Chapter 12

Design Concepts

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Design

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

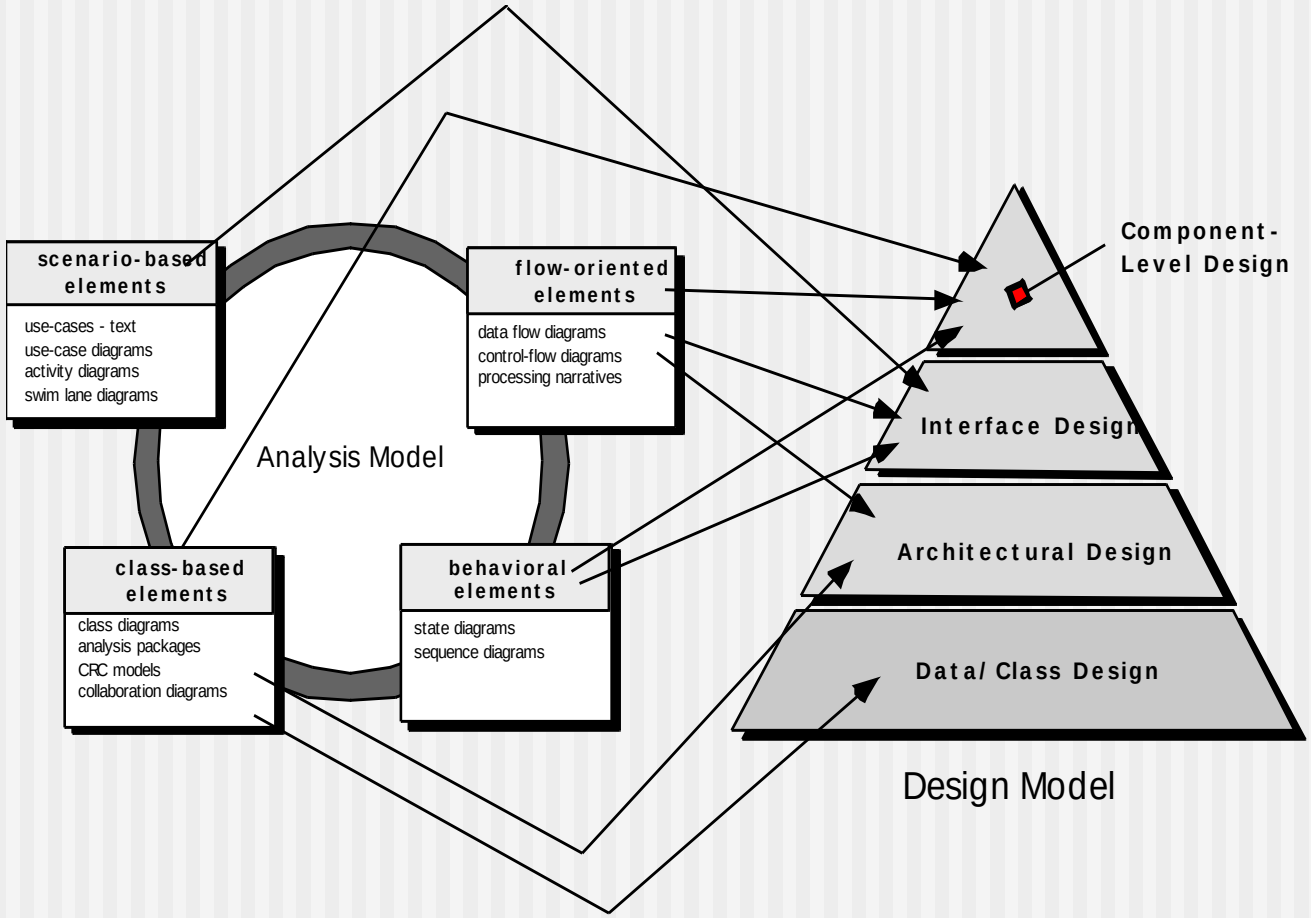
Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish and overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Software Engineering Design

- Data/Class design – transforms analysis classes into implementation classes and data structures
- Architectural design – defines relationships among the major software structural elements
- Interface design – defines how software elements, hardware elements, and end-users communicate
- Component-level design – transforms structural elements into procedural descriptions of software components

Analysis Model -> Design Model



Design and Quality

- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

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

Design Principles

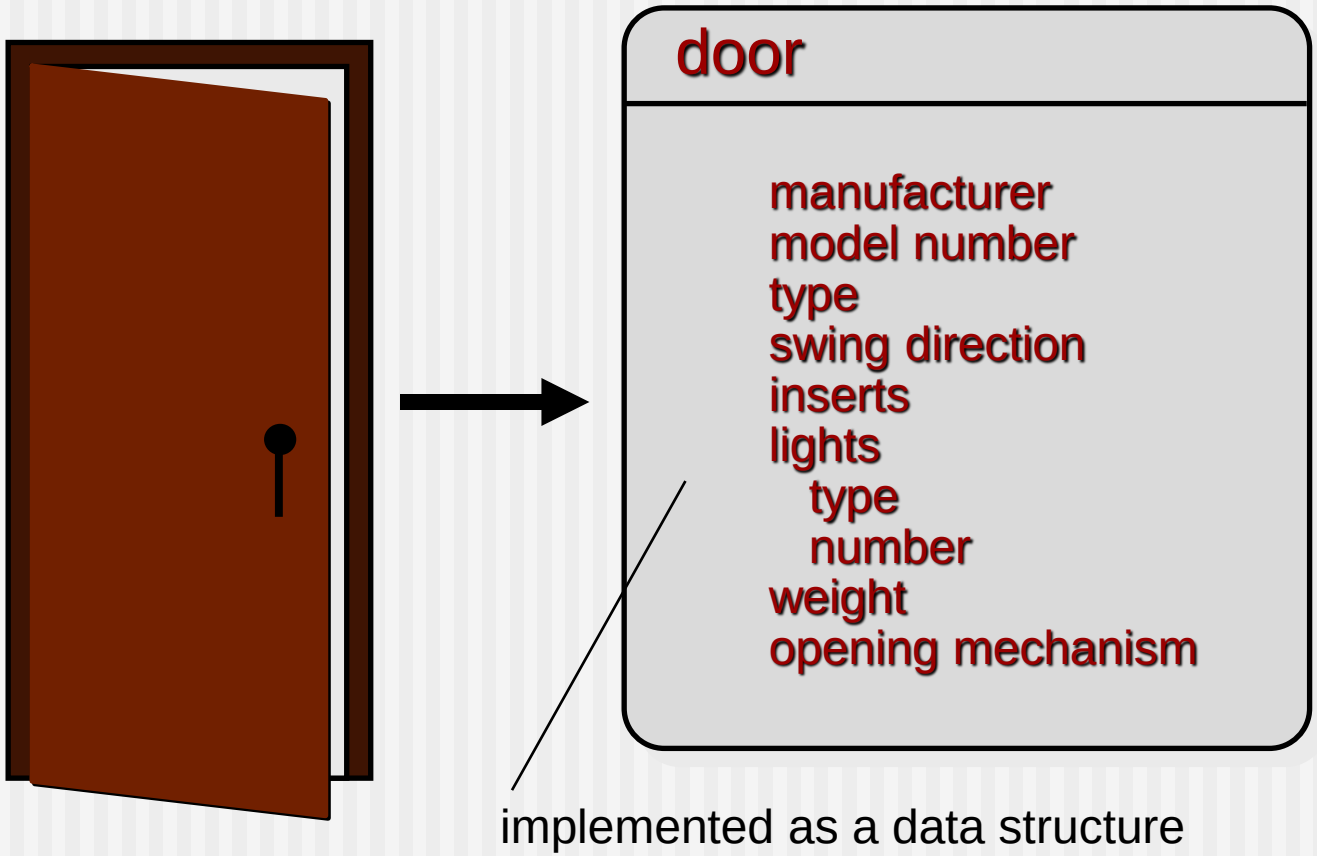
- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

From Davis [DAV95]

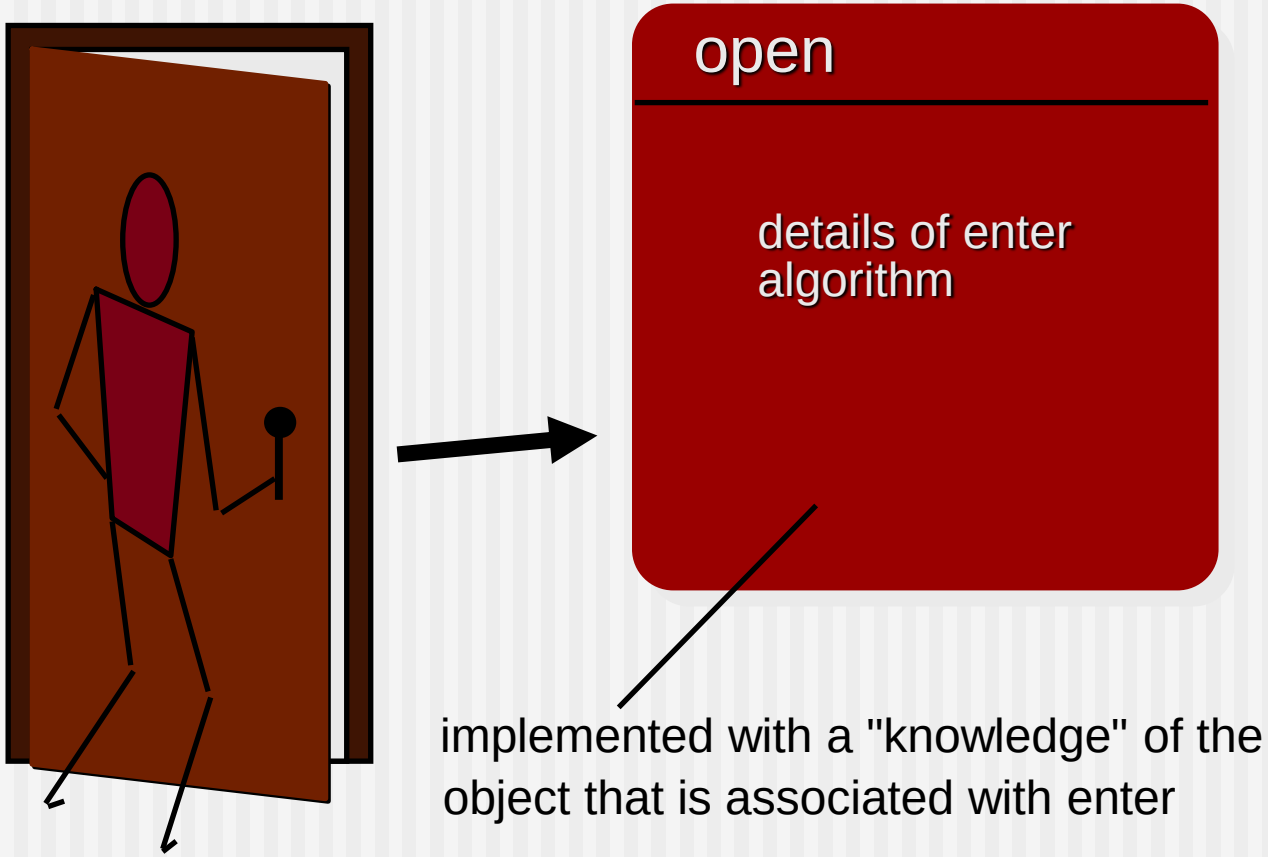
Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

Data Abstraction



Procedural Abstraction



Architecture

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system." [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

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

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

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

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

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

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

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

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

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

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

Separation of Concerns

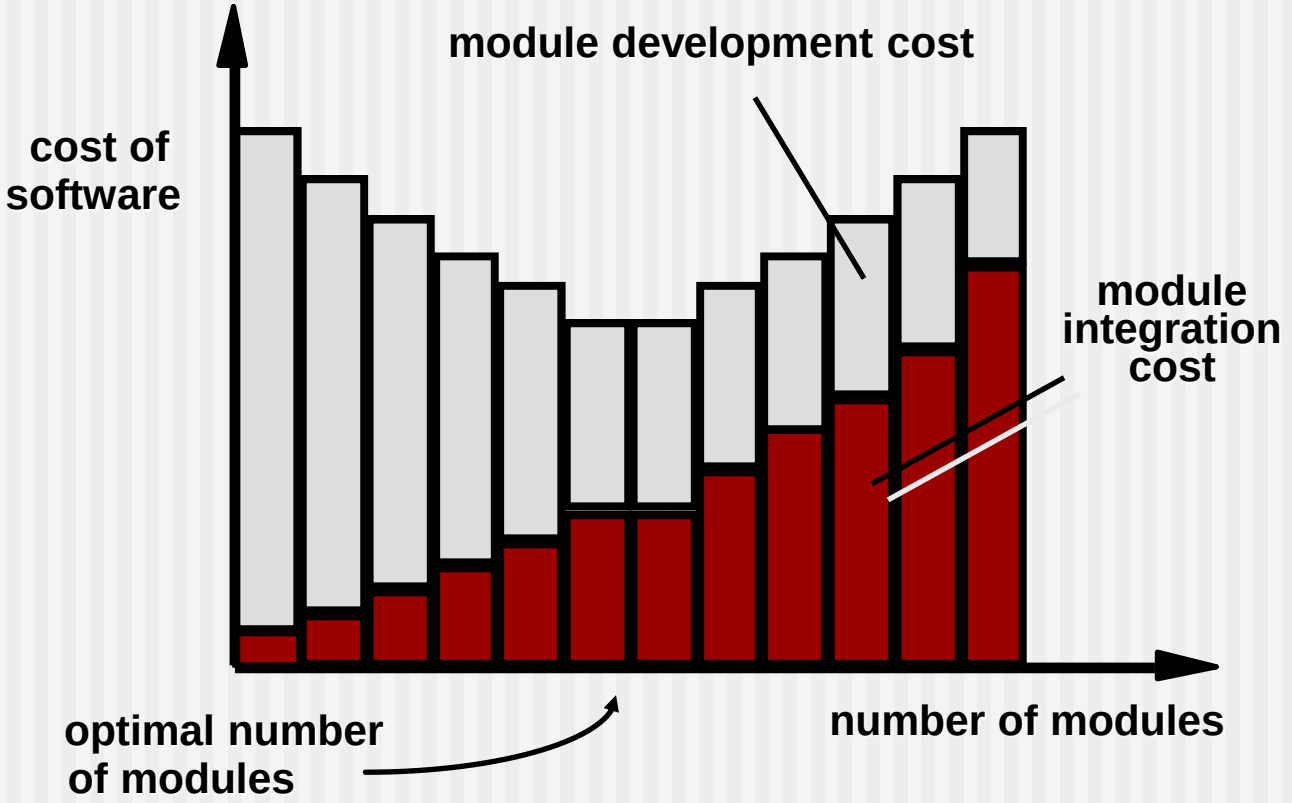
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

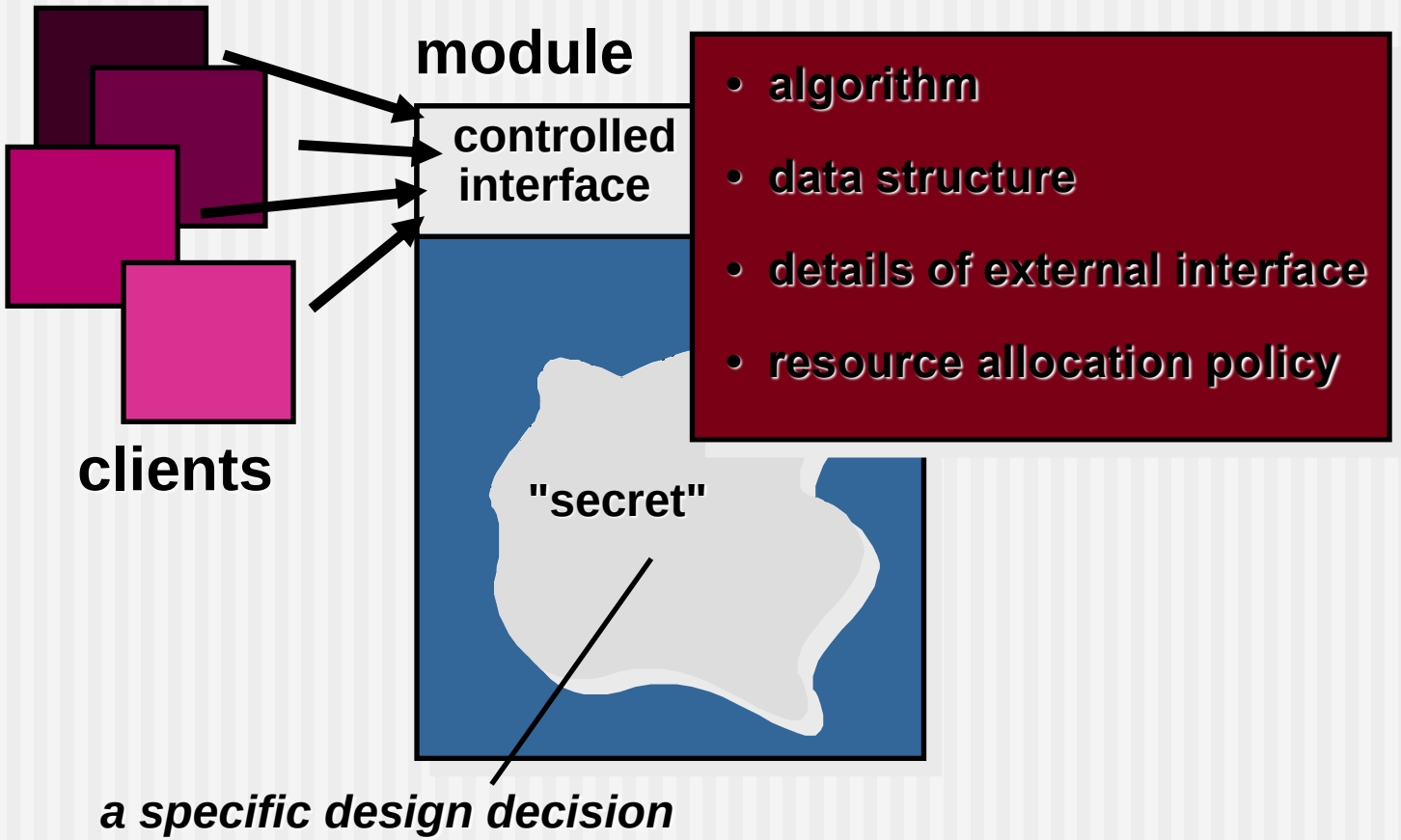
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



