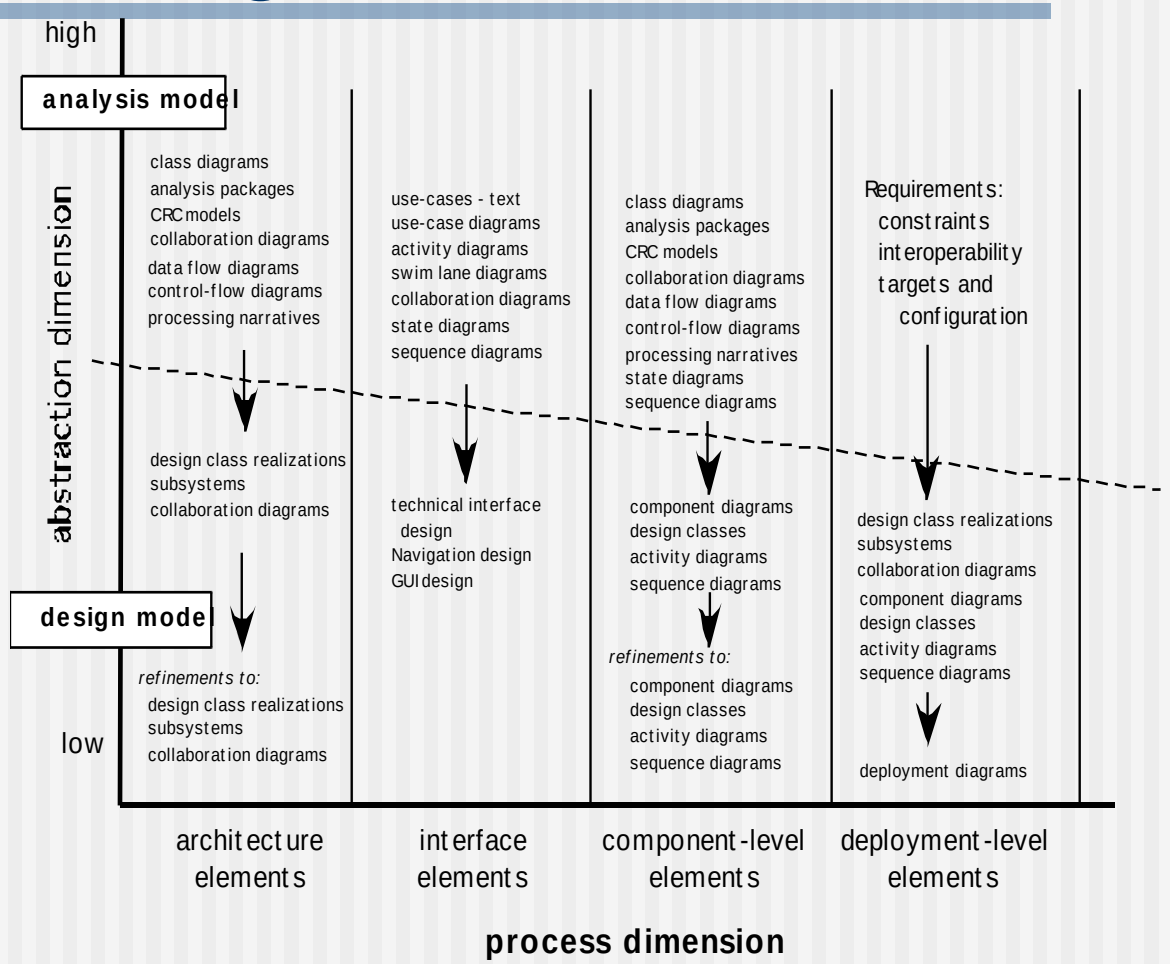
Lecture 6 Software Design II

- **Design Modeling**

“How to create the representations of the software?”

The Design Model



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014) Slides copyright 2014 by Roger Pressman.

