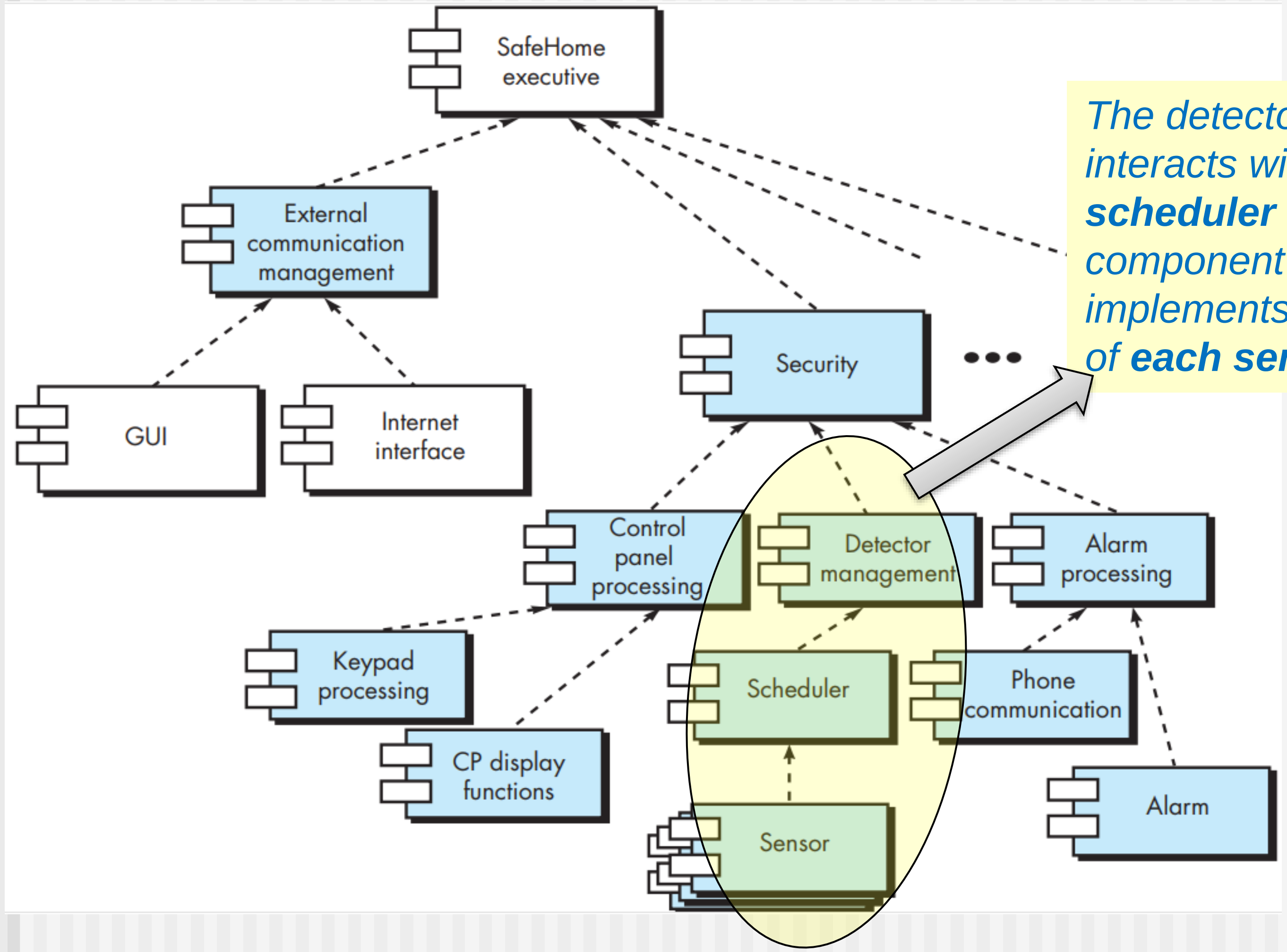
The detector management polls sensors to detect an alarm condition