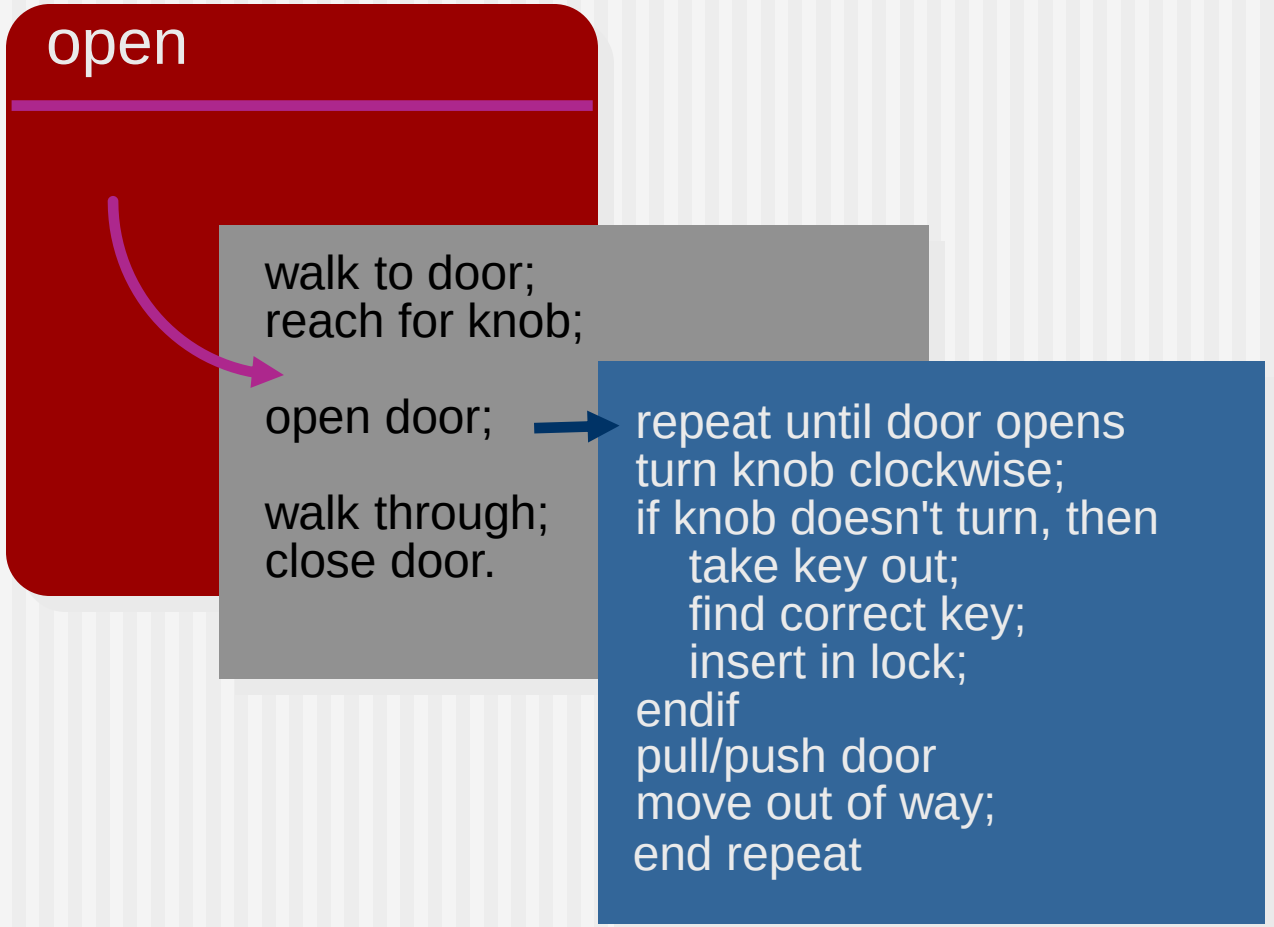
Information Hiding



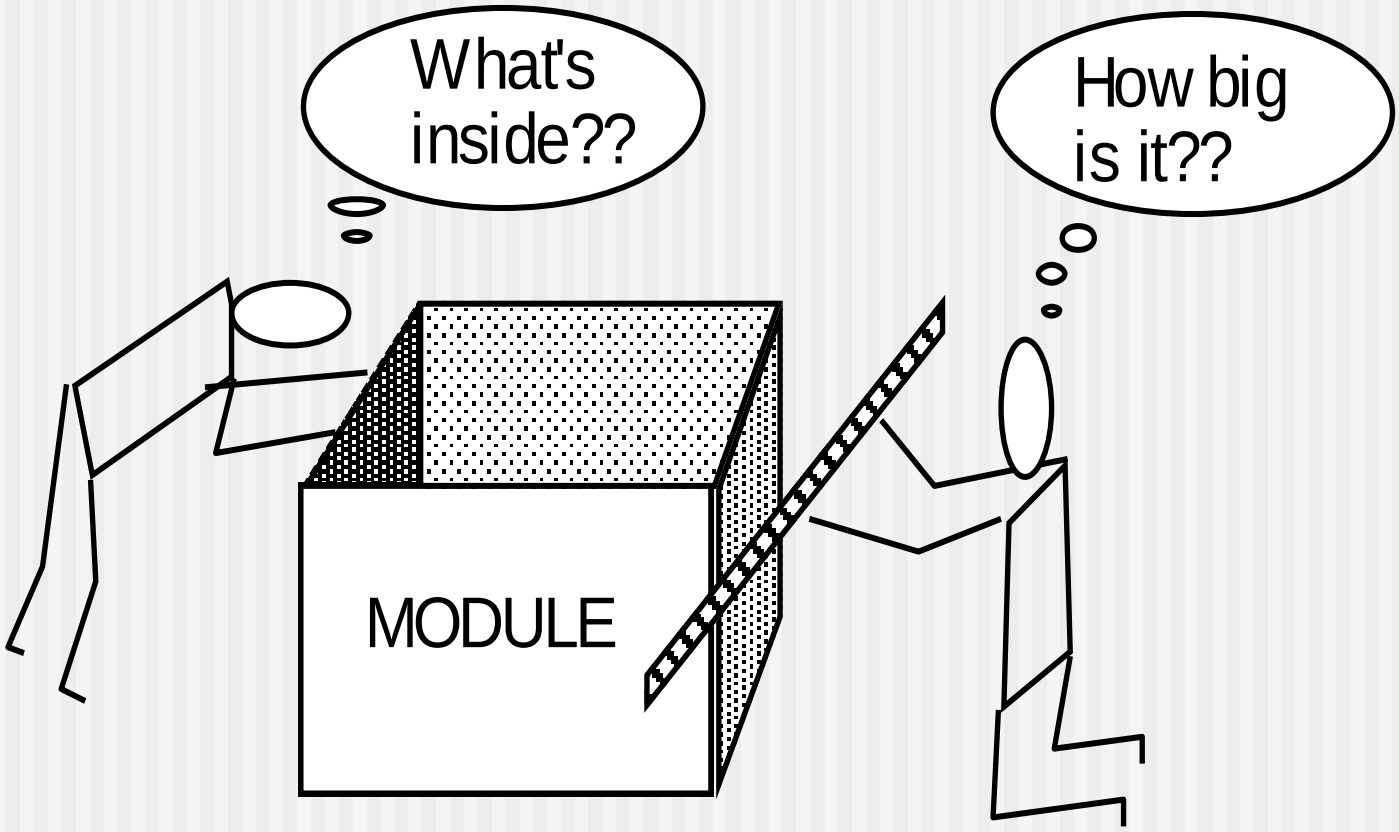
Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Stepwise Refinement



Sizing Modules: Two Views



Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

OO Design Concepts

- 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

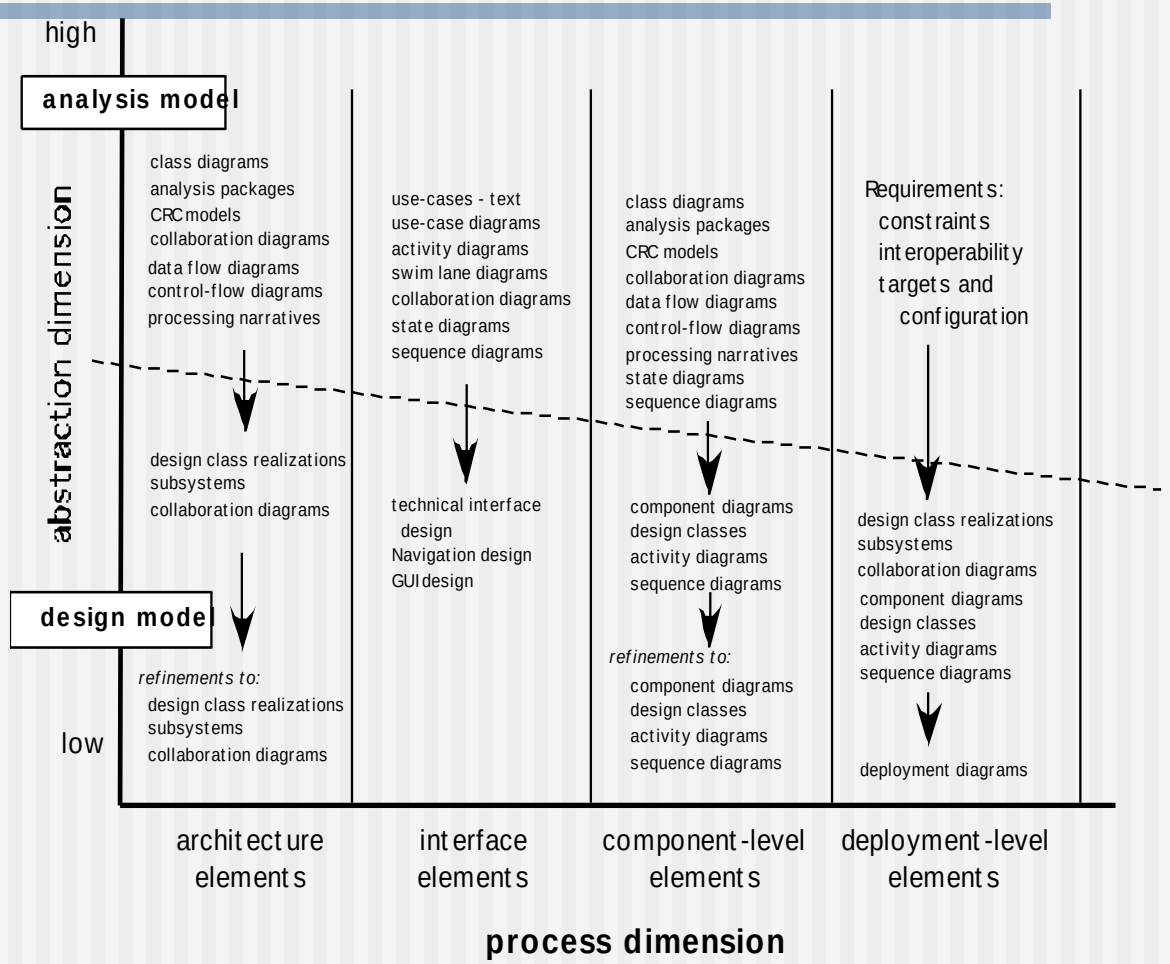
Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

Design Class Characteristics

- Complete - includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

The Design Model



Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (Chapters 9 and 12)
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

Data Modeling

- examines data objects independently of processing
- focuses attention on the data domain
- creates a model at the customer's level of abstraction
- indicates how data objects relate to one another

What is a Data Object?

- a representation of almost any composite information that must be understood by software.
  - *composite information*—something that has a number of different properties or attributes
- can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object

object: automobile

attributes:
- **make**
- **model**
- **body type**
- **price**
- **options code**

What is a Relationship?

- Data objects are connected to one another in different ways.
  - A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - A person *is insured to drive* a car
- The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.**
- Several instances of a relationship can exist
- Objects can be related in many different ways

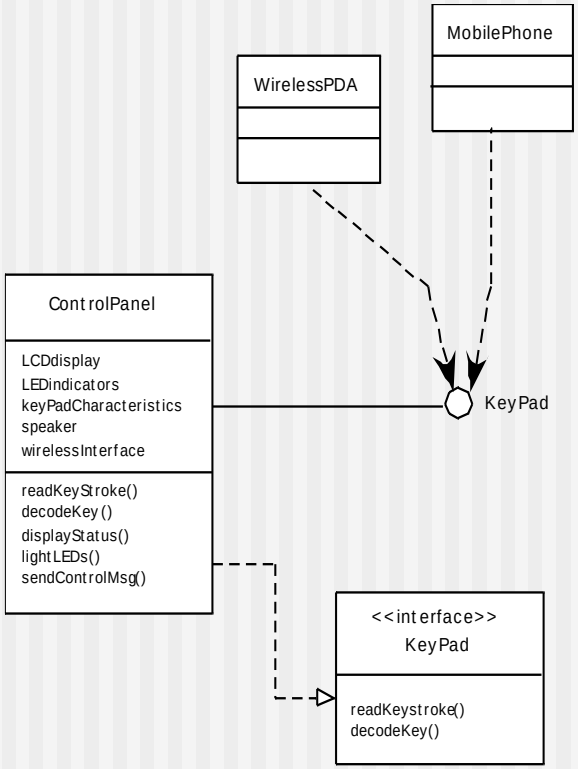
Architectural Elements

- The architectural model [Sha96] is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns (Chapter 16) and styles (Chapter 13).

Interface Elements

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaboration diagrams in UML 1.x)

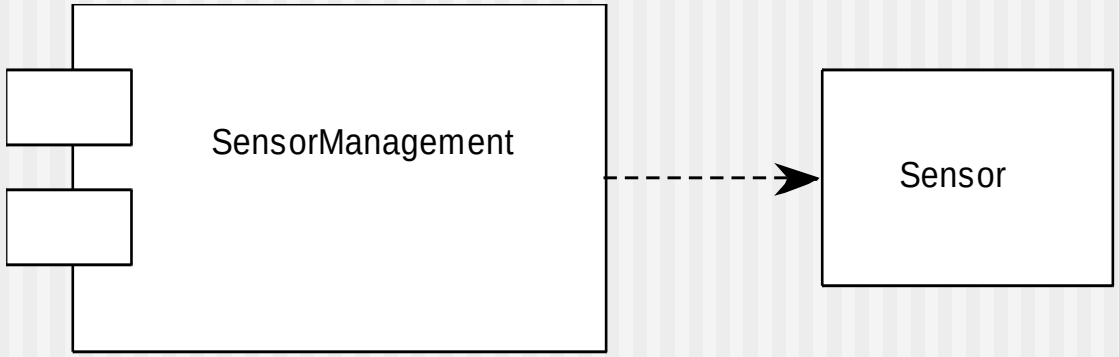
Interface Elements



Component Elements

- Describes the internal detail of each software component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations
- Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

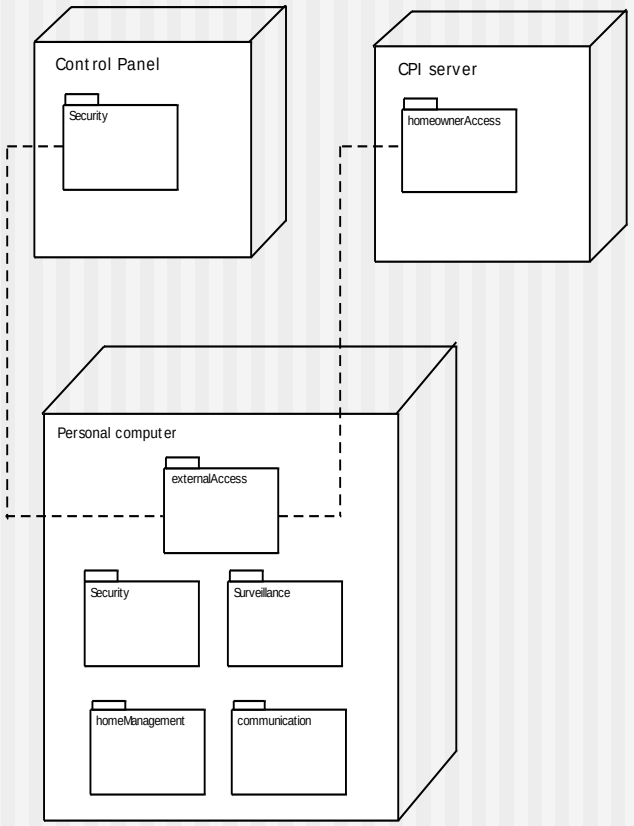
Component Elements



Deployment Elements

- Indicates how software functionality and subsystems will be allocated within the physical computing environment
- Modeled using UML deployment diagrams
- *Descriptor form* deployment diagrams show the computing environment but does not indicate configuration details
- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

Deployment Elements



Chapter 13

- **Architectural Design**

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Why Architecture?