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/Class Design**

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

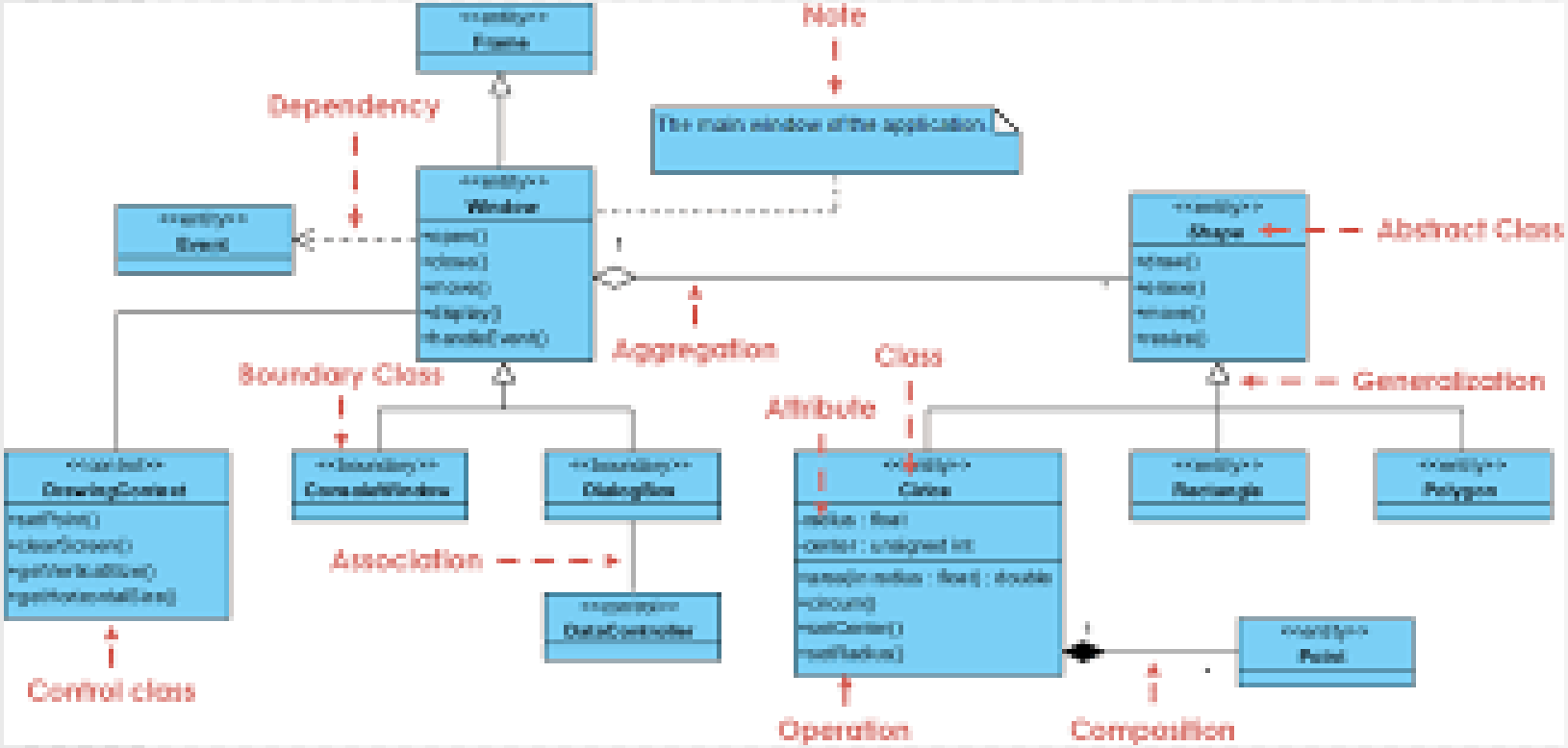
object: automobile

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

Data/Class Design Example



Data Modeling Tasks

- Elaborate the classes with additional attributes and methods required for the software
- Add classes to manage data and provide functionalities
- Identify and enhance relationships between classes
- Map the data classes to storage such as file formats and database schemas

- **Architectural Design**

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).

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.

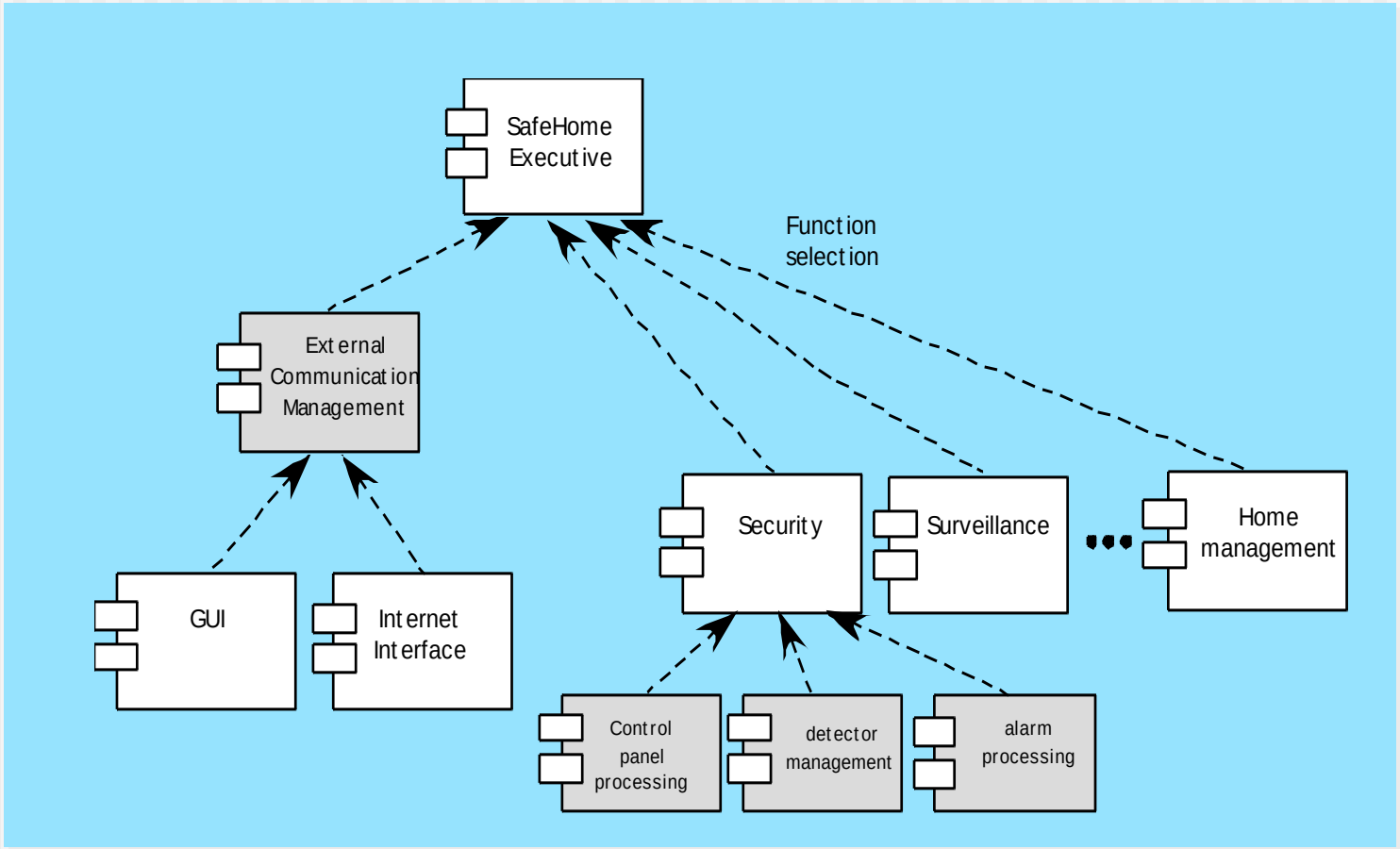
Why is Architecture Important?