The alarm processing produces output when an alarm is detected

Describing Instantiations of the System

- The architectural design to this point is still relatively **high level**
- The **major software components** have been identified
- However, **further refinement** is still necessary (recall that all design is iterative)
- To accomplish this, an actual instantiation of the architecture is developed
  - **Component elaboration**

Refined Component Structure



*The detector interacts with a **scheduler** component that implements polling of **each sensor***

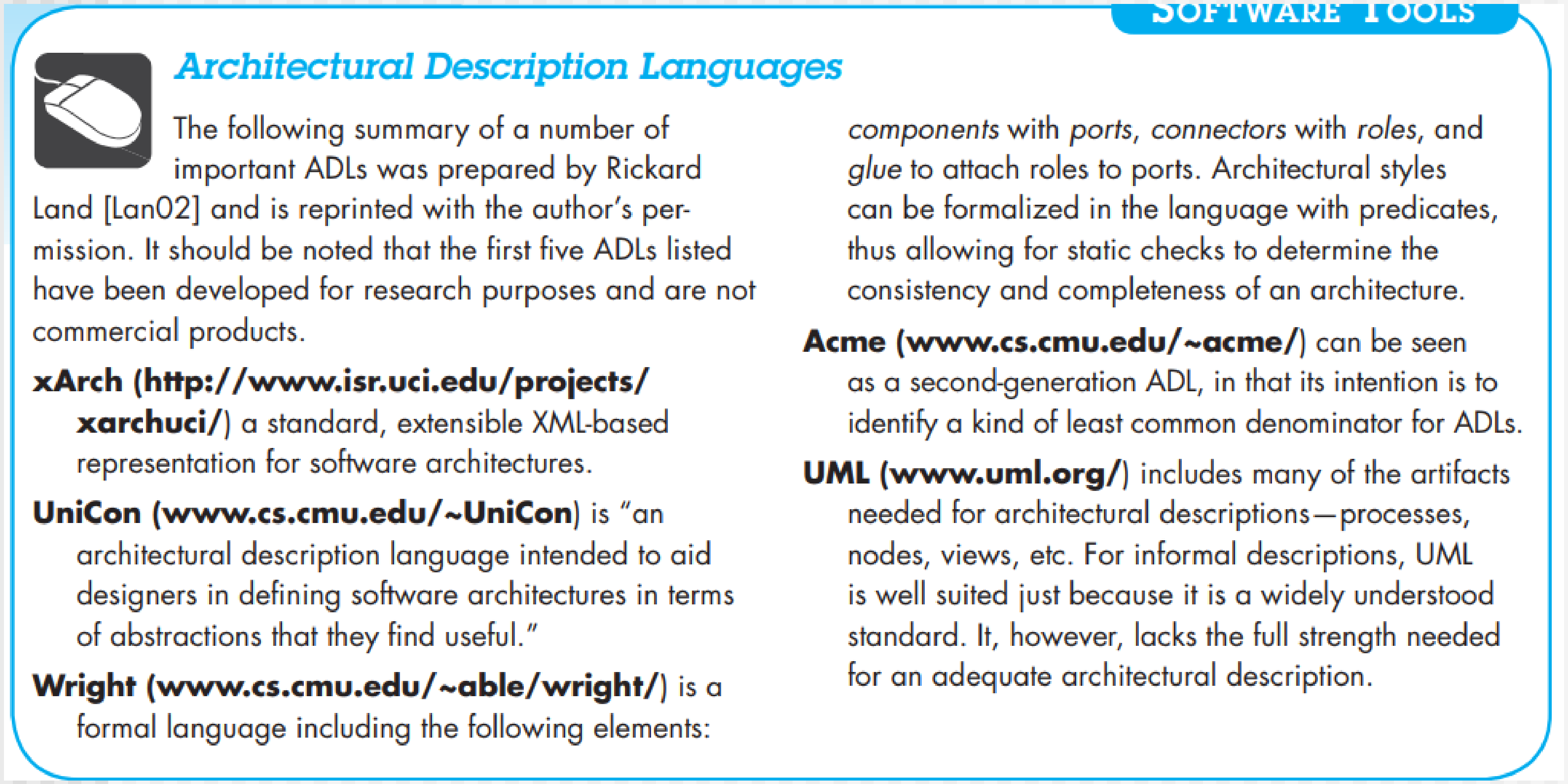
Architectural Considerations

- Economy – The best architecture is uncluttered and relies on a proper abstraction level to reduce unnecessary detail
- Visibility – Architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time
- Spacing – Separation of concerns in a design without introducing hidden dependencies
  - Too much spacing leads to fragmentation
  - Domain-driven design can help to identify what to separate in a design and what to treat as a coherent unit
- Symmetry – Architectural symmetry implies that a system is consistent and balanced in its attributes
  - Example: a *customer account* object with both *open()* and *close()* methods
- Emergence – Emergent, self-organized behavior and control

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.

Some Important ADLs



SOFTWARE TOOLS

Architectural Description Languages

The following summary of a number of important ADLs was prepared by Rickard Land [Lan02] and is reprinted with the author's permission. It should be noted that the first five ADLs listed have been developed for research purposes and are not commercial products.

xArch (http://www.isr.uci.edu/projects/xarchuci/) a standard, extensible XML-based representation for software architectures.

UniCon (www.cs.cmu.edu/~UniCon) is "an architectural description language intended to aid designers in defining software architectures in terms of abstractions that they find useful."

Wright (www.cs.cmu.edu/~able/wright/) is a formal language including the following elements: *components* with *ports*, *connectors* with *roles*, and *glue* to attach roles to ports. Architectural styles can be formalized in the language with predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