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.

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

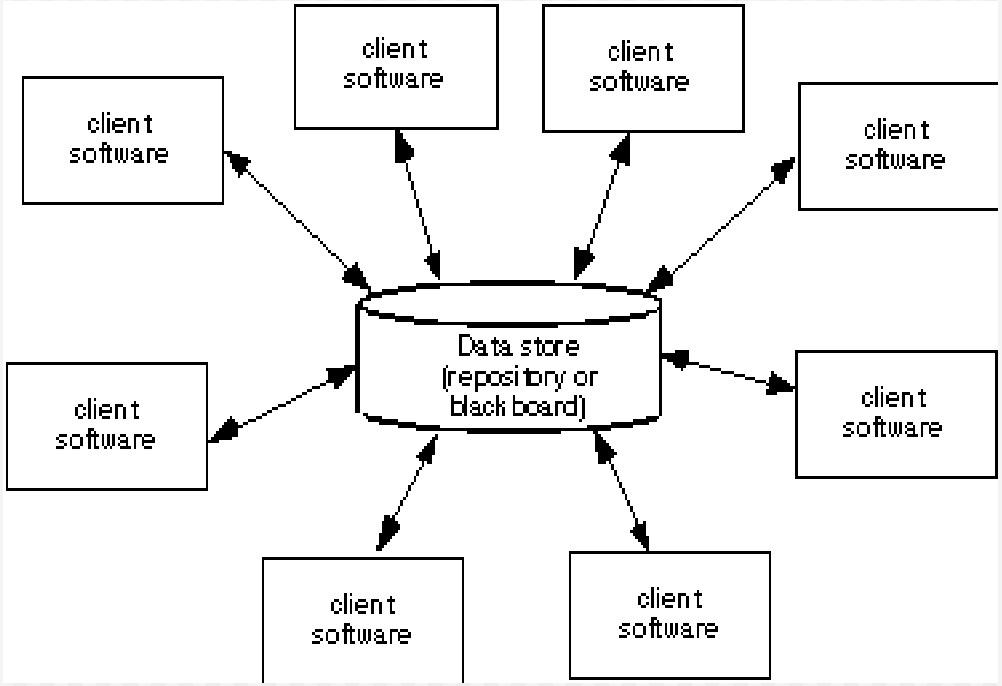
- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial 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

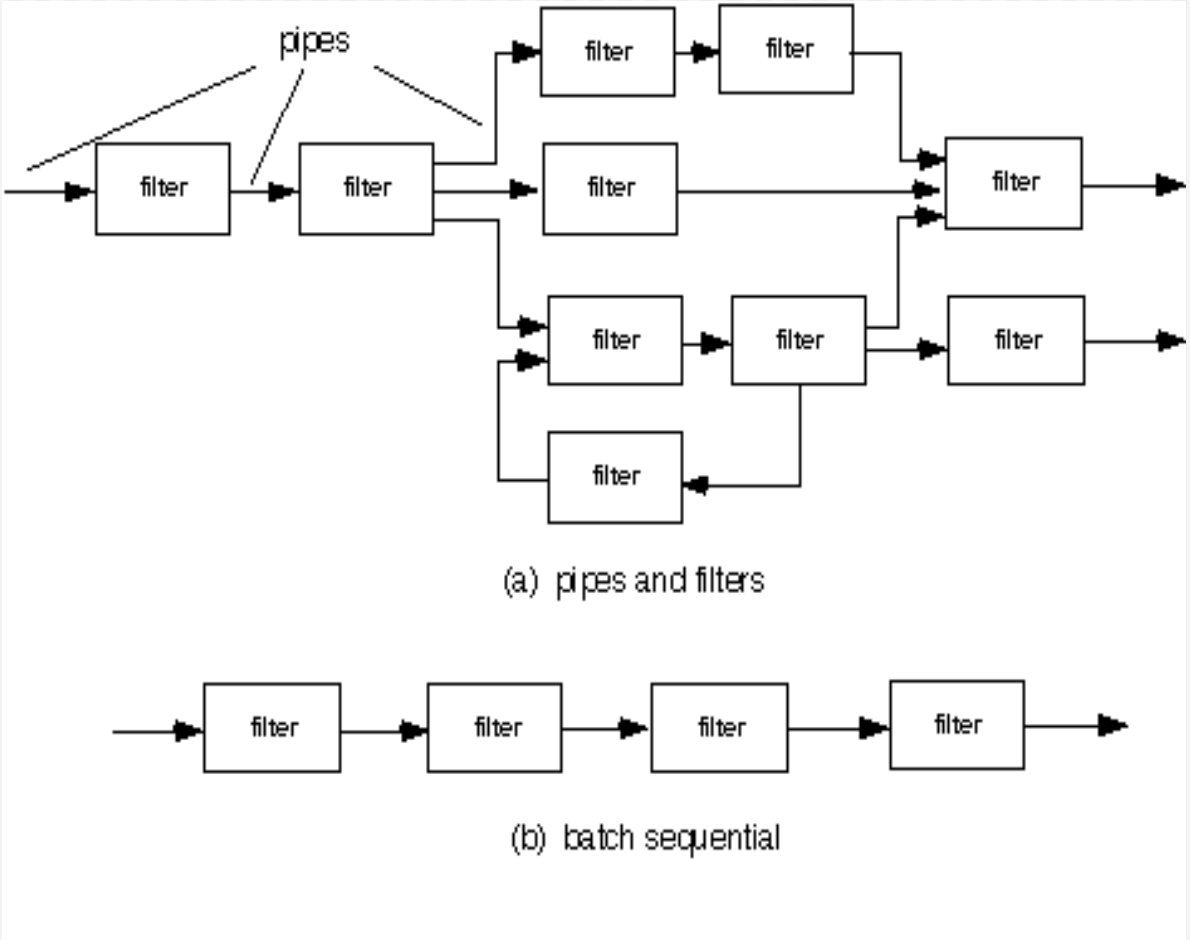
Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable "communication, coordination and cooperation" among components, (3) **constraints** that define how components can be integrated to form the system, and (4) **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