- 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].

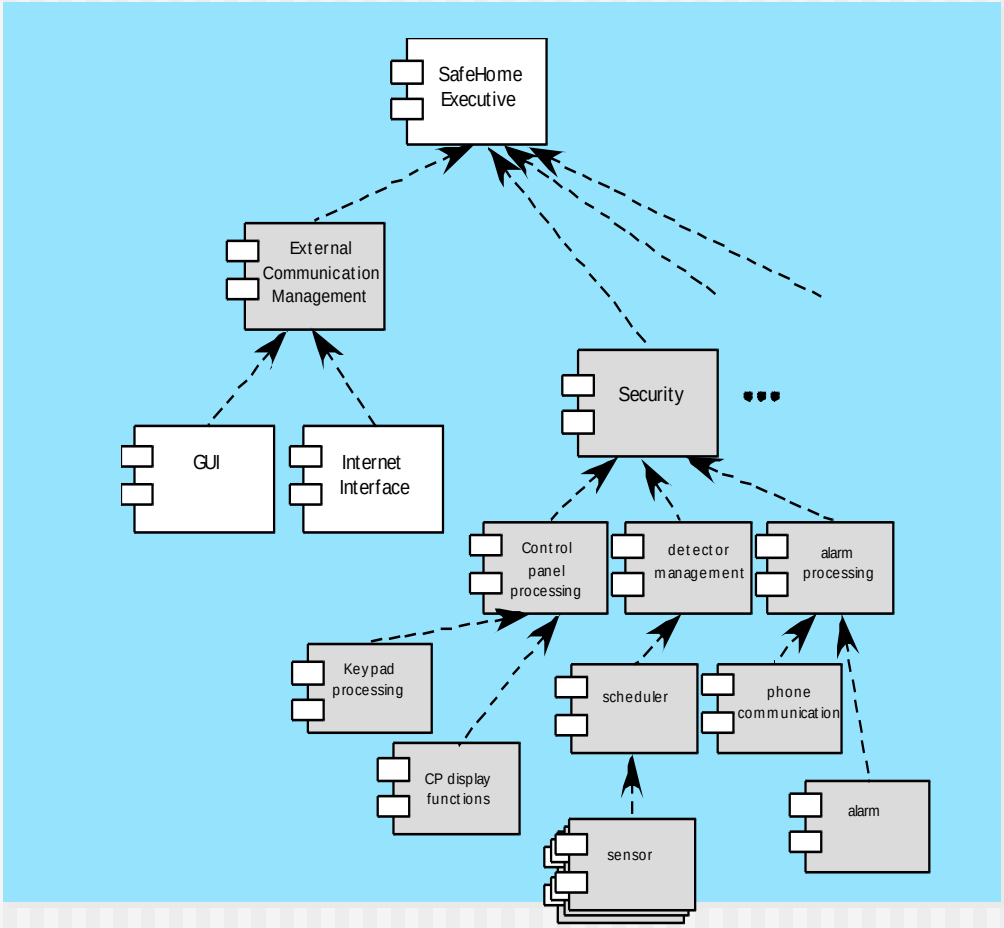
Architectural Descriptions

- 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."