Acme (www.cs.cmu.edu/~acme/) can be seen as a second-generation ADL, in that its intention is to identify a kind of least common denominator for ADLs.

UML (www.uml.org/) includes many of the artifacts needed for architectural descriptions—processes, nodes, views, etc. For informal descriptions, UML is well suited just because it is a widely understood standard. It, however, lacks the full strength needed for an adequate architectural description.

Architecture Reviews

- A type of specialized **technical review**
- Assess the ability of the software architecture to meet the system’s quality requirements (e.g., scalability or performance) and identify potential risks
- Involve only **software engineering team** members supplemented by **independent experts**
- Have the potential to reduce project costs by detecting design problems early
- The most common architectural review techniques:
  - experience-based reasoning
  - prototype evaluation
  - scenario reviews
  - checklists

Pattern-Based Architecture Review

- A **lightweight** architectural review process known as PBAR
  - The best option in situations in which short build cycles, tight deadlines, volatile requirements, and/or small teams are the norm
- A face-to-face audit meeting involving architecture experts
  - Scheduled after the first working prototype or a baseline architecture is completed
- PBARs are well-suited to small, agile teams and require a relatively small amount of extra project time and effort

PBAR Steps

1. Identify and discuss the most important quality attributes by walking through the relevant use cases.
2. Discuss a diagram of system's architecture in relation to its requirements.
3. Identify the architecture patterns used and match the system's structure to the patterns' structure.
4. Use existing documentation and use cases to examine each pattern's effect on quality attributes.
5. Identify and discuss all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of issues uncovered during the meeting and make revisions to the architecture.