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

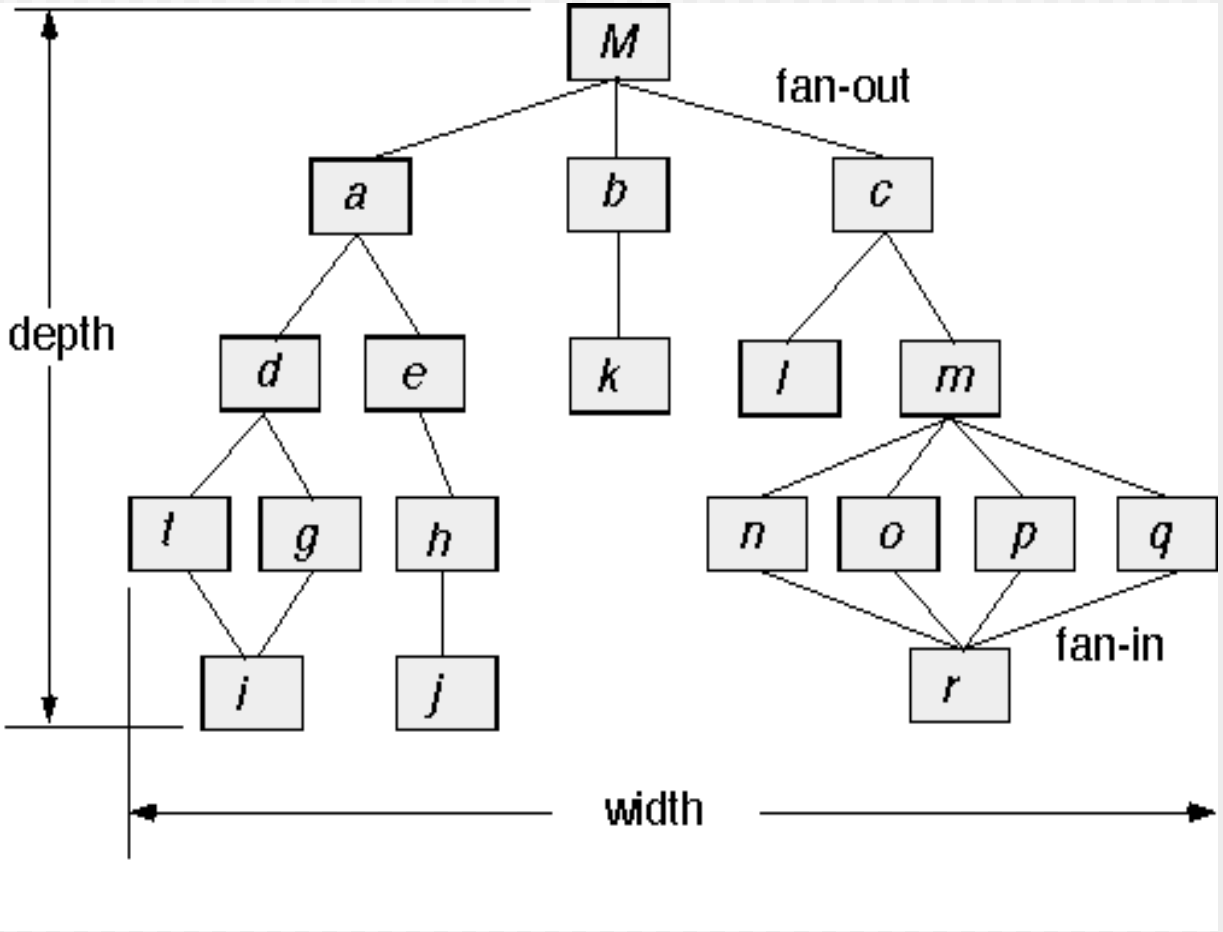
Data-Centered Architecture



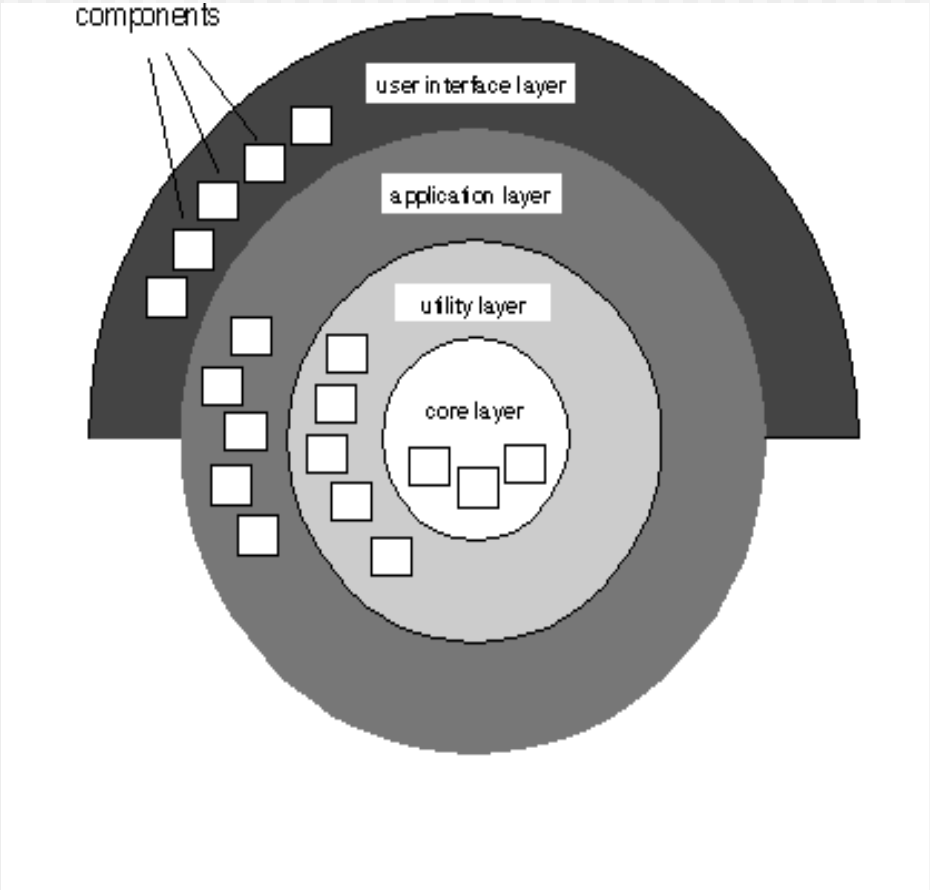
Data Flow Architecture



Call and Return Architecture



Layered Architecture



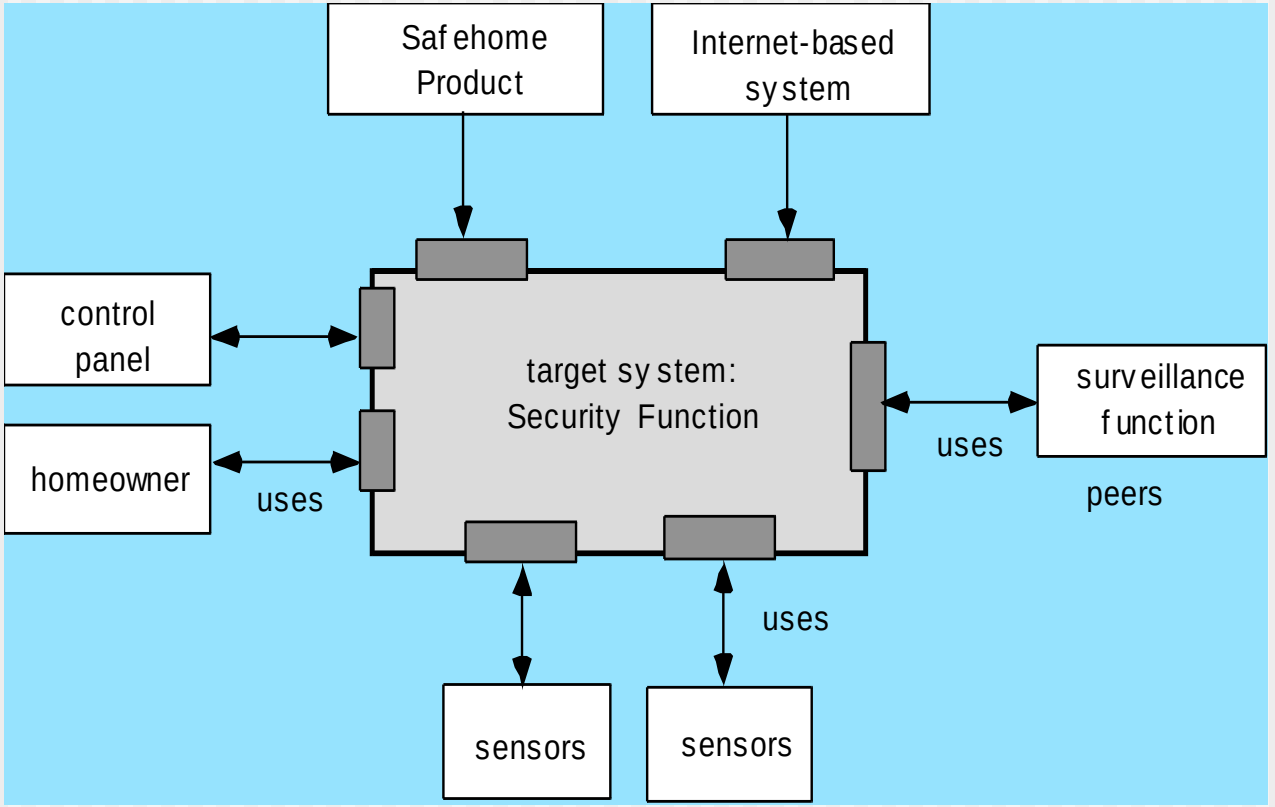
Architectural Patterns

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

Architectural Design

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

Architectural Context



Archetypes

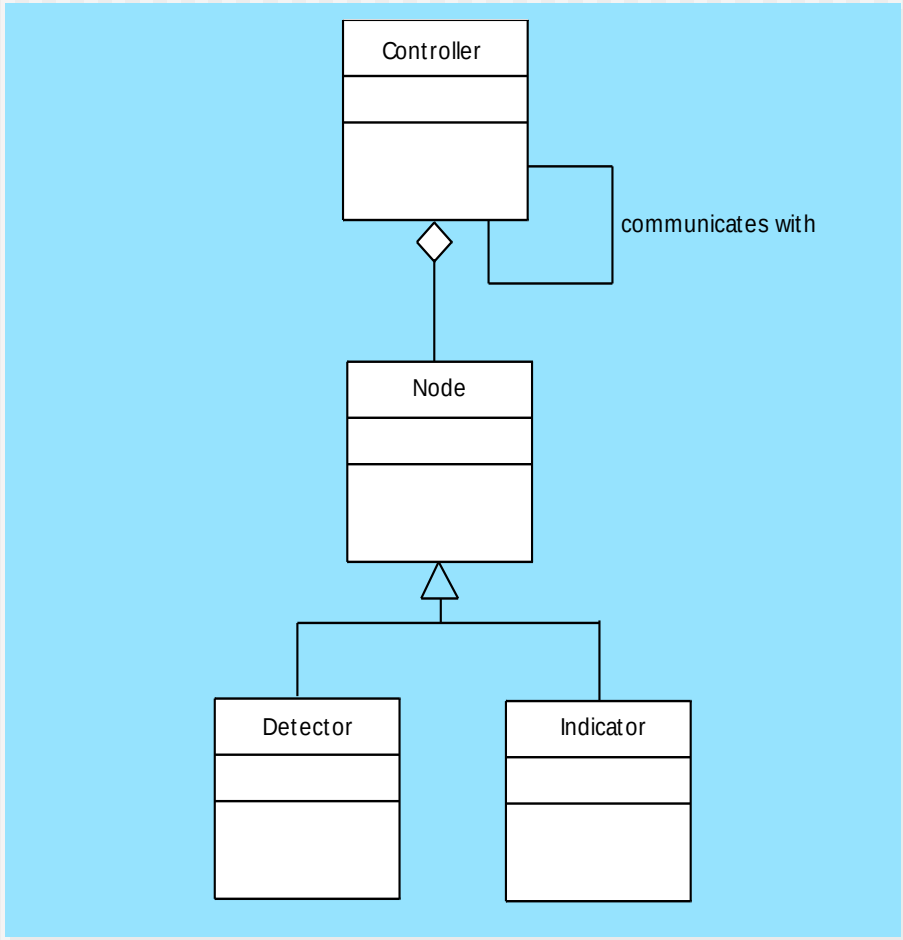
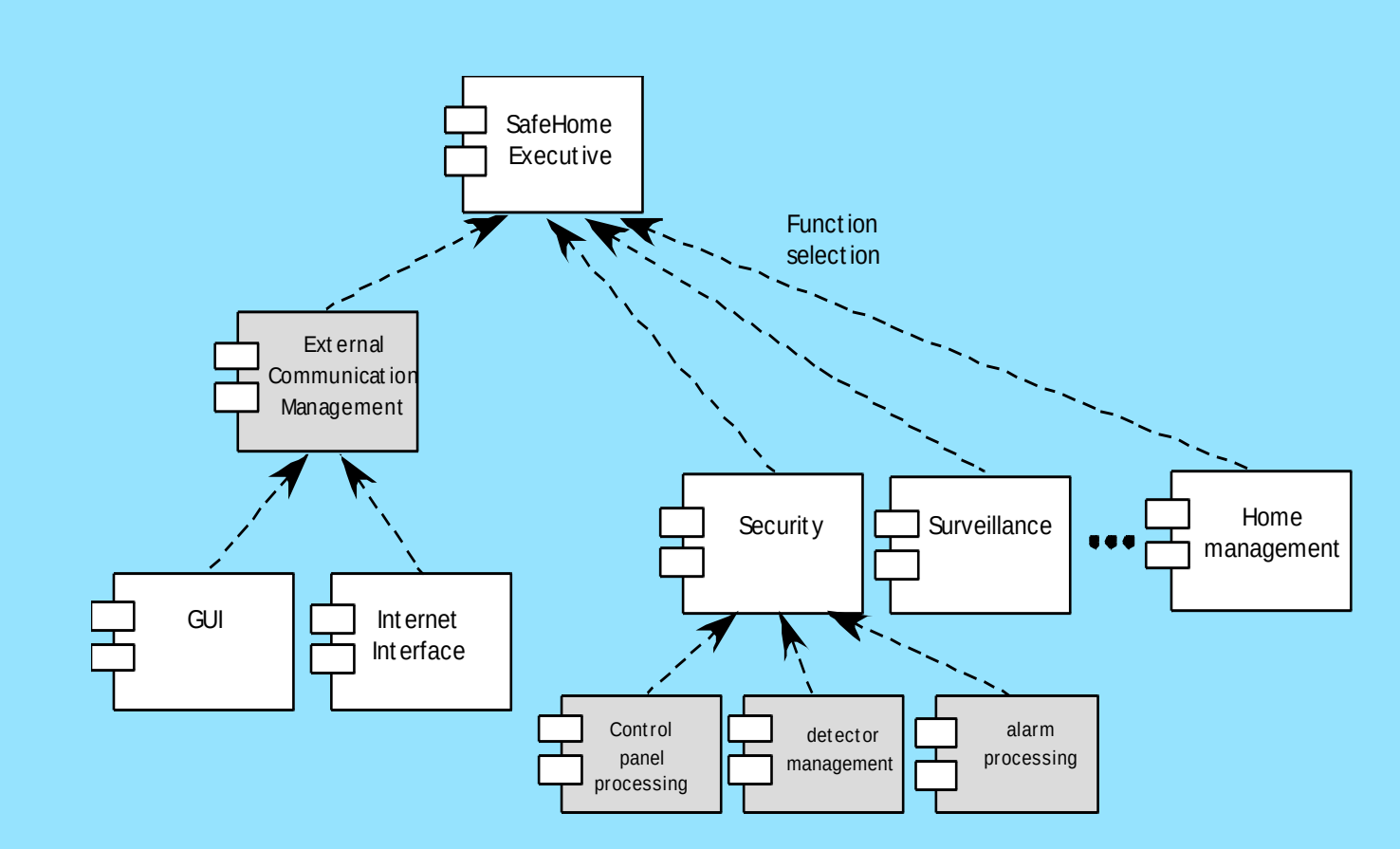
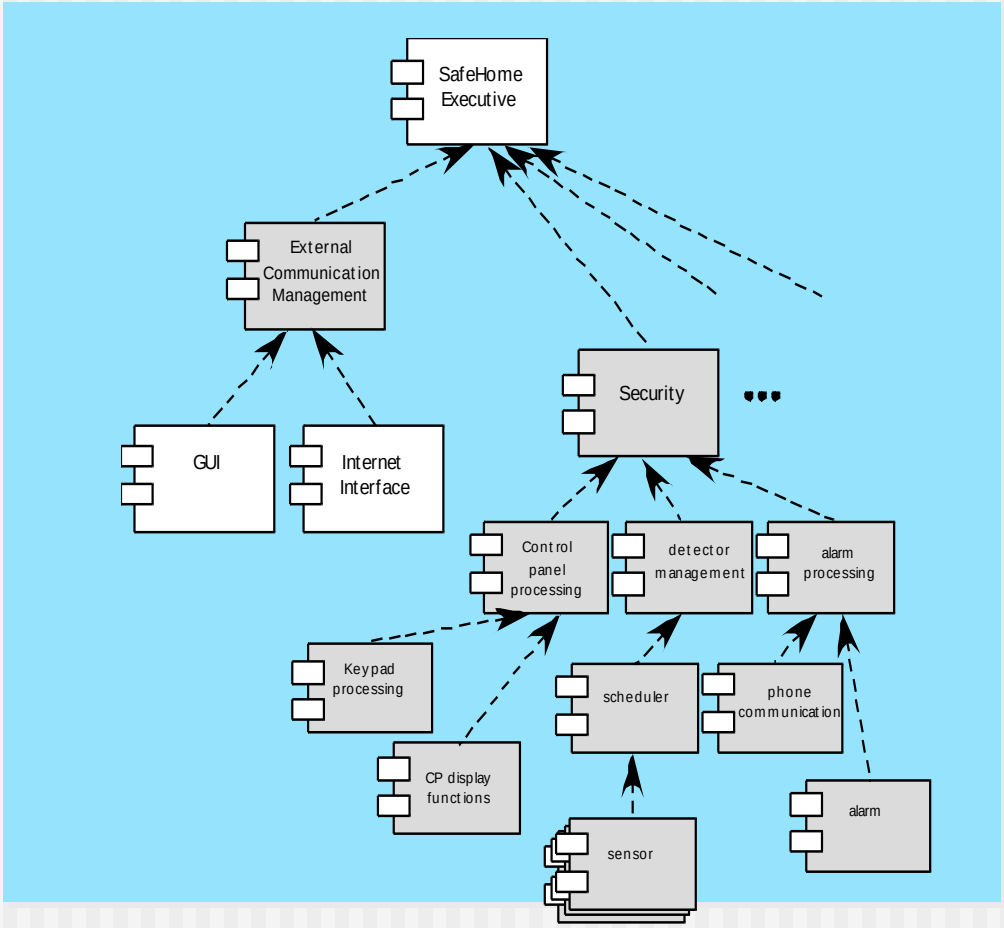


Figure 10.7 UML relationships for SafeHome archetypes

Component Structure



Refined Component Structure



Architectural Considerations

- Economy – The best software is uncluttered and relies on abstraction 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.
- Symmetry – Architectural symmetry implies that a system is consistent and balanced in its attributes.
- Emergence – Emergent, self-organized behavior and control.

Architectural Decision Documentation

1. Determine which information items are needed for each decision.
2. Define links between each decision and appropriate requirements.
3. Provide mechanisms to change status when alternative decisions need to be evaluated.
4. Define prerequisite relationships among decisions to support traceability.
5. Link significant decisions to architectural views resulting from decisions.
6. Document and communicate all decisions as they are made.

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
    - process view
    - data flow view
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

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.

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.

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

Patter-Based Architecture Review

1. Identify and discuss the quality attributes by walking through the 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 determine each pattern's effect on quality attributes.
5. Identify 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 walking skeleton.

Agility and Architecture

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

Chapter 14

- **Component-Level Design**

Slide Set to accompany

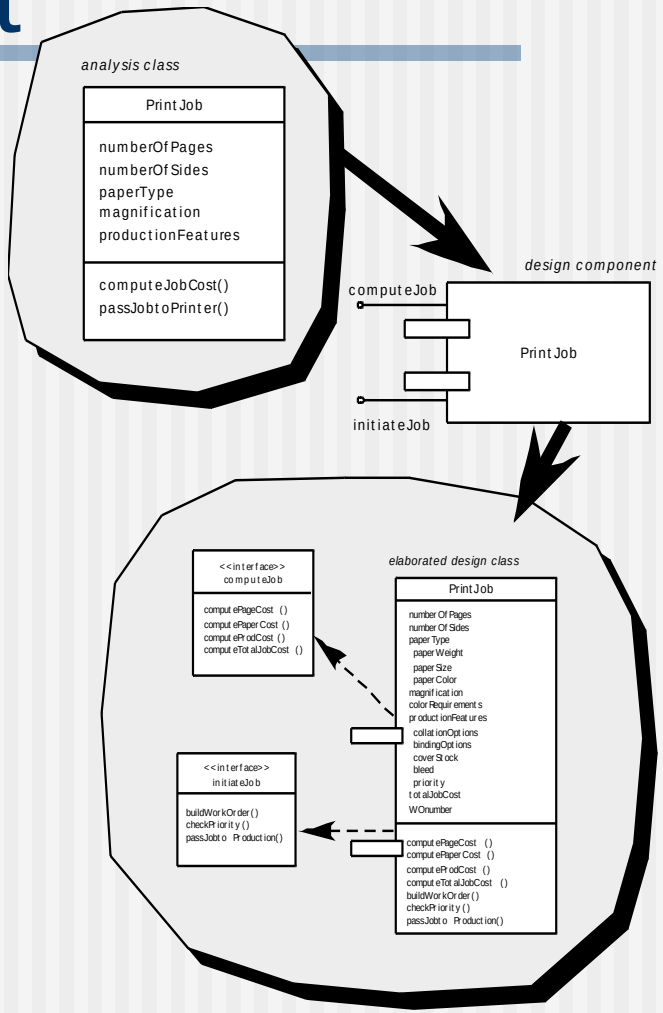
Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

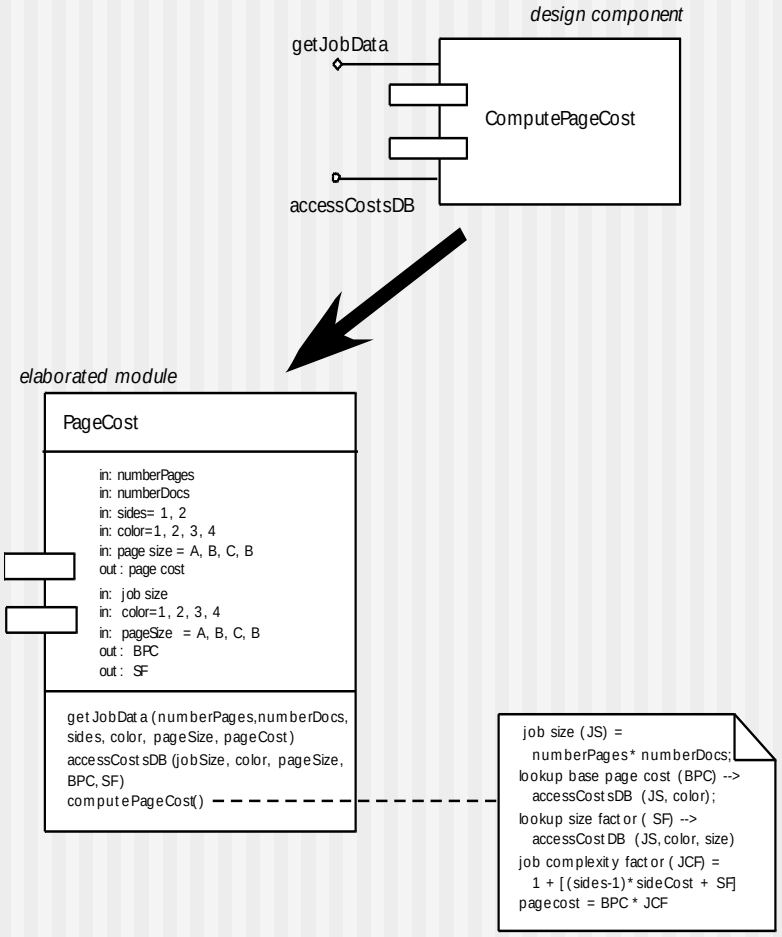
What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- *OO 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 and data to be passed to it.

OO Component



Conventional Component



Basic Design Principles

- The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification.*
- 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 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."*

Source: Martin, R., "Design Principles and Design Patterns," downloaded from http:www.objectmentor.com, 2000.

Design Guidelines

- Components
  - Naming conventions 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
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- 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

- Conventional view:
  - the “single-mindedness” of a module
- OO 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

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
  - Data
  - Routine call
  - Type use
  - Inclusion or import
  - External