Component Structure



Refined Component Structure

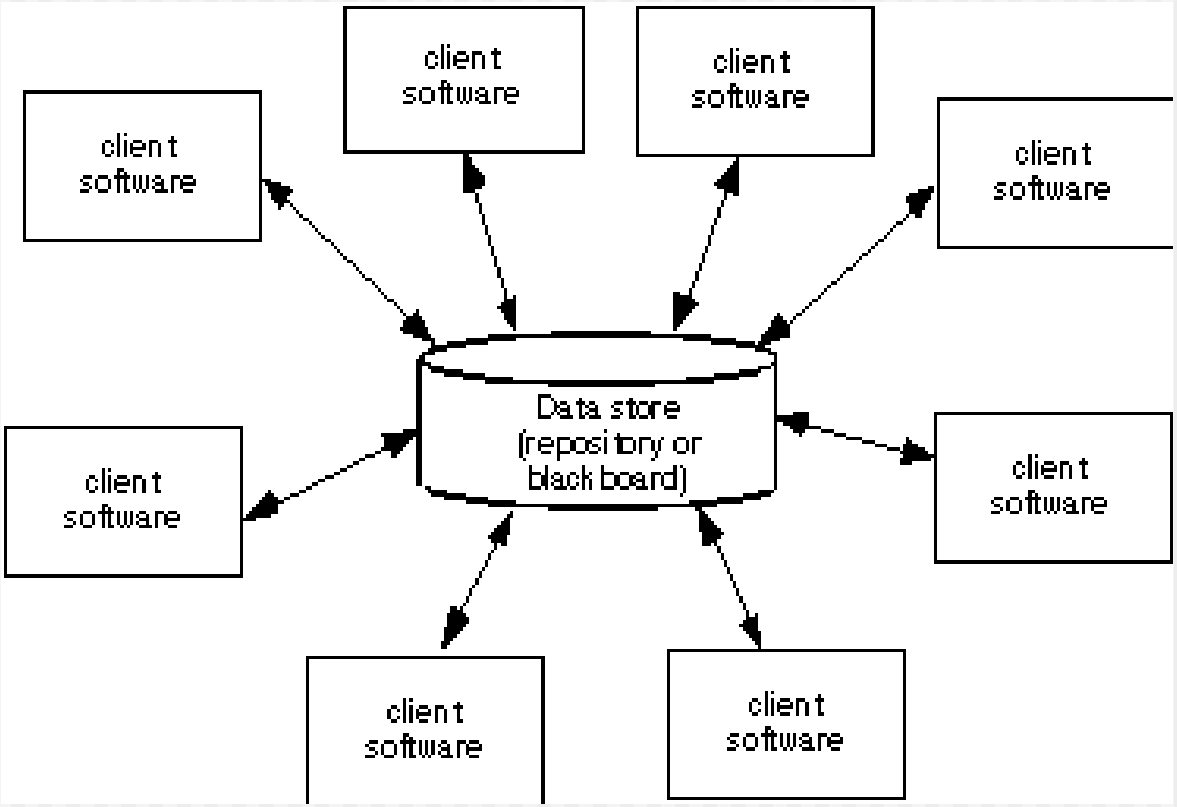


Architectural Styles

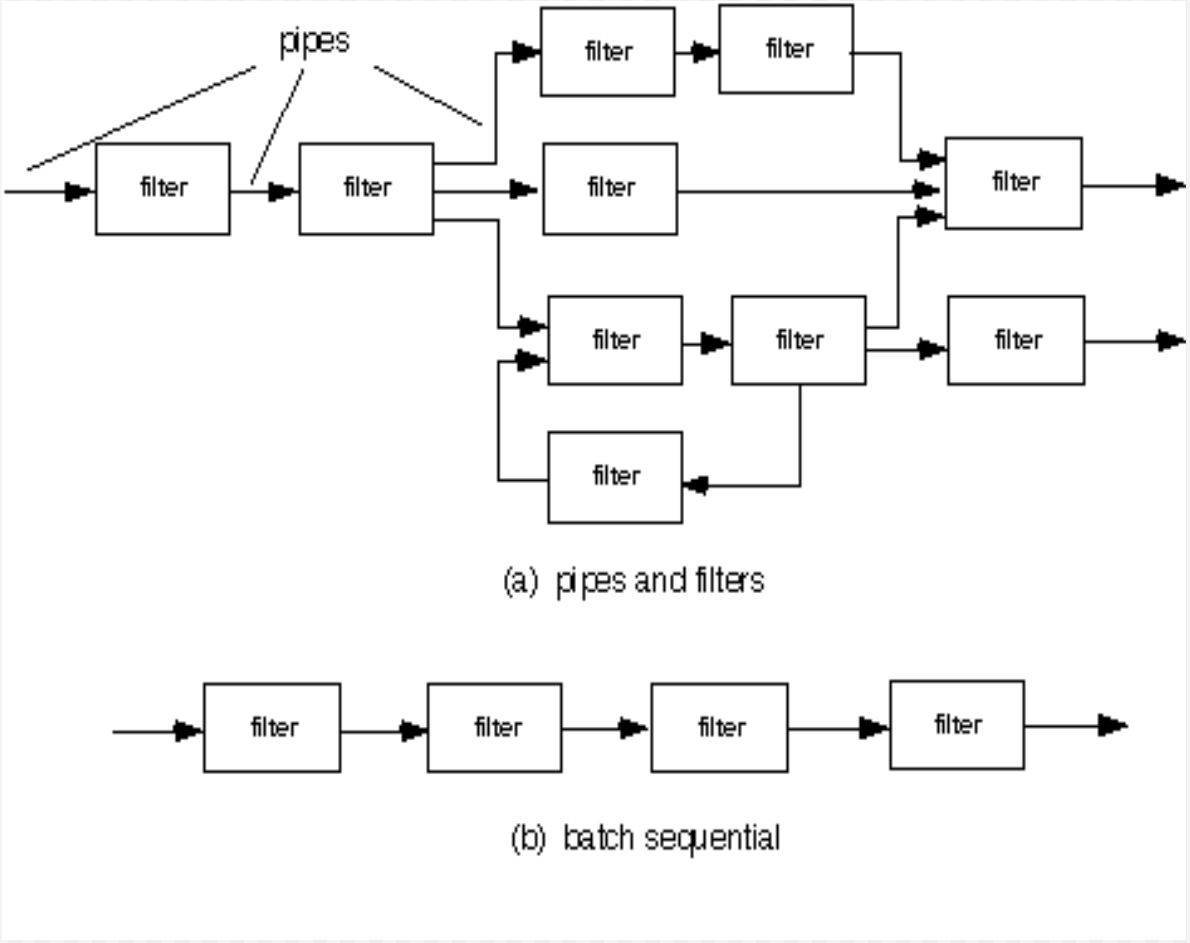
Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable "communication, coordination and cooperation" among components, (3) **constraints** that 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