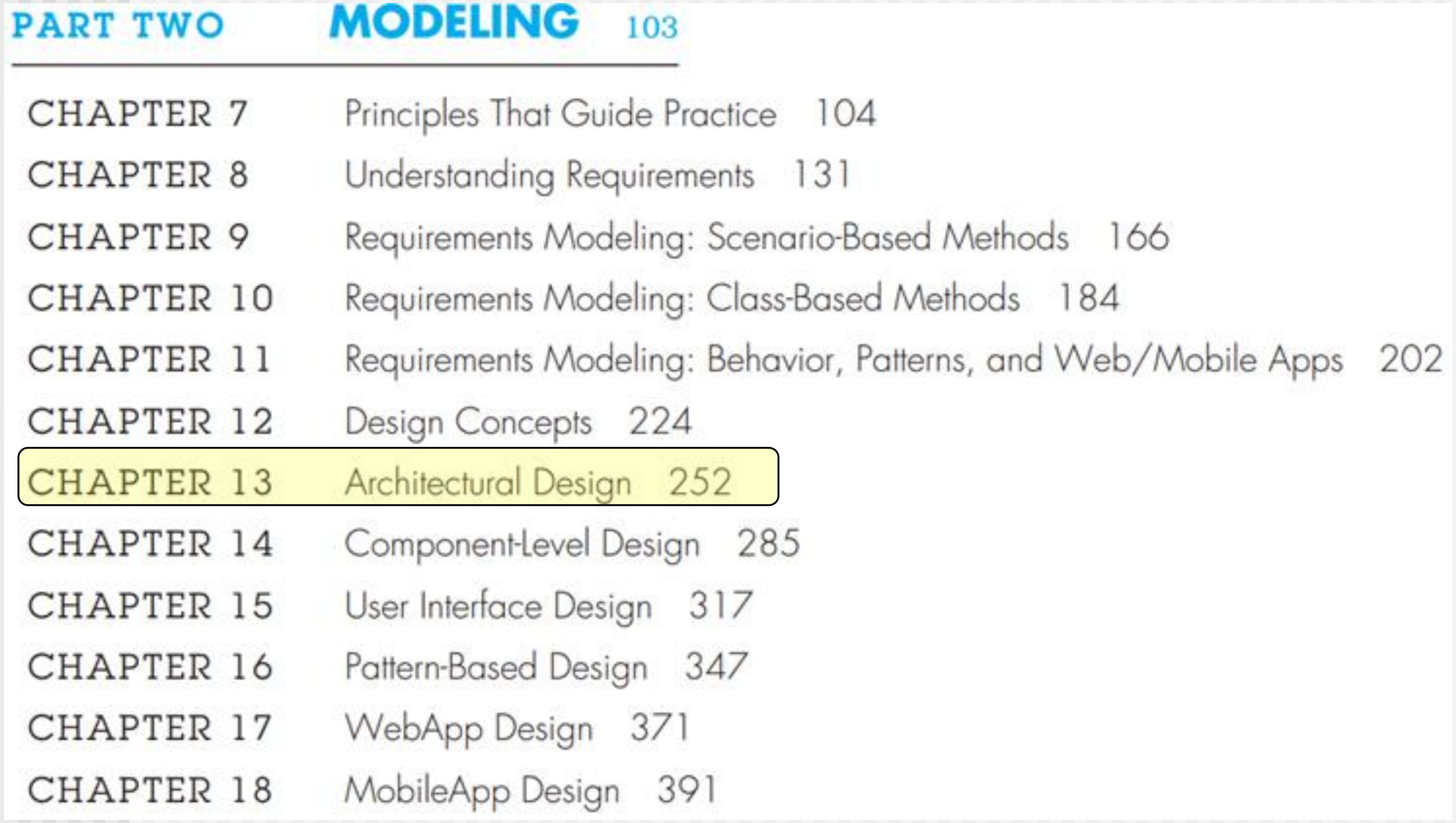
Agility and Architecture-I

- Most agile developers agree that it is important to focus on software architecture when a system is **complex**
  - i.e., a product has a large number of requirements, many stakeholders, or wide geographic distribution
  - When the wrong architecture is chosen, we encounter quality problems and so the rework is required
- To avoid rework, user stories are used to create and evolve an architectural model (**walking skeleton**) before coding