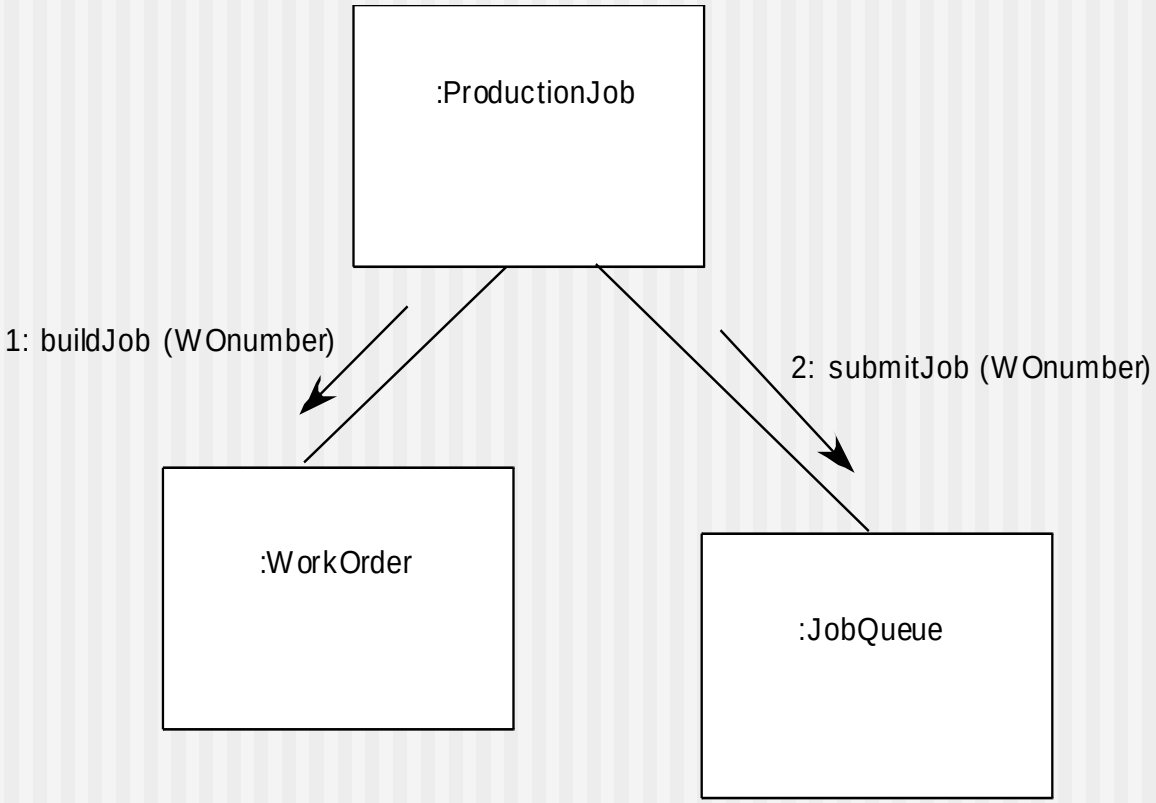
Component Level Design-I

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

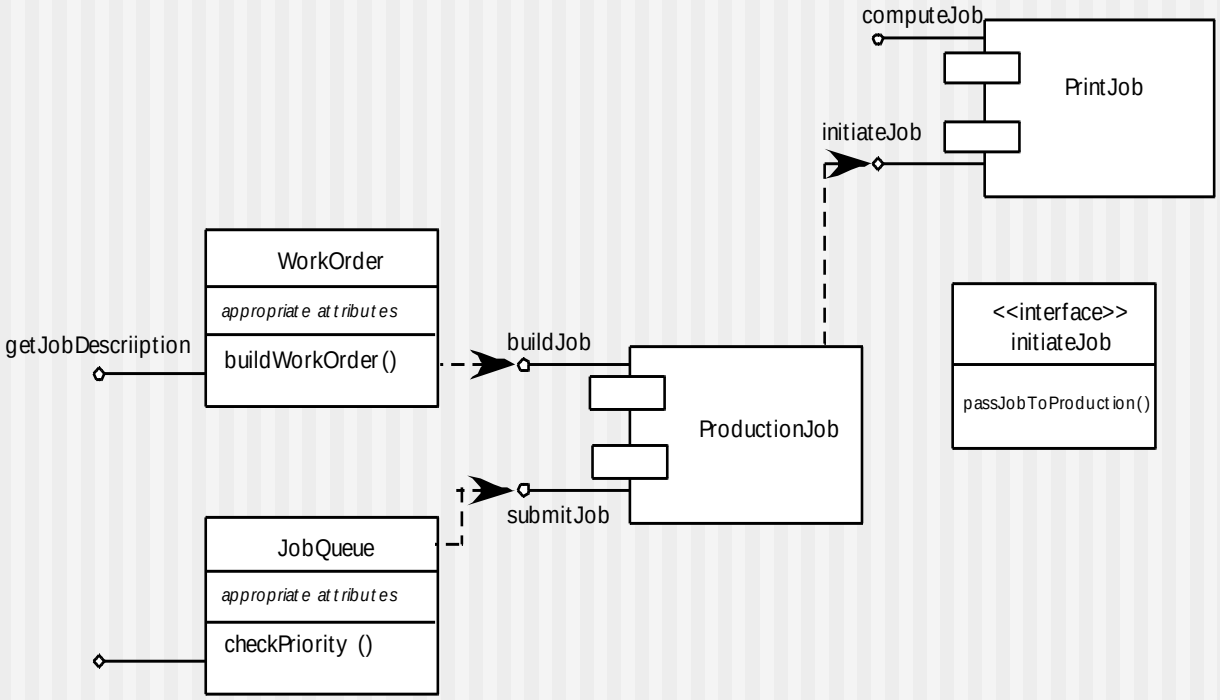
Component-Level Design-II

- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

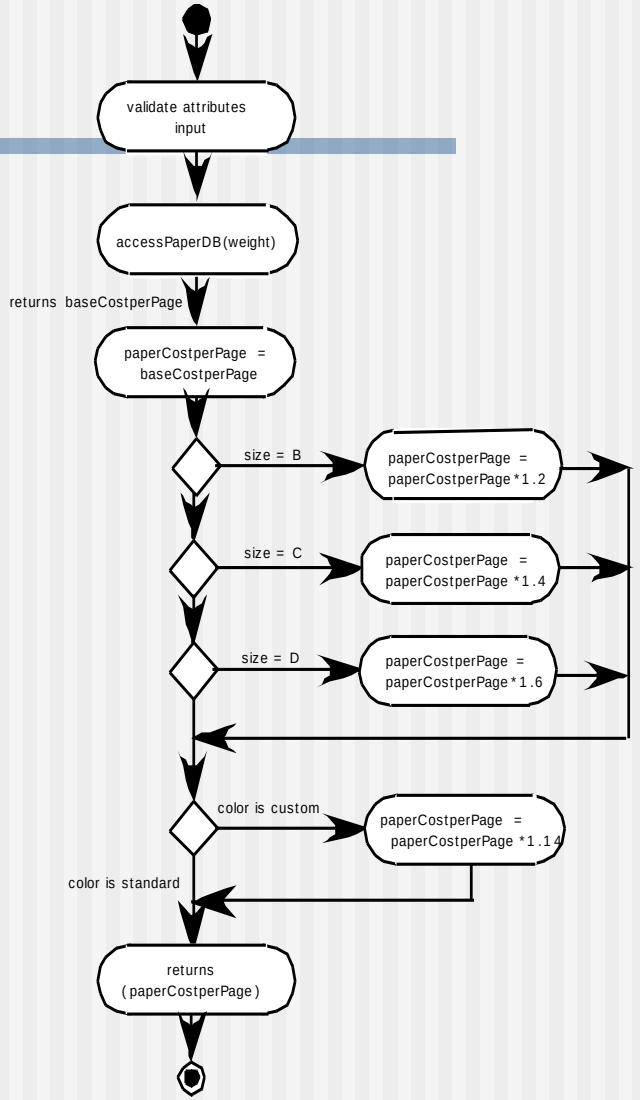
Collaboration Diagram



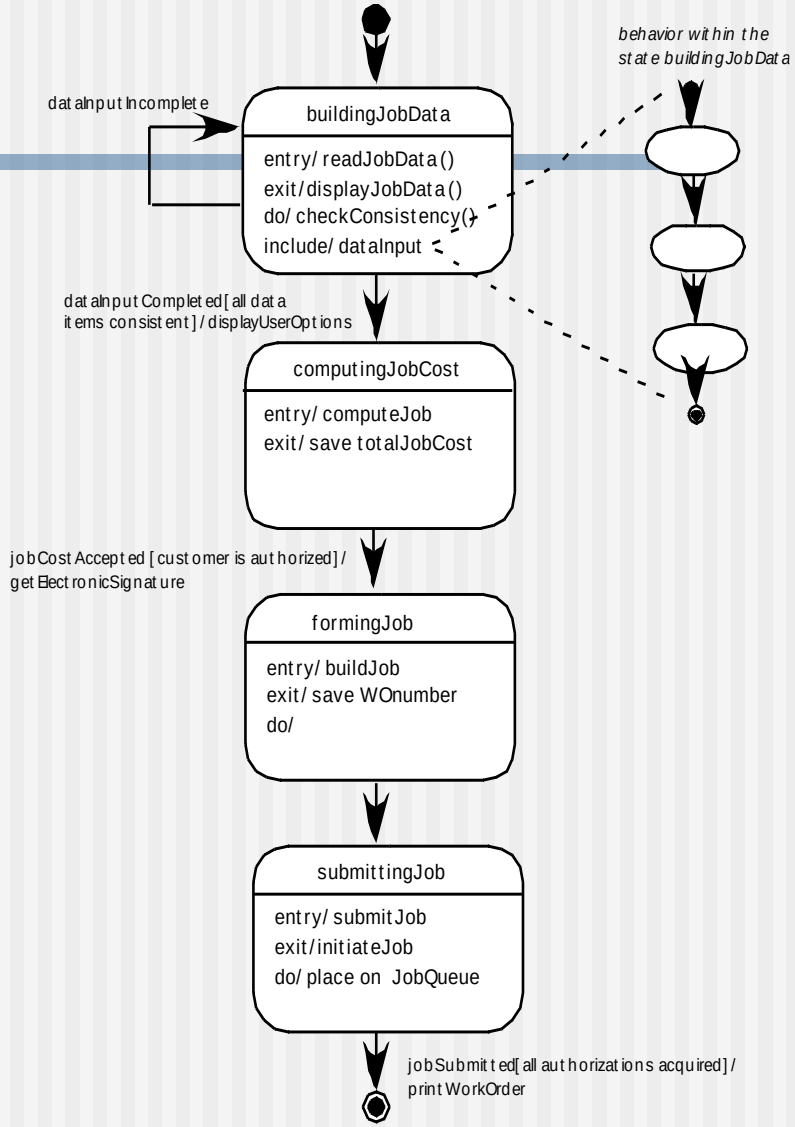
Refactoring



Activity Diagram



Statechart



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 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.

Component Design for Mobile Apps

- Thin web-based client
  - Interface layer only on device
  - Business and data layers implemented using web or cloud services
- Rich client
  - All three layers (interface, business, data) implemented on device
  - Subject to mobile device limitations

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

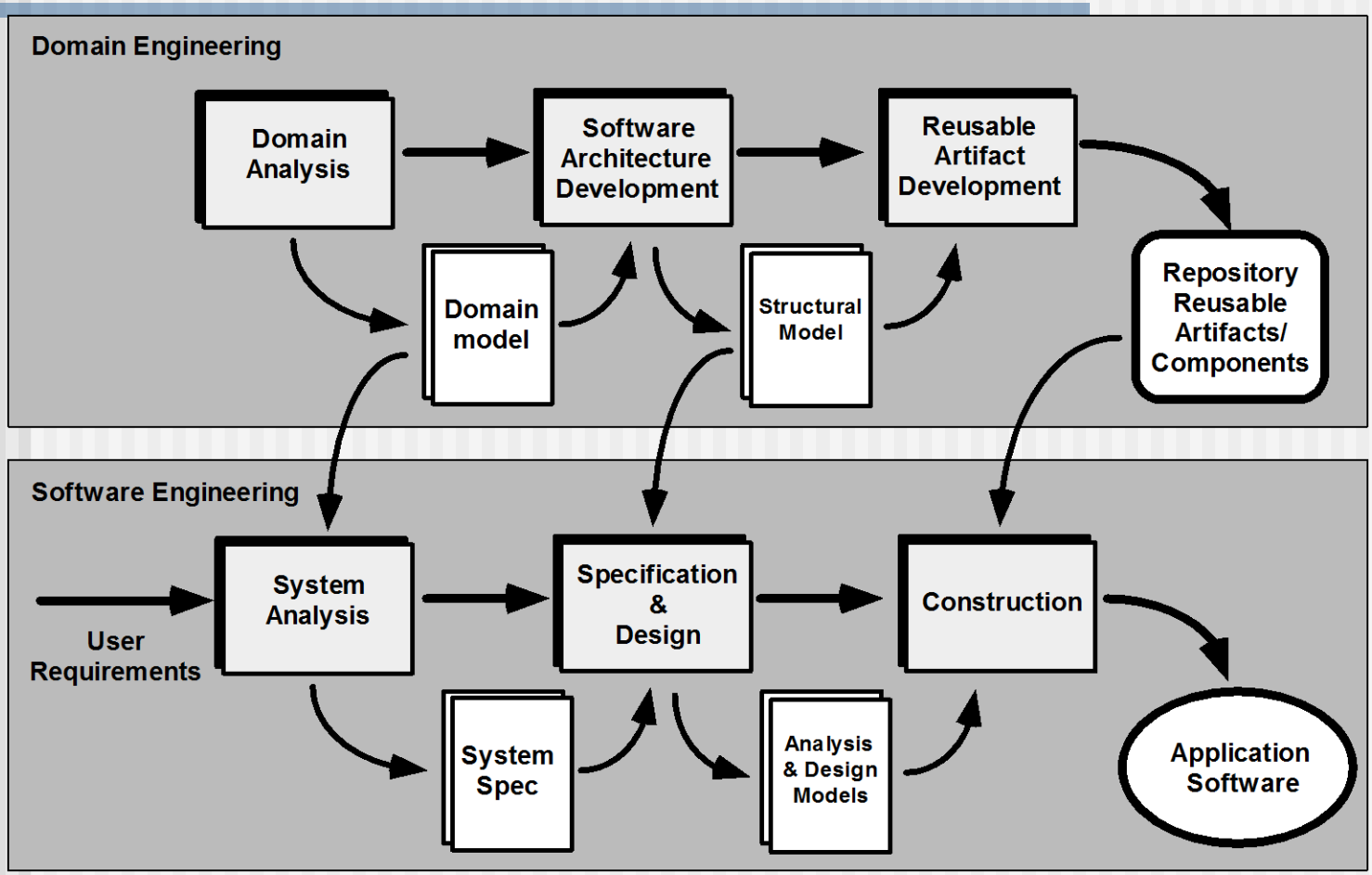
Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”
- Many companies continue to encourage of software development methodologies which do not facilitate reuse
- Few companies provide an incentives to produce reusable program components.

The CBSE Process



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.

Identifying Reusable Components

- Is component functionality required on future implementations?
- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware-dependent?
- Does the hardware remain unchanged between implementations?
- Can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a non-reusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?
- Is reuse through modification feasible?
- Can a non-reusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

Component-Based SE

- 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

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

Qualification

Before a component can be used, you must consider:

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling

Adaptation

The implication of "easy integration" is:

(1) that consistent methods of resource management have been implemented for all components in the library;

(2) that common activities such as data management exist for all components, and

(3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

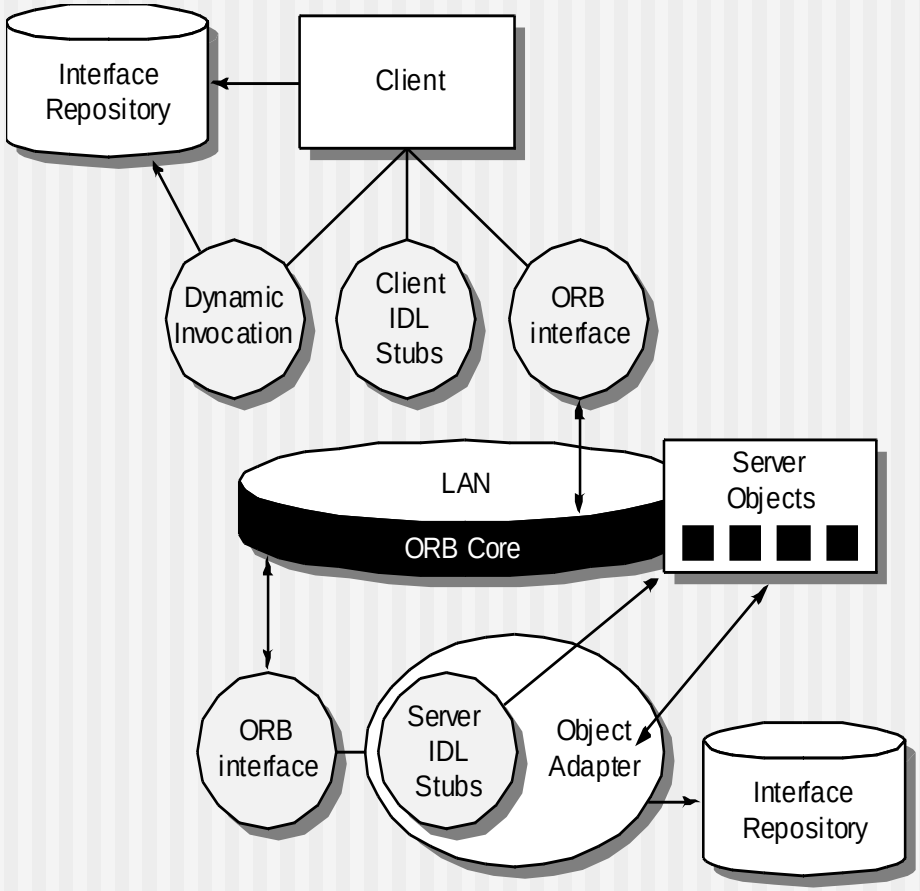
Composition

- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage
  - Underlying object model

OMG/ CORBA

- The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).
- An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
- Integration of CORBA components (without modification) within a system is assured if an interface definition language (IDL) interface is created for every component.
- Objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time.
- An interface repository contains all necessary information about the service's request and response formats.

ORB Architecture



Microsoft COM

- The *component object model* (COM) provides a specification for using components produced by various vendors 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.

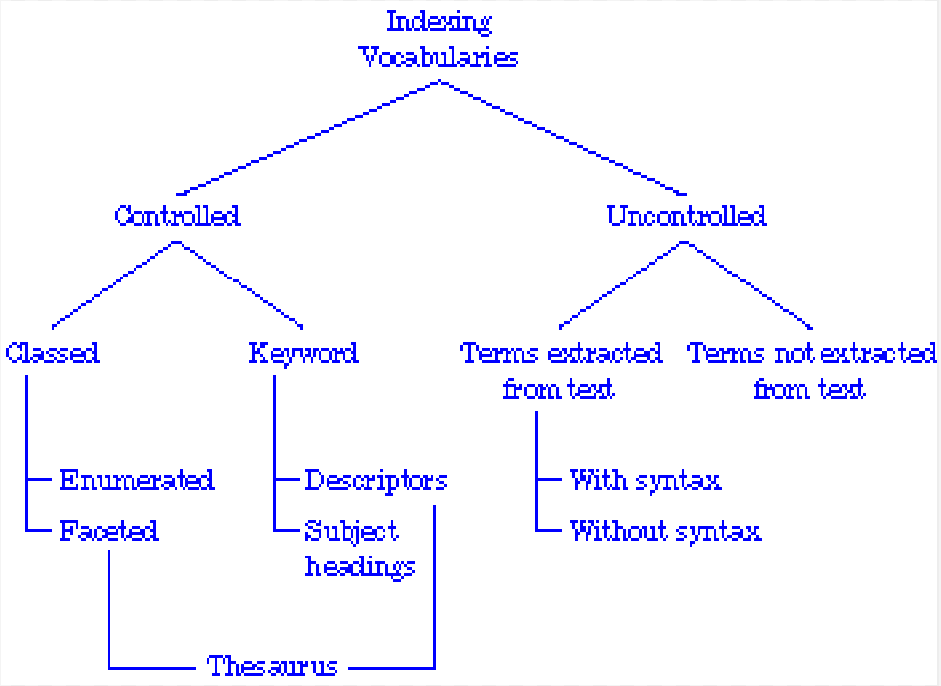
Sun JavaBeans

- 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.

Classification

- Enumerated classification—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined
- Faceted classification—a domain area is analyzed and a set of basic descriptive features are identified
- Attribute-value classification—a set of attributes are defined for all components in a domain area

Indexing



The Reuse Environment

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

Chapter 15

User Interface Design

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Interface Design

Easy to learn?