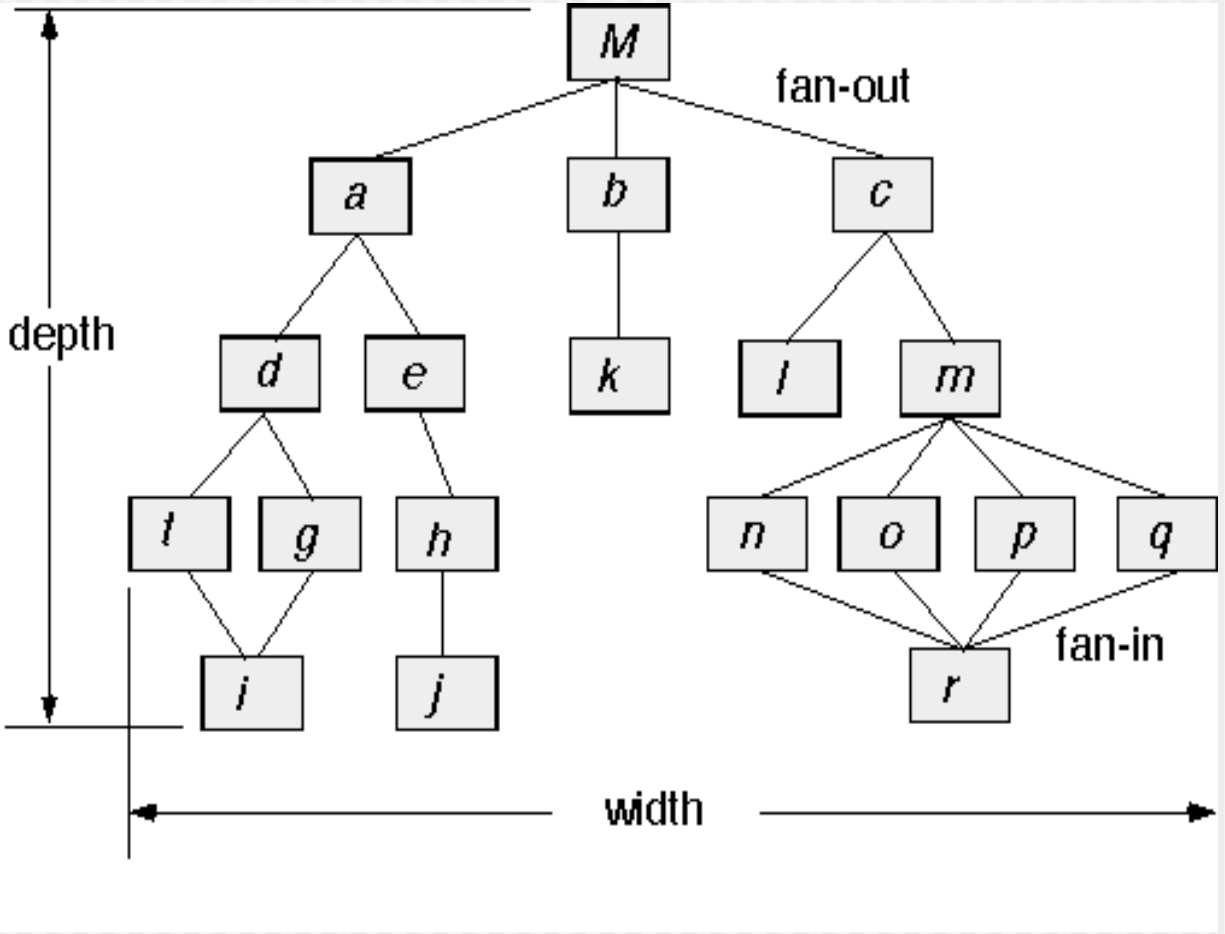
Data-Centered Architecture



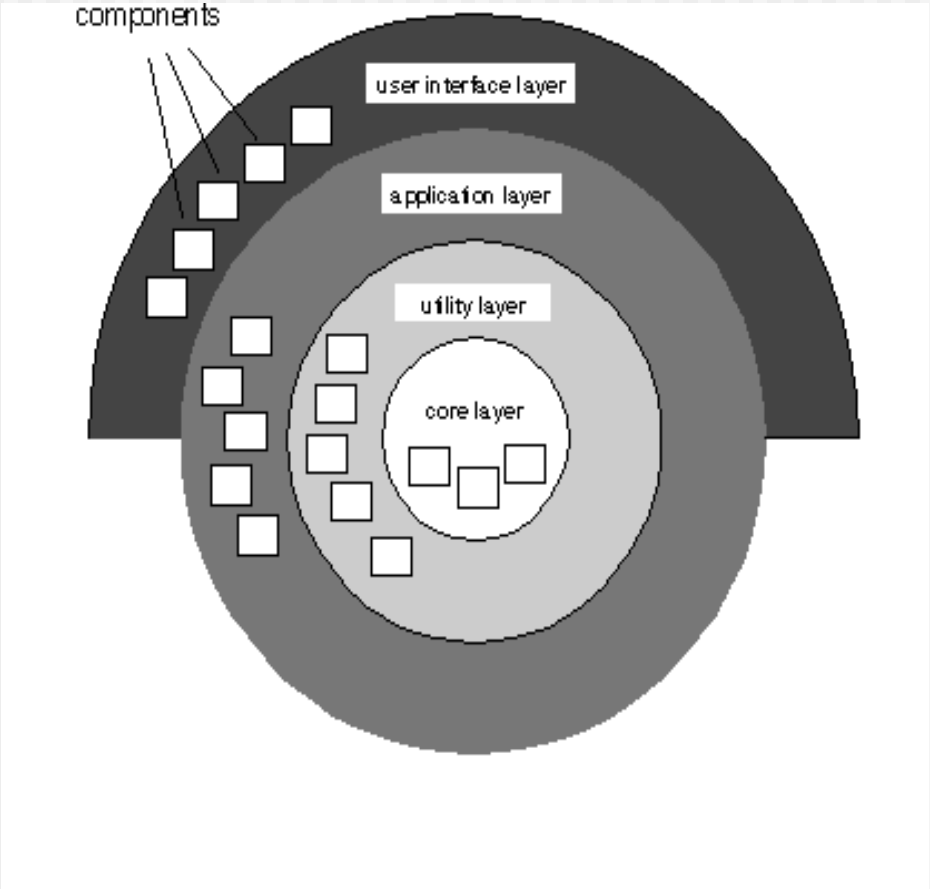
Data Flow Architecture



Call and Return Architecture



Layered Architecture



Architectural 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 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.

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

Agility and Architecture

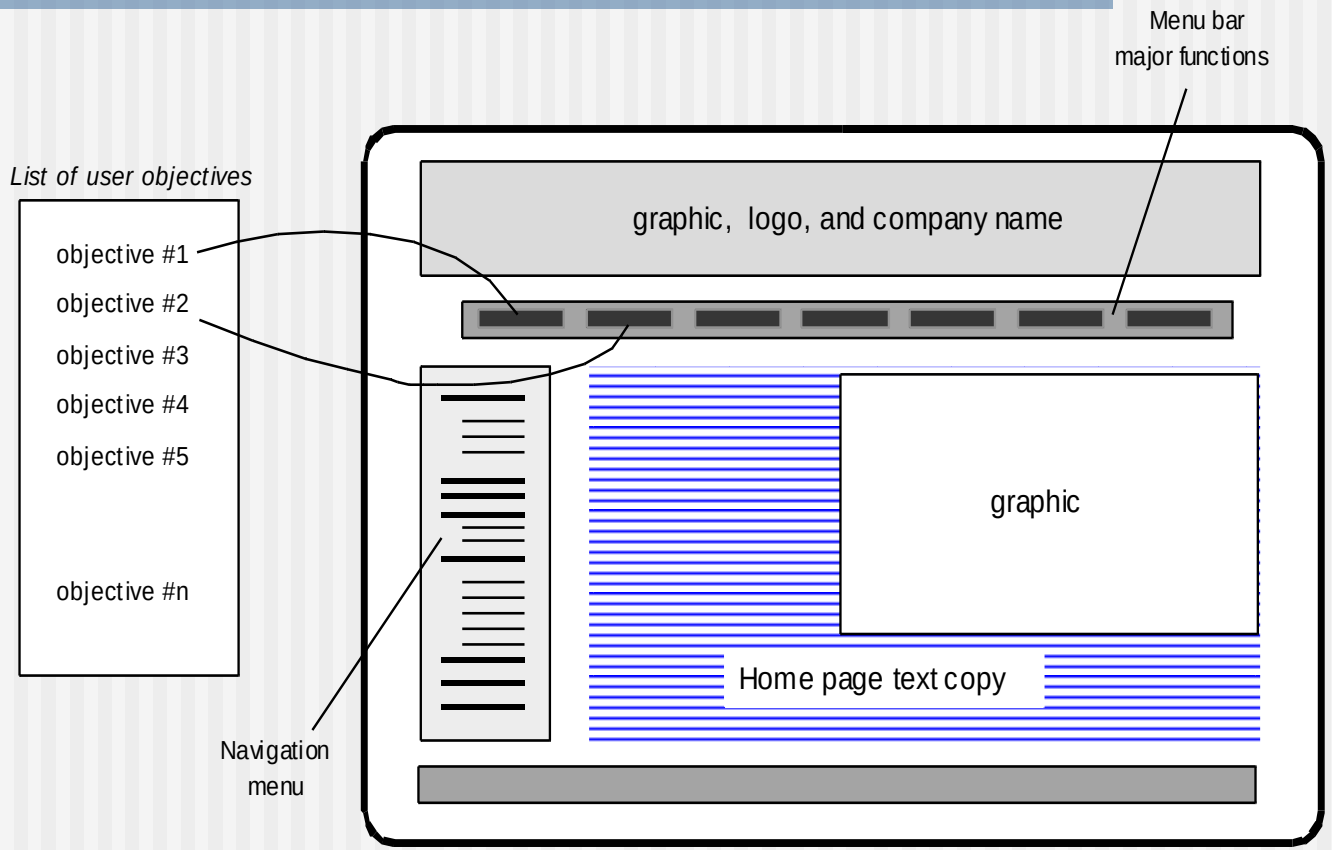
- To avoid rework, user stories are used to create and evolve an architectural model (walking skeleton) before coding
- Hybrid models which allow software architects contributing users stories to the evolving storyboard
- Well run agile projects include delivery of work products during each sprint
-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

- **Interface Design**

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)

User Interface Design Example



Interface Design

Easy to learn?