Agility and Architecture-II

- Software architects contributes **architectural user stories** to the evolving storyboard
- For example:
  - User story: “As an user I want my information to be available in all of my devices”
  - Architectural story: “Any UI element renders properly in desktop and mobile browsers (e.g., Firefox, Chrome, IE, as well as Apple and Android default mobile browsers)
- The architect works with
  - the product owner to prioritize the architectural stories
  - the team during the sprint to ensure that the evolving software continues to show high architectural quality
- Reviewing working product emerging from the sprint can be a useful form of architectural review

Further Reading



PART TWO **MODELING** 103

The End

Repository Architecture Style

- The data store is passive and

  the clients (software components) of the data store are active, which control the logic flow
- The participating components check the data-store for changes
- The client sends a request to the system to perform actions (e.g. insert data)
- This approach is widely used in DBMS, library information system, and CASE environments

Blackboard Architecture Style

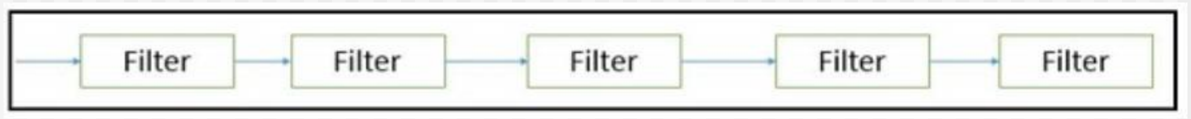
- The data store is active and its clients are passive
- Therefore the logical flow is determined by the data store
- It has a blackboard component, acting as a central data repository
- The data-store alerts the clients whenever there is a data-store change
- This approach is found in certain AI applications and complex applications, such as speech recognition, image recognition, security system, and business resource management systems etc.

Data Flow Architecture

- The data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output or a data store)
- The connections between the components may be implemented as I/O stream, I/O buffer, or other types of connections
- Suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications
- Two types of execution sequences between modules
  - Batch sequential
  - Pipe and filter or non-sequential pipeline mode

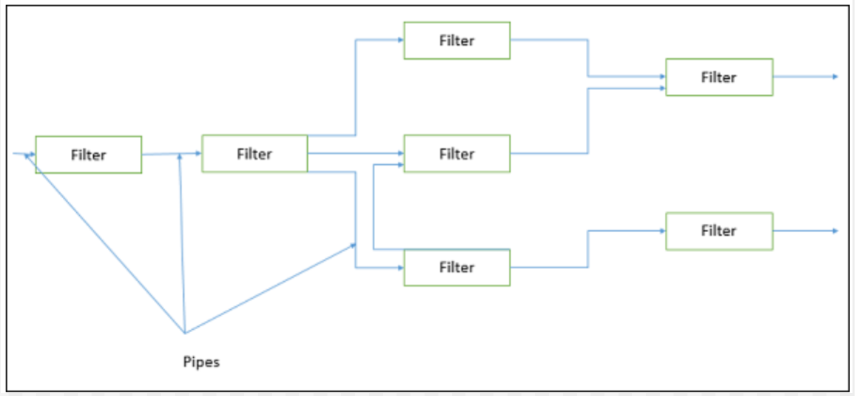
Batch Sequential Architecture

- A data transformation subsystem can initiate its process only after its previous subsystem is completely through
- A batch of data as a whole from one subsystem to another
- The communications between the modules are conducted through temporary intermediate files which can be removed by successive subsystems
- Typical application of this architecture includes business data processing such as banking



Pipe and Filter Architecture

- This approach lays emphasis on the incremental transformation of data by successive component
- The connections between modules are data stream which is first-in/first-out buffer
  - can be stream of bytes, characters, or any other type of such kind
- The main feature of this architecture is its concurrent and incremented execution
- Example of pipe and filter architectural style can be found in compilers
  - consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation


Filter
Filter
Filter
Filter
Filter
Filter
Filter
Pipes