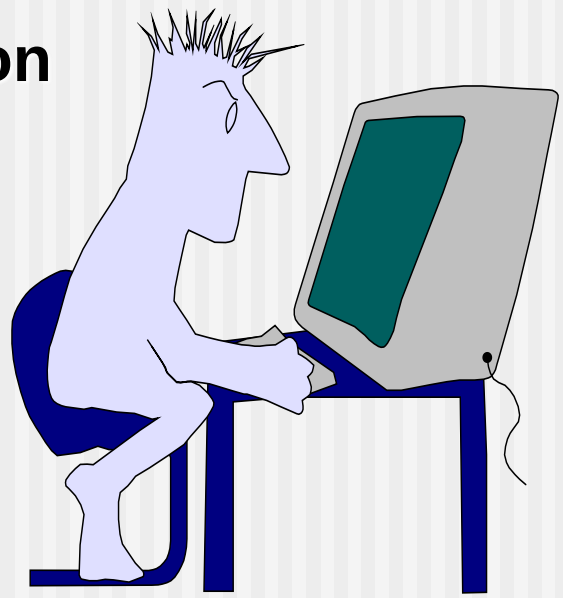
Easy to use?

Easy to understand?

Interface Design

Typical Design Errors

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
Arcane/unfriendly



Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

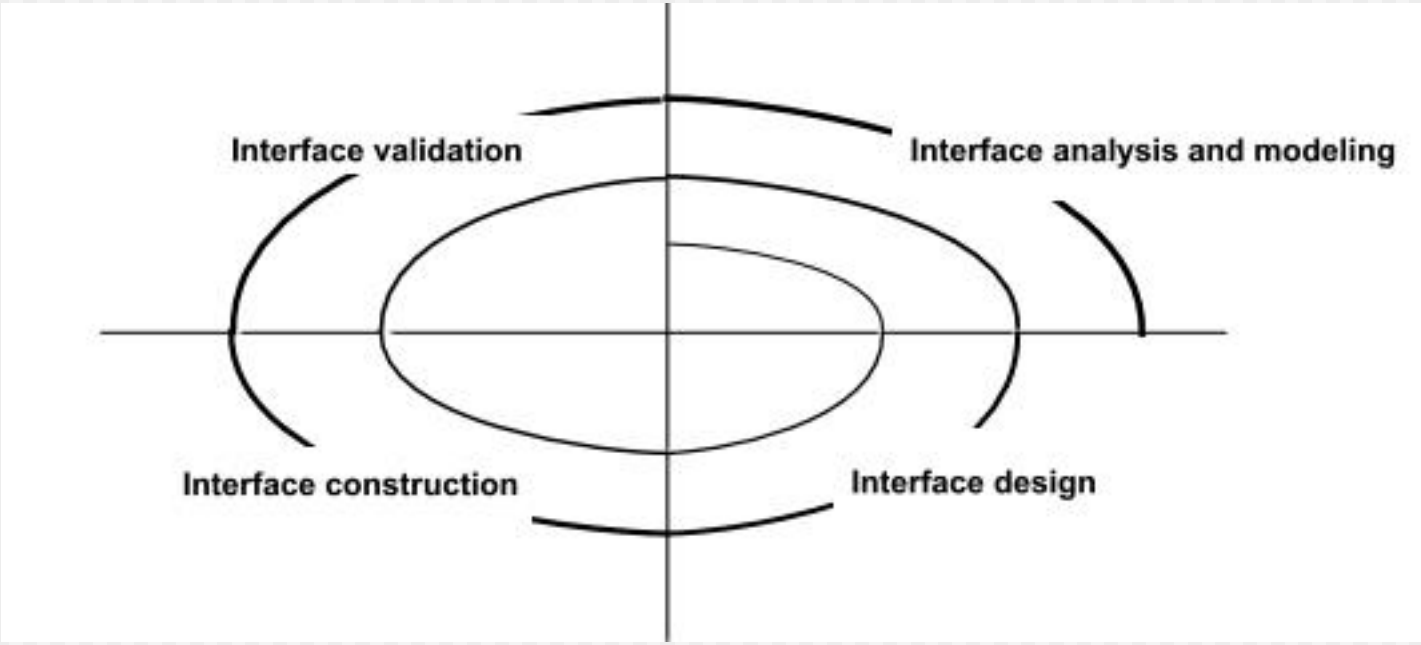
Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design Models

- User model — a profile of all end users of the system
- Design model — a design realization of the user model
- Mental model (system perception) — the user's mental image of what the interface is
- Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process



Interface Analysis

- Interface analysis means understanding
  - (1) the people (end-users) who will interact with the system through the interface;
  - (2) the tasks that end-users must perform to do their work,
  - (3) the content that is presented as part of the interface
  - (4) the environment in which these tasks will be conducted.

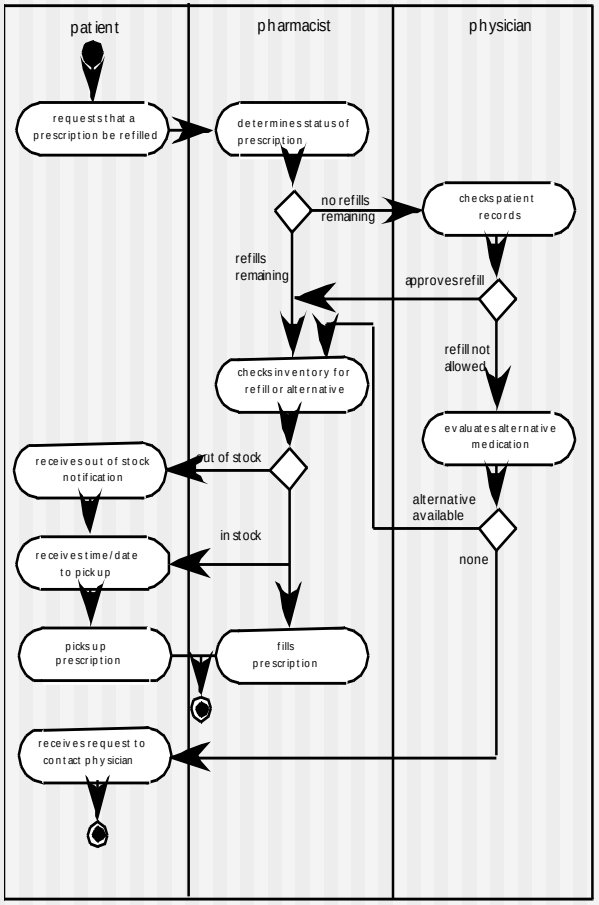
User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

Task Analysis and Modeling

- Answers the following questions …
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks—the workflow?
  - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

Swimlane Diagram



Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

Interface Design Steps

- Using information developed during interface analysis, define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

Web and Mobile App Interface Design

- *Where am I?* The interface should
  - provide an indication of the Web or Mobie App that has been accessed
  - inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
  - Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the Web or Mobile App.

Effective Web and Mobile App Interfaces

- Bruce Tognozzi [TOG01] suggests…
  - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

Interface Design Principles-I

- Anticipation—A Web or Mobile App should be designed so that it anticipates the use's next move.
- Communication—The interface should communicate the status of any activity initiated by the user
- Consistency—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy—The interface should facilitate user movement throughout the Web or Mobile App, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency—The design of the Web or Mobile App and its interface should optimize the user's work efficiency, not the efficiency of the software engineer who designs and builds it or the client-server environment that executes it.

Interface Design Principles-II

- Focus—The Web or Mobile App interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for Web or Mobile Apps.
- Latency reduction—The Web or Mobile App should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— A Web or Mobile App interface should be designed to minimize learning time, and once learned, to minimize relearning required when the App is revisited.

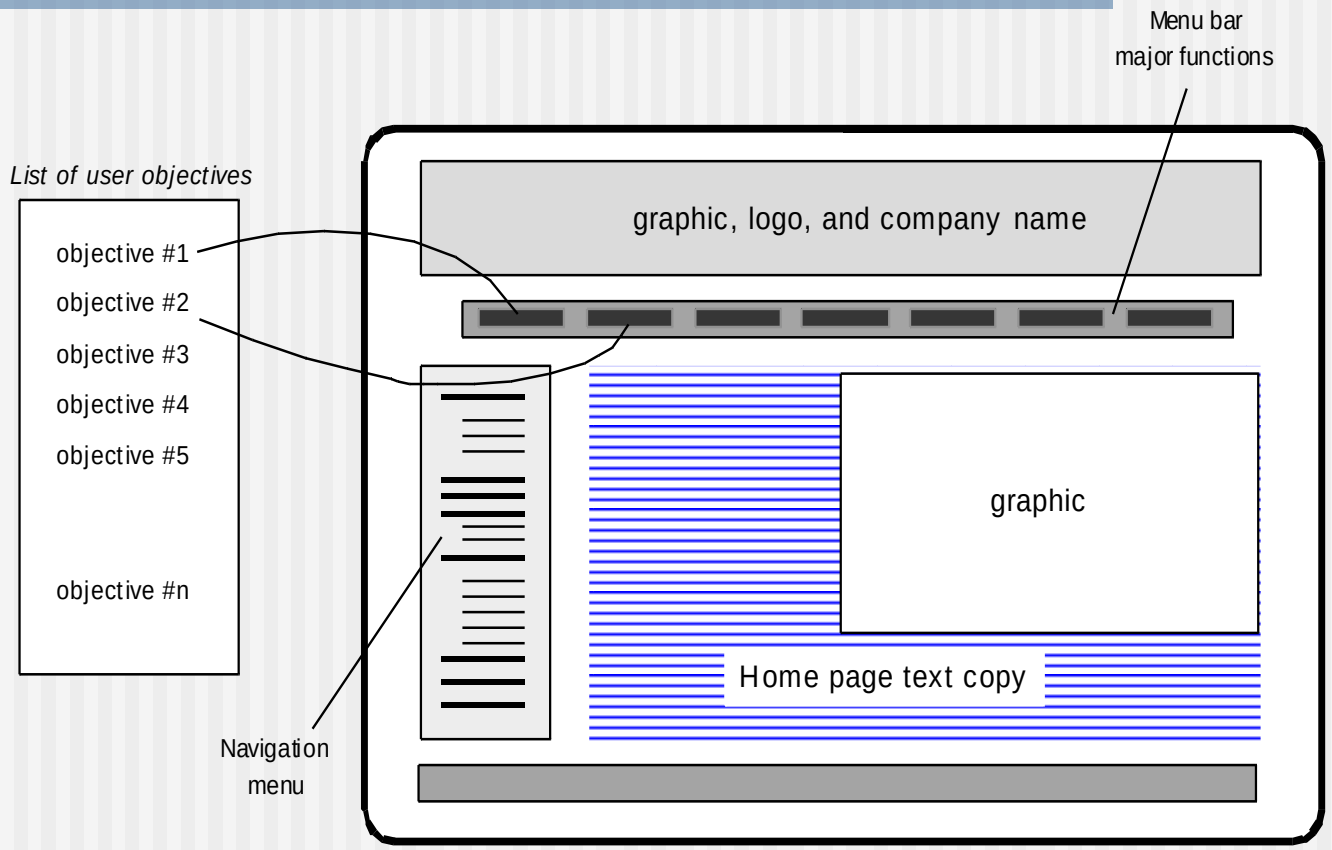
Interface Design Principles-III

- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable by young and old.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed Web or Mobile App interface provides “the illusion that users are in the same place, with the work brought to them.”

Interface Design Workflow-I

- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the Web or Mobile App interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.

Mapping User Objectives



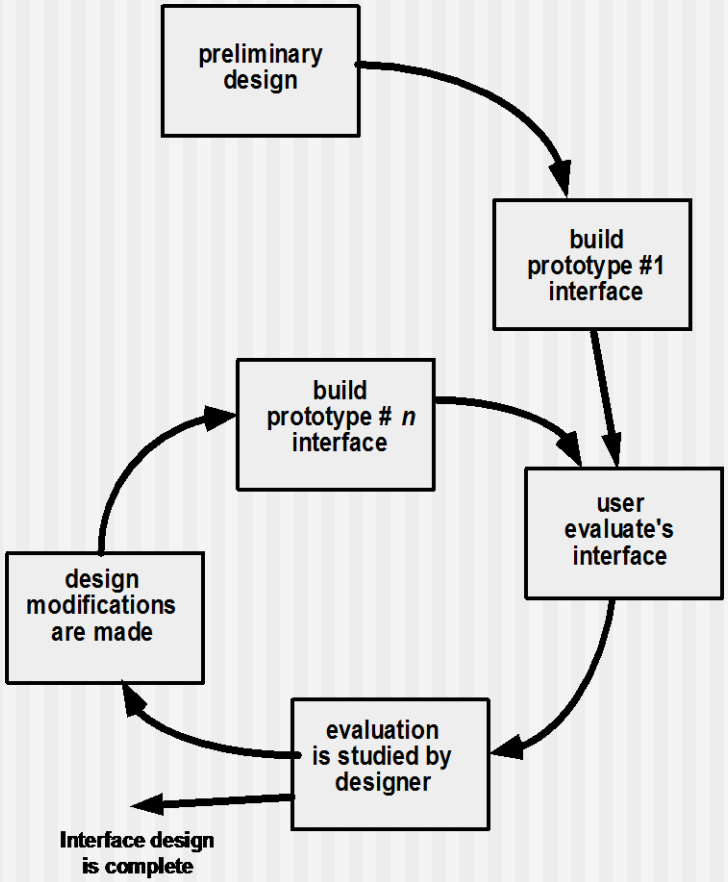
Interface Design Workflow-II

- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface.
- Develop a behavioral representation of the interface.
- Describe the interface layout for each state.
- Refine and review the interface design model.

Aesthetic Design

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

Design Evaluation Cycle



Chapter 16

Pattern-Based Design

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to for this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

Design Patterns

- *Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.*
  - Christopher Alexander, 1977
- "a three-part rule which expresses a relation between a certain context, a problem, and a solution."

Basic Concepts

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.

Effective Patterns

- Coplien [Cop05] characterizes an effective design pattern in the following way:
  - *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
  - *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
  - *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
  - *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
  - *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Generative Patterns

- *Generative patterns* describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that are unique to a given context.
- A collection of generative design patterns could be used to "generate" an application or computer-based system whose architecture enables it to adapt to change.

Kinds of Patterns

- *Architectural patterns* describe broad-based design problems that are solved using a structural approach.
- *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture
- *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.
- *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned.

Kinds of Patterns

- ***Creational patterns*** focus on the "creation, composition, and representation of objects, e.g.,
  - **Abstract factory pattern:** centralize decision of what factory to instantiate
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adapter pattern:** 'adapts' one interface for a class into one that a client expects
  - **Aggregate pattern:** a version of the Composite pattern with methods for aggregation of children
- ***Behavioral patterns*** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects
  - **Command pattern:** Command objects encapsulate an action and its parameters

Frameworks

- Patterns themselves may not be sufficient to develop a complete design.
  - In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework*, for design work.
  - That is, you can select a "*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context … which specifies their collaboration and use within a given domain." [Amb98]
- A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
  - The plug points enable you to integrate problem specific classes or functionality within the skeleton.

Describing a Pattern

- ***Pattern name***—describes the essence of the pattern in a short but expressive name
- ***Problem***—describes the problem that the pattern addresses
- ***Motivation***—provides an example of the problem
- ***Context***—describes the environment in which the problem resides including application domain
- ***Forces***—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- ***Solution***—provides a detailed description of the solution proposed for the problem
- ***Intent***—describes the pattern and what it does
- ***Collaborations***—describes how other patterns contribute to the solution
- ***Consequences***—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- ***Implementation***—identifies special issues that should be considered when implementing the pattern
- ***Known uses***—provides examples of actual uses of the design pattern in real applications
- ***Related patterns***—cross-references related design patterns

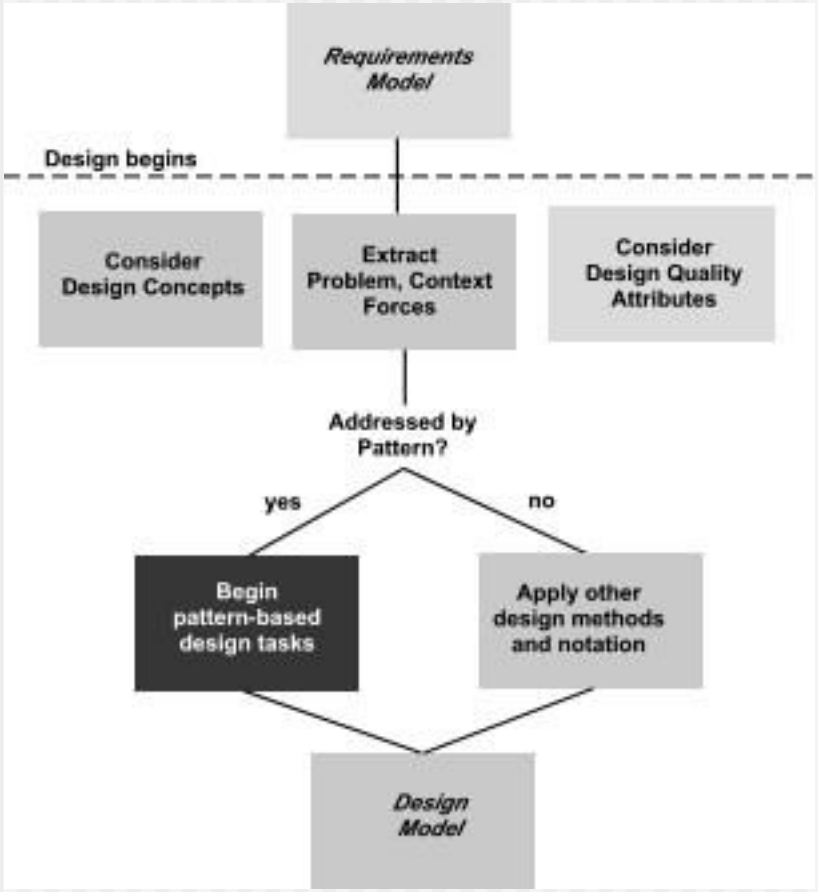
Pattern Languages

- A *pattern language* encompasses a collection of patterns
  - each described using a standardized template (Section 12.1.3) and
  - interrelated to show how these patterns collaborate to solve problems across an application domain.
- a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain.
  - The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction.

Pattern-Based Design

- A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
- The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway.
- Then …

Pattern-Based Design



Thinking in Patterns

- Shalloway and Trott [Sha05] suggest the following approach that enables a designer to think in patterns:
  - 1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
  - 2. Examining the big picture, extract the patterns that are present at that level of abstraction.
  - 3. Begin your design with 'big picture' patterns that establish a context or skeleton for further design work.
  - 4. "Work inward from the context" [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
  - 5. Repeat steps 1 to 4 until the complete design is fleshed out.
  - 6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.

Design Tasks—I

- Examine the requirements model and develop a problem hierarchy.
- Determine if a reliable pattern language has been developed for the problem domain.
- Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
- Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.
- Repeat steps 2 through 5 until all broad problems have been addressed.

Design Tasks—II

- If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
- Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
- Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

Pattern Organizing Table

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User Interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

Common Design Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
- In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

Architectural Patterns

- Example: every house (and every architectural style for houses) employs a **Kitchen** pattern.
- The **Kitchen** pattern and patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- In addition, the pattern might address problems associated with counter tops, lighting, wall switches, a central island, flooring, and so on.
- Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the 'solution' suggested by the **Kitchen** pattern.

Patterns Repositories

- There are many sources for design patterns available on the Web. Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository.
- A list of patterns repositories is presented in the sidebar near Section 12.3

Component-Level Patterns

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirement model.
- In many cases, design patterns of this type focus on some functional element of a system.
- For example, the **SafeHomeAssured.com** application must address the following design sub-problem: *How can we get product specifications and related information for any SafeHome device?*

Component-Level Patterns

- Having enunciated the sub-problem that must be solved, consider context and the system of forces that affect the solution.
- Examining the appropriate requirements model use case, the specification for a *SafeHome* device (e.g., a security sensor or camera) is used for informational purposes by the consumer.
  - However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.
- The solution to the sub-problem involves a **search.** Since searching is a very common problem, it should come as no surprise that there are many search-related patterns.
- See Section 12.4

User Interface (UI) Patterns

- **Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.
- **Page layout.** Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications)
- **Forms and input.** Consider a variety of design techniques for completing form-level input.
- **Tables.** Provide design guidance for creating and manipulating tabular data of all kinds.
- **Direct data manipulation.** Address data editing, modification, and transformation.
- **Navigation.** Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.
- **Searching.** Enable content-specific searches through information maintained within a Web site or contained by persistent data stores that are accessible via an interactive application.
- **Page elements.** Implement specific elements of a Web page or display screen.
- **E-commerce.** Specific to Web sites, these patterns implement recurring elements of e-commerce applications.

WebApp Patterns

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action; how a user expands content based on usage context and user desires; how to best describe the destination that is implied by a link; how to inform the user about the status of an on-going interaction, and interface related issues.
- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability; how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communications, and other algorithmic elements within a WebApp.

Design Granularity

- When a problem involves "big picture" issues, attempt to develop solutions (and use relevant patterns) that focus on the big picture.
- Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly.
- In terms of the level of granularity, patterns can be described at the following levels:

Design Granularity

- **Architectural patterns.** This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.
- **Design patterns.** These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component to component communication. An example might be the *Broadsheet* pattern for the layout of a WebApp homepage.
- **Component patterns.** This level of abstraction relates to individual small-scale elements of a WebApp. Examples include individual interaction elements (e.g. radio buttons, text books), navigation items (e.g. how might you format links?) or functional elements (e.g. specific algorithms).

Mobile User Interface Patterns

- Check-in screens
- Maps
- Popovers
- Sign-up flows
- Custom Tab Navigation
- Invitations

Mobile App Design Patterns

- Active Objects
- Applications Controller
- Communicator
- Data Transfer Object
- Domain Model
- Entity Translator
- Lazy Acquisition
- Model-View-Comtroller
- Pagination
- Reliable Sessions
- Synchronization
- Transaction Script

Chapter 17

- **WebApp Design**

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Design & WebApps

"There are essentially two basic approaches to design: the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer."

Jakob Nielsen

- *When should we emphasize WebApp design?*
  - when content and function are complex
  - when the size of the WebApp encompasses hundreds of content objects, functions, and analysis classes
  - when the success of the WebApp will have a direct impact on the success of the business

Design & WebApp Quality

- Security
  - Rebuff external attacks
  - Exclude unauthorized access
  - Ensure the privacy of users/customers
- Availability
  - the measure of the percentage of time that a WebApp is available for use
- Scalability
  - **Can** the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume
- Time to Market

Quality Dimensions for End-Users

- ***Time***
  - How much has a Web site changed since the last upgrade?
  - How do you highlight the parts that have changed?
- ***Structural***
  - How well do all of the parts of the Web site hold together.
  - Are all links inside and outside the Web site working?
  - Do all of the images work?
  - Are there parts of the Web site that are not connected?
- ***Content***
  - Does the content of critical pages match what is supposed to be there?
  - Do key phrases exist continually in highly-changeable pages?
  - Do critical pages maintain quality content from version to version?
  - What about dynamically generated HTML pages?

Quality Dimensions for End-Users

- ***Accuracy and Consistency***
  - Are today's copies of the pages downloaded the same as yesterday's? Close enough?
  - Is the data presented accurate enough? How do you know?
- ***Response Time and Latency***
  - Does the Web site server respond to a browser request within certain parameters?
  - In an E-commerce context, how is the end to end response time after a SUBMIT?
  - Are there parts of a site that are so slow the user declines to continue working on it?
- ***Performance***
  - Is the Browser-Web-Web site-Web-Browser connection quick enough?
  - How does the performance vary by time of day, by load and usage?
  - Is performance adequate for E-commerce applications?

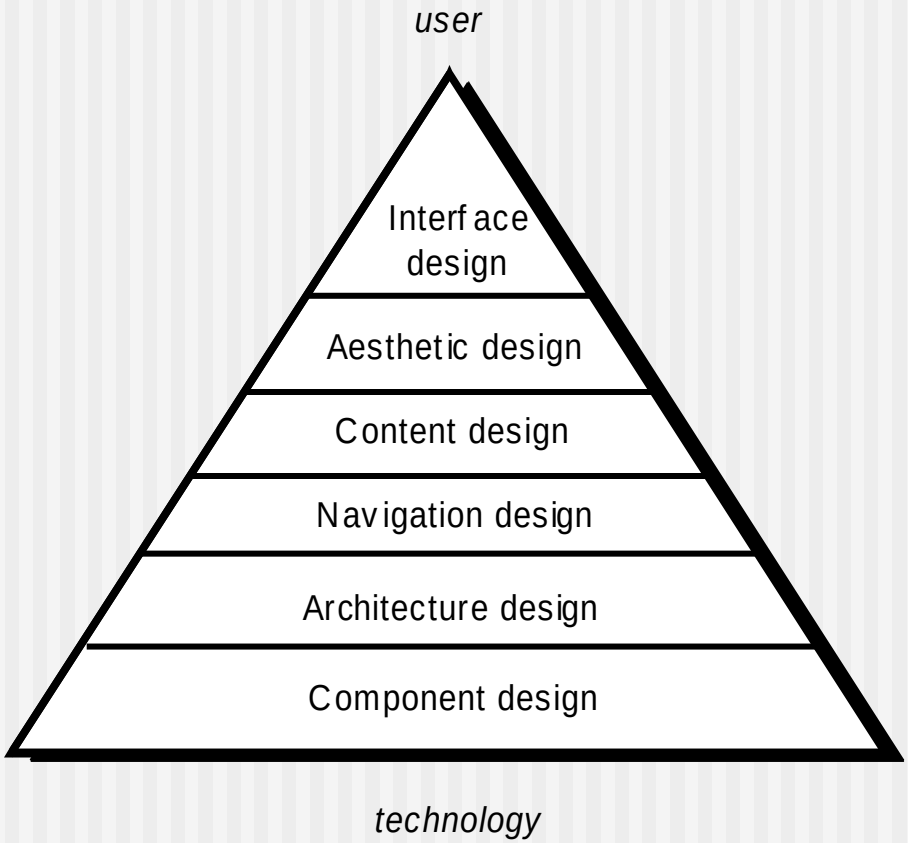
WebApp Design Goals

- Consistency
  - Content should be constructed consistently
  - Graphic design (aesthetics) should present a consistent look across all parts of the WebApp
  - Architectural design should establish templates that lead to a consistent hypermedia structure
  - Interface design should define consistent modes of interaction, navigation and content display
  - Navigation mechanisms should be used consistently across all WebApp elements

WebApp Design Goals

- Identity
  - Establish an “identity” that is appropriate for the business purpose
- Robustness
  - The user expects robust content and functions that are relevant to the user’s needs
- Navigability
  - designed in a manner that is intuitive and predictable
- Visual appeal
  - the look and feel of content, interface layout, color coordination, the balance of text, graphics and other media, navigation mechanisms must appeal to end-users
- Compatibility
  - With all appropriate environments and configurations

WebE Design Pyramid



WebApp Interface Design

- *Where am I?* The interface should
  - provide an indication of the WebApp that has been accessed
  - inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
  - Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

Effective WebApp Interfaces

- Bruce Tognozzi [TOG01] suggests…
  - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

Interface Design Principles-I

- Anticipation—A WebApp should be designed so that it anticipates the use's next move.
- Communication—The interface should communicate the status of any activity initiated by the user
- Consistency—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

Interface Design Principles-II

- Focus—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for WebApps.
- Latency reduction—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

Interface Design Principles-III

- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable by young and old.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”

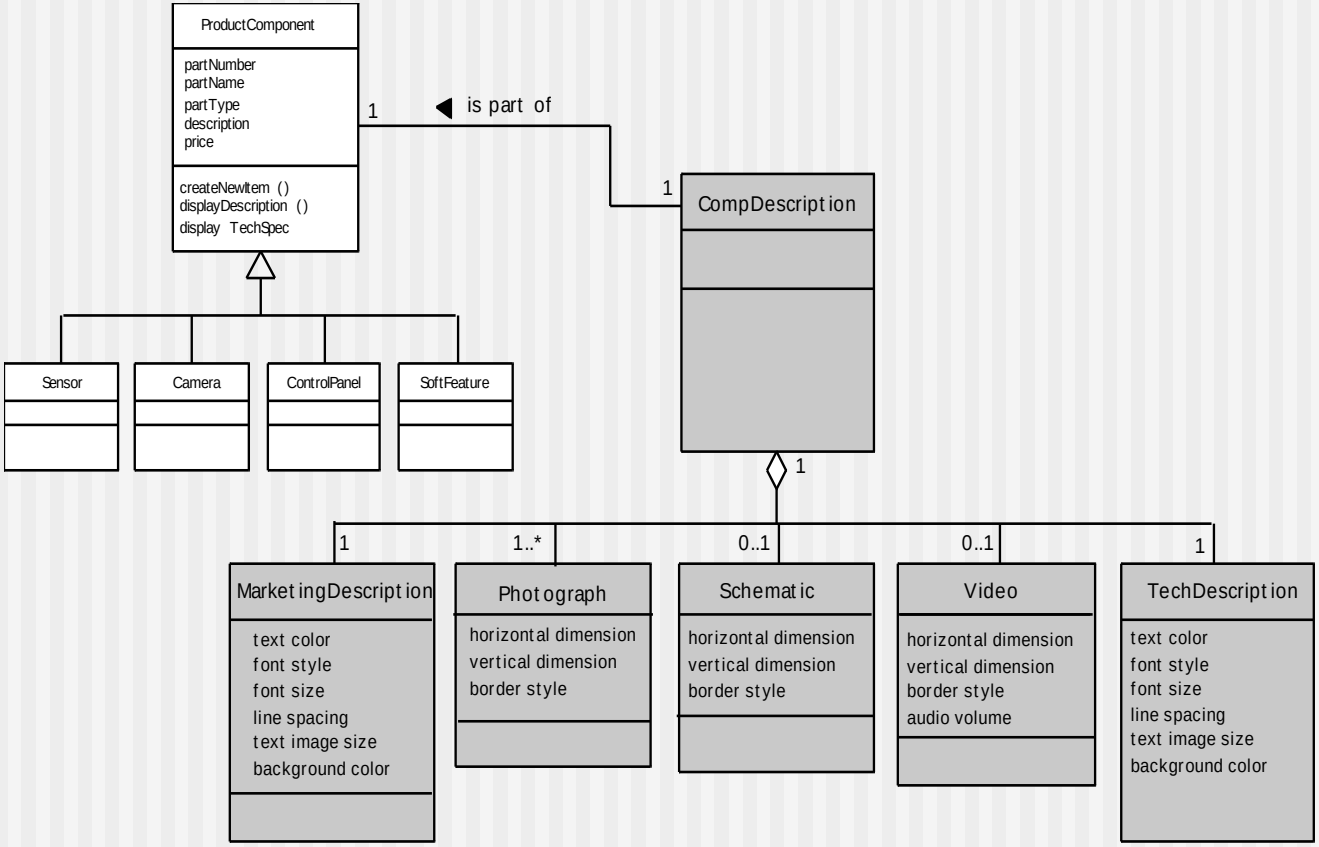
Aesthetic Design

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

Content Design

- Develops a design representation for content objects
  - For WebApps, a content object is more closely aligned with a data object for conventional software
- Represents the mechanisms required to instantiate their relationships to one another.
  - analogous to the relationship between analysis classes and design components described in Chapter 11
- A content object has attributes that include content-specific information and implementation-specific attributes that are specified as part of design

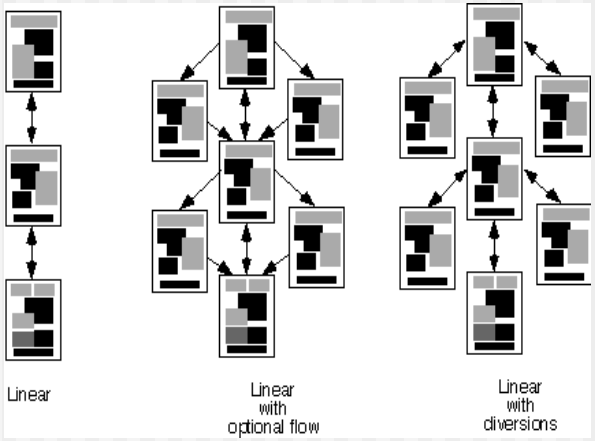
Design of Content Objects

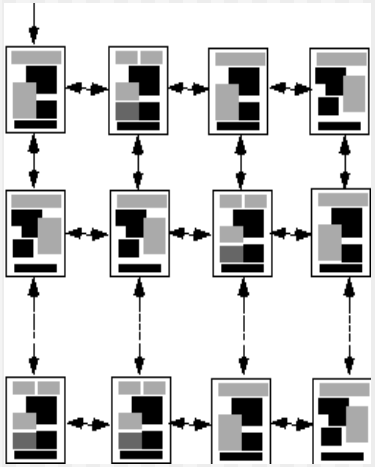


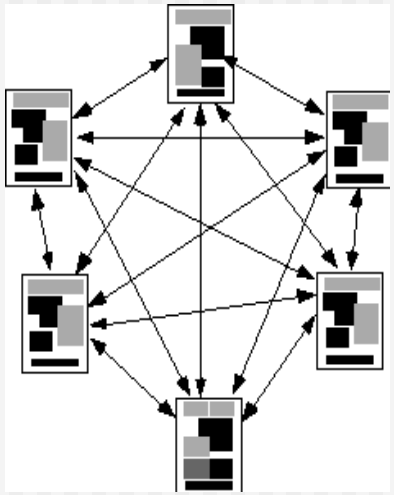
Architecture Design

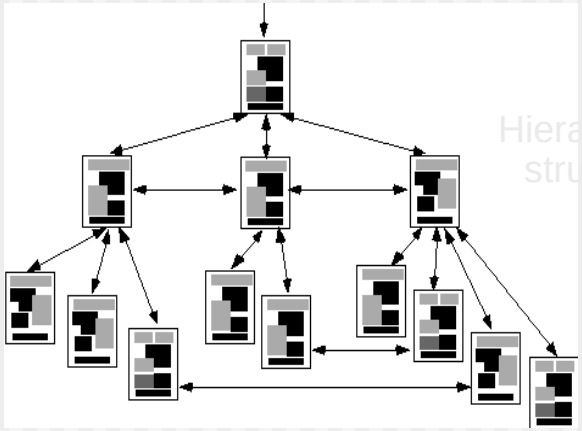
- *Content architecture* focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.
  - The term information architecture is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.
- *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.
- Architecture design is conducted in parallel with interface design, aesthetic design and content design.

Content Architecture





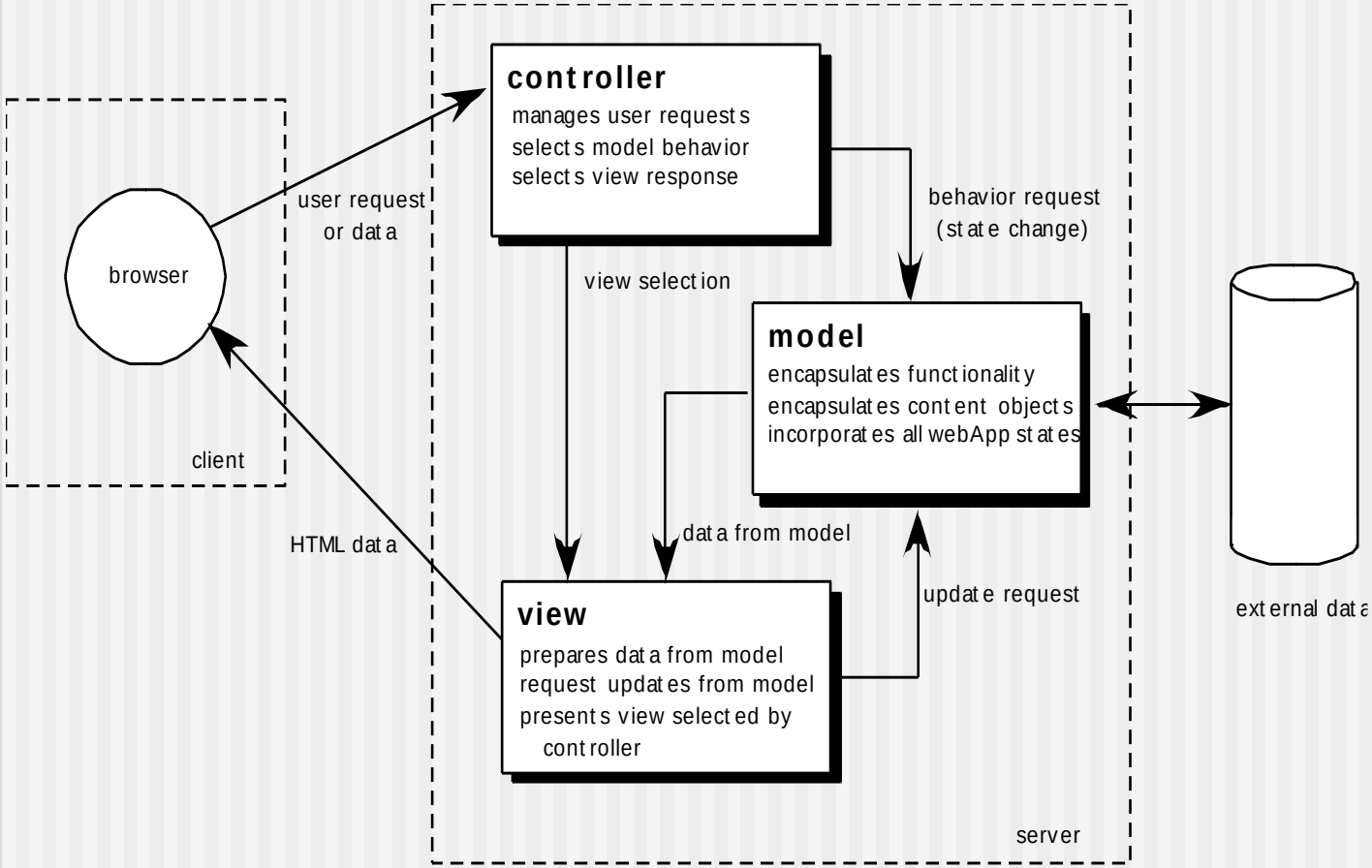




MVC Architecture

- The *model* contains all application specific content and processing logic, including
  - all content objects
  - access to external data/information sources,
  - all processing functionality that are application specific
- The *view* contains all interface specific functions and enables
  - the presentation of content and processing logic
  - access to external data/information sources,
  - all processing functionality required by the end-user.
- The *controller* manages access to the model and the view and coordinates the flow of data between them.

MVC Architecture

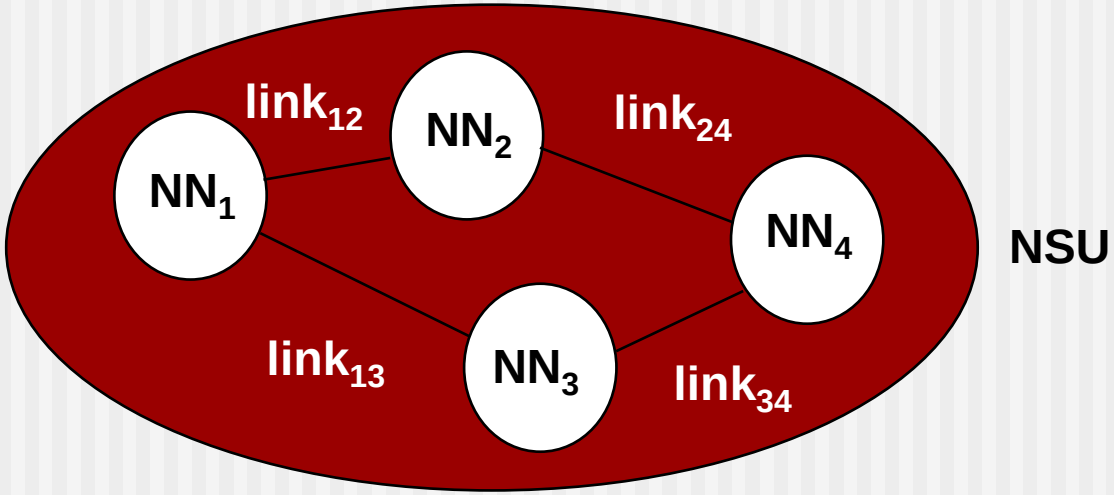


Navigation Design

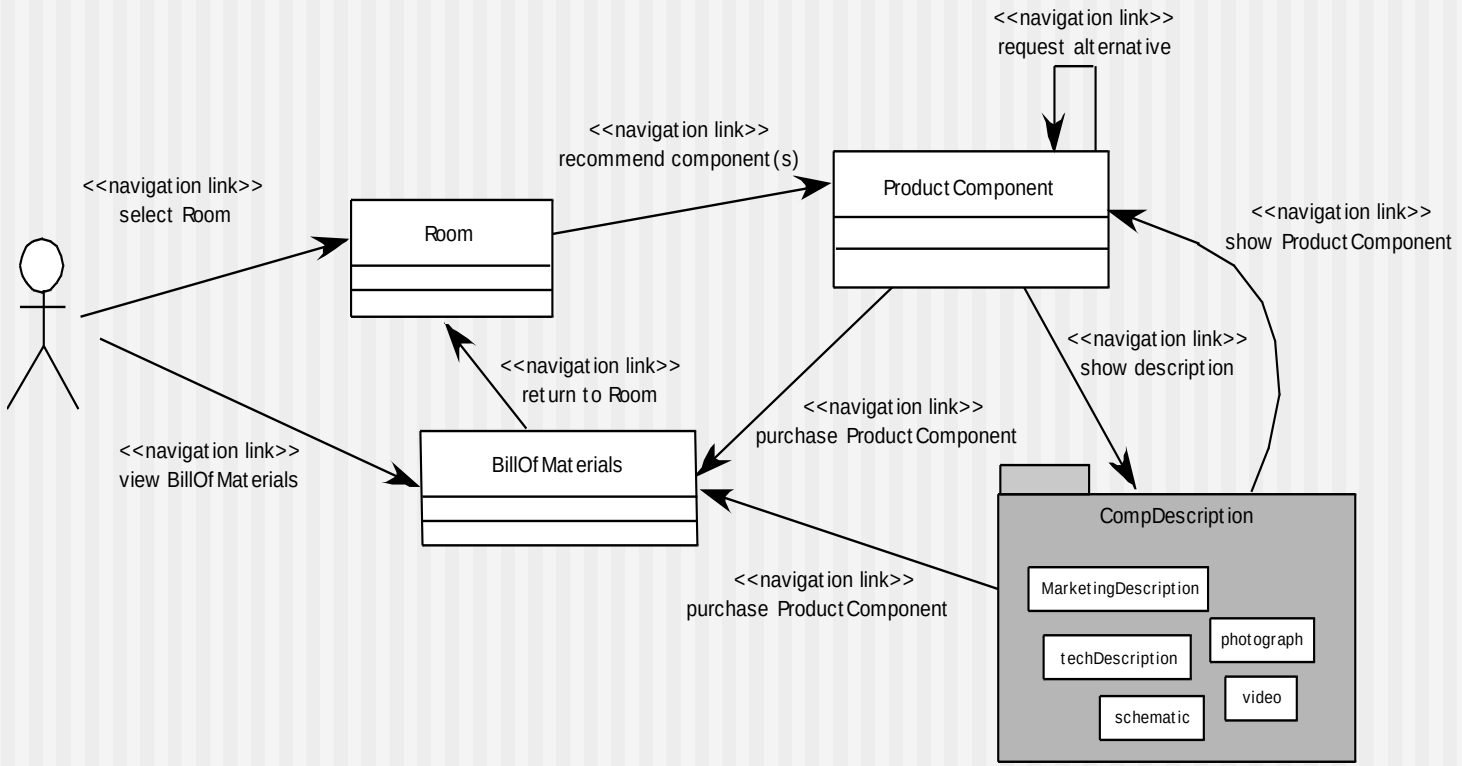
- Begins with a consideration of the user hierarchy and related use-cases
  - Each actor may use the WebApp somewhat differently and therefore have different navigation requirements
- As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)
  - NSU—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements"

Navigation Semantic Units

- Navigation semantic unit
  - Ways of navigation (WoN)—represents the best navigation way or path for users with certain profiles to achieve their desired goal or sub-goal. Composed of …
    - Navigation nodes (NN) connected by Navigation links



Creating an NSU



Navigation Syntax

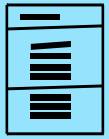
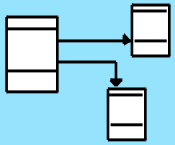
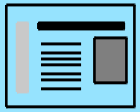
- *Individual navigation link*—text-based links, icons, buttons and switches, and graphical metaphors..
- *Horizontal navigation bar*—lists major content or functional categories in a bar containing appropriate links. In general, between 4 and 7 categories are listed.
- *Vertical navigation column*
  - lists major content or functional categories
  - lists virtually all major content objects within the WebApp.
- *Tabs*—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.
- *Site maps*—provide an all-inclusive tab of contents for navigation to all content objects and functionality contained within the WebApp.

Component-Level Design

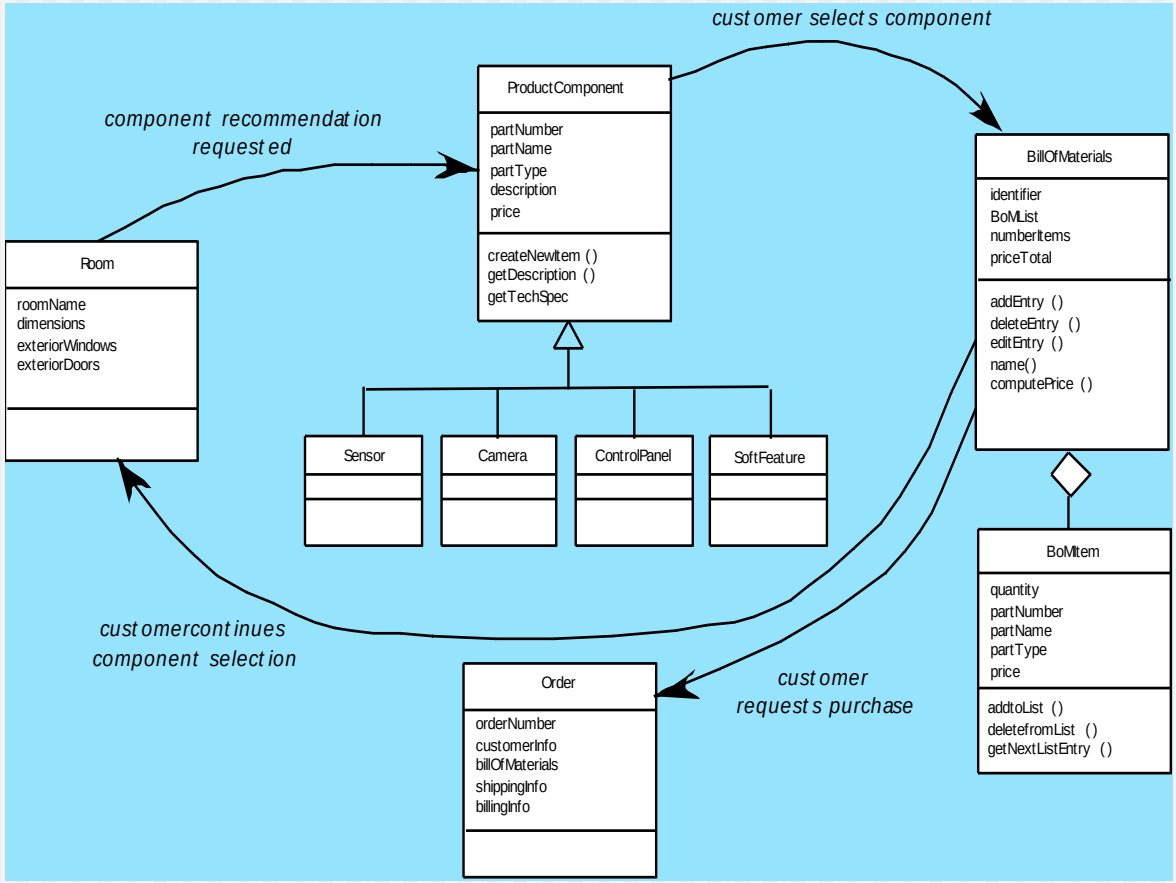
- WebApp components implement the following functionality
  - perform localized processing to generate content and navigation capability in a dynamic fashion
  - provide computation or data processing capability that are appropriate for the WebApp's business domain
  - provide sophisticated database query and access
  - establish data interfaces with external corporate systems.

OOHDM

- *Object-Oriented Hypermedia Design Method* (OOHDM)

	 conceptual design	 navigational design	 abstract interface design	 implementation
work products	Classes, sub-systems, relationships, attributes	Nodes, links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	executable WebApp
design mechanisms	Classification, composition, aggregation, generalization specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resource provided by target environment
design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects	Correctness; Application performance; completeness

Conceptual Schema



Chapter 18

- **MobileApp Design**

Slide Set to accompany

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

by Roger S. Pressman and Bruce R. Maxim

Mobile Development Considerations – 1

- Multiple hardware and software platforms
- Many development frameworks and programming languages.
- Many app stores with differing acceptance rules and tool requirements
- Short development cycles
- User interface limitations

Mobile Development Considerations – 2

- Complex camera/sensor interaction
- Effective use of context
- Power management
- Security and privacy models/policies
- Device limitations (computation and storage)
- Integration of external services
- Texting complexities

MobileApp Development Process Model

- Formulation
- Planning
- Analysis
- Engineering
- Implementation and testing
- User evaluation

MobileApp Quality Checklist - 1

- Can content and/or function and/or navigation options be tailored to the user's preferences?
- Can content and/or functionality be customized to the bandwidth at which the user communicates? Does the app account for weak or lost signal in an acceptable manner?
- Can content and/or function and/or navigation options be made context aware according to the user's preferences?
- Has adequate consideration been given to the power availability on the target device(s)?
- Have graphics, media (audio, video), and other web or cloud services been used appropriately?

MobileApp Quality Checklist - 2

- Is the overall page design easy to read and navigate?
- Does the app take screen size differences into account?
- Does the user interface conform to the display and interaction standards adopted for the targeted mobile device(s)?
- Does the app conform to the reliability, security, and privacy expectations of its users?
- What provisions have been made to ensure app remains current?
- Has the MobileApp been tested in all targeted user environments and for all targeted devices?

MobileApp User Interface Quality Requirements

- Is the overall page design easy to read and navigate?
- Does the app take screen size differences into account?
- Does the user interface conform to the display and interaction standards adopted for the targeted mobile device(s)?
- Does the app conform to the reliability, security, and privacy expectations of its users?
- What provisions have been made to ensure app remains current?
- Has the MobileApp been tested in all targeted user environments and for all targeted devices?

MobileApp User Interface Design Considerations

- Define user interface brand signatures
- Focus the portfolio of products
- Identify core user stories
- Optimize UI flows and elements
- Define scaling rules
- Create user performance dashboard
- Rely on dedicated champion with user interface engineering skills

MobileApp Design Mistakes

- Kitchen sink
- Inconsistency
- Overdesigning
- Lack of speed
- Verbiage
- Non-standard interaction
- Help-and –FAQ-itis

MobileApp Design Best Practices

- Identify the audience
- Design for context of use
- Recognize line between simplicity is not laziness
- Use the platform to its advantage
- Allow for discoverability of advanced functionality
- Use clear and consistent labels
- Cleaver icons should never be developed at the expense of user understanding
- Long scrolling forms trump multiple screens

Assessing Mobile Interactive Development Environments

- General productivity features
- Third-party SDK integration
- Post-compilation tooks
- Over the air development support
- End-to-end mobile application development
- Documentation and tutorials
- Graphical user interface builders

MobileApp Middleware

- Facilitates communication and coordination of distributed components
- Allows developers to rely on abstractions and hide mobile environment details
- Helps MobileApps to achieve context awareness as required

Service Computing

- Focuses on architectural design and enables application development through service discovery and composition
- Allows MobileApp developers to avoid the need to integrate service source code into the client running on a mobile device
- Runs out of the provider's server
- Loosely coupled with applications
- Provides an API to allow service to be treated like an abstract black box

Cloud Computing

- Focuses on the effective delivery of services to users through flexible and scalable resource virtualization and loading balancing
- Lets the client (either a user or program) request computing capabilities as needed, across network boundaries anywhere or any time

Cloud Computing Architecture

- Cloud architecture has three service layers
  - **Software as service** layer consists of software components and applications hosted by third-party service providers
  - **Platform as service** layer provides a collaborative development platform to assist with design, implementation, and testing by geographically distributed team members
  - **Infrastructure and service** provides virtual computing resources (storage, processing power, network connectivity) on the cloud