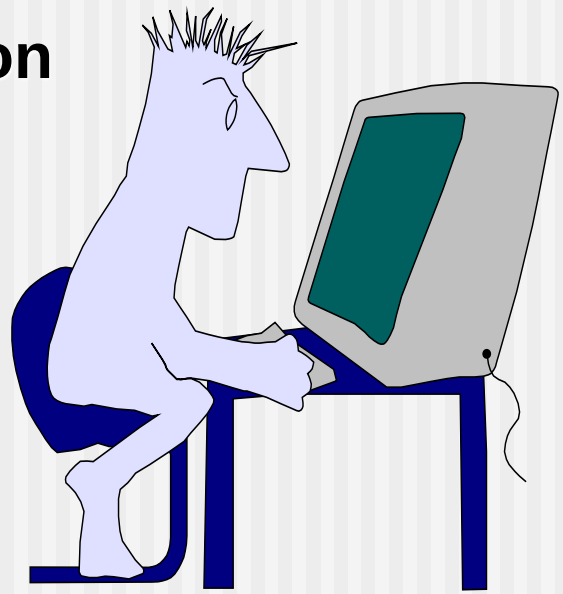
Easy to use?

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.

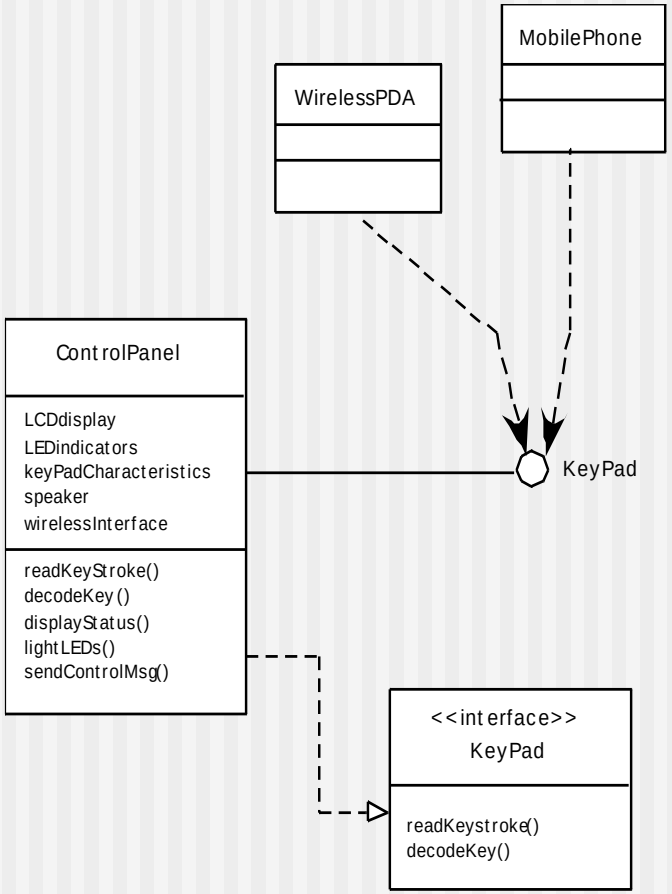
Make the Interface Consistent

- 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

Interface Elements



- **Component Design**

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

What is a Component?

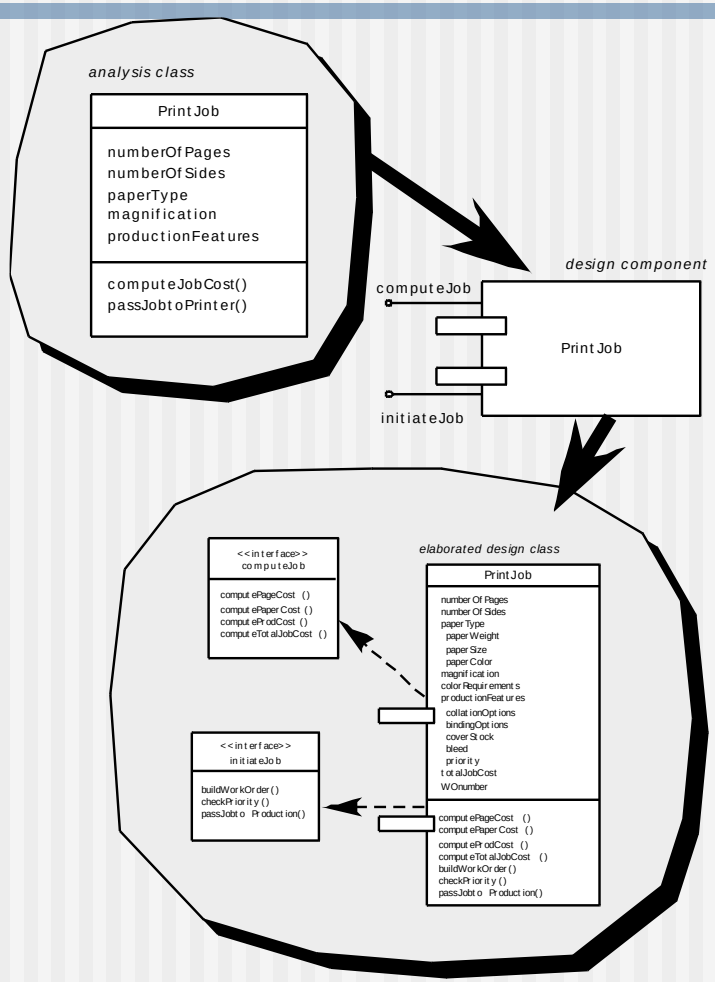
- OMG Unified Modeling Language Specification defines a component as:

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

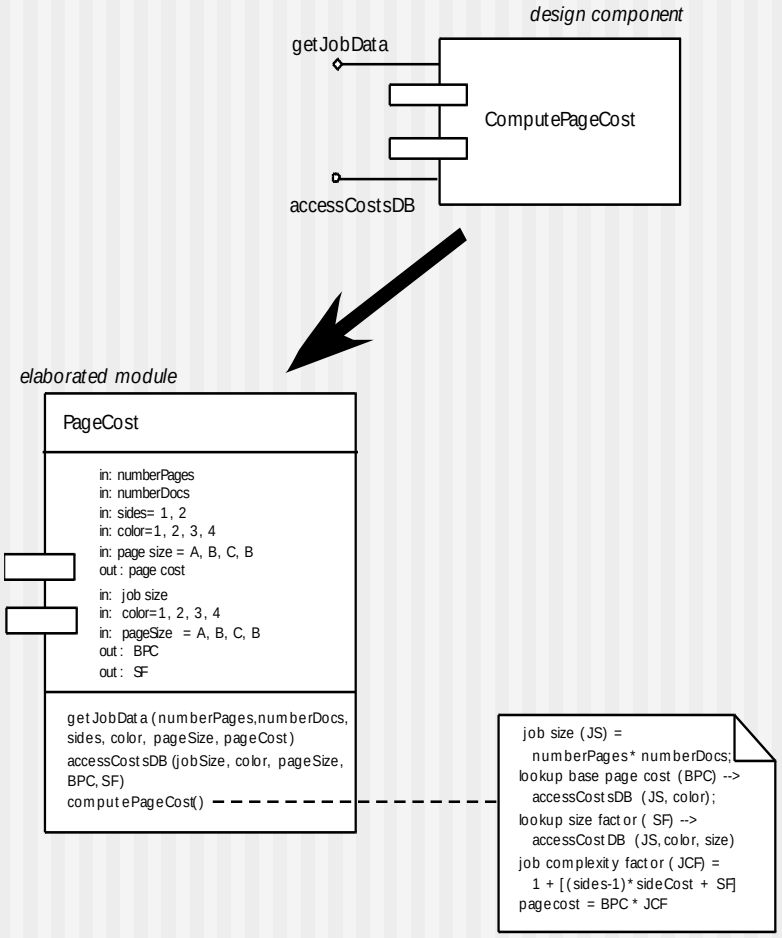
- **OO View** - A component contains a set of collaborating classes.

- **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



Component-Level Design

- **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.
- **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.

Component-Level Design

- **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.

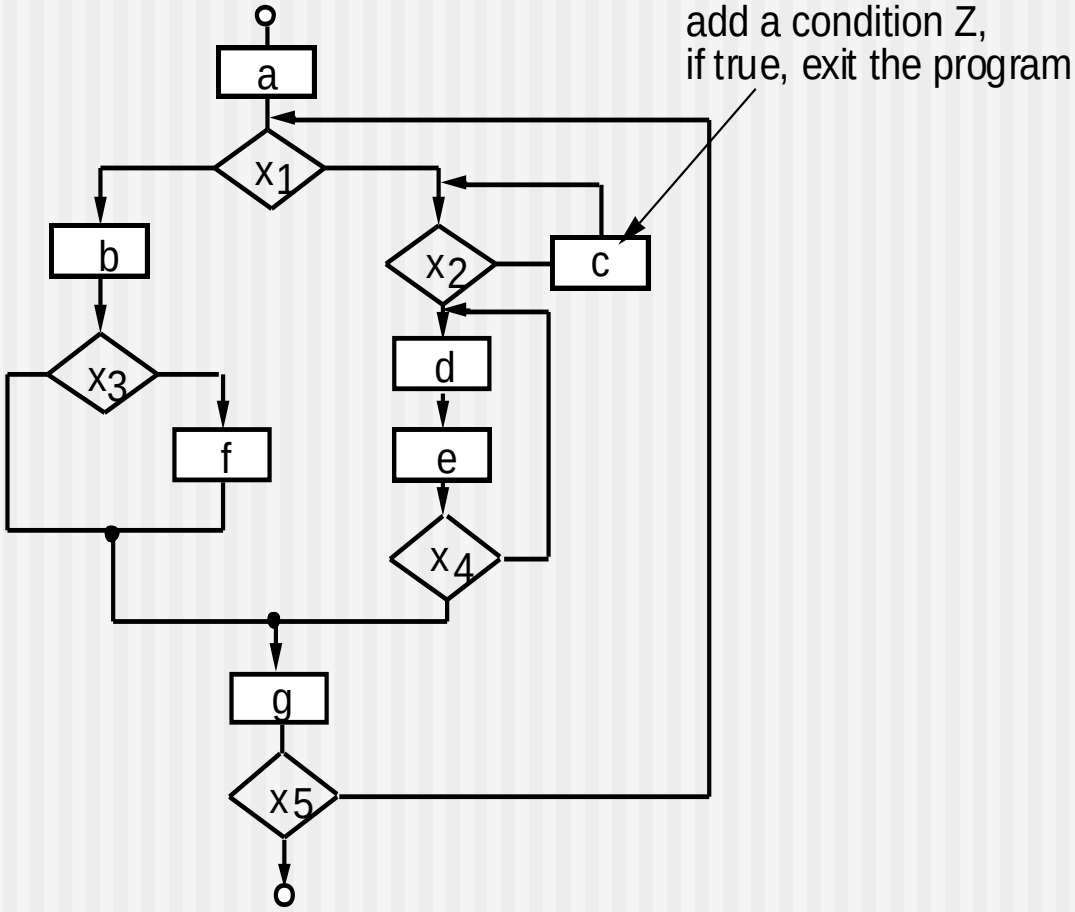
Algorithm Design

- The closest design activity to coding.
- The approach:
  - Review the design description for the component.
  - Use stepwise refinement to develop algorithm.
  - Use structured programming to implement procedural logic.
  - Use ‘formal methods’ to prove logic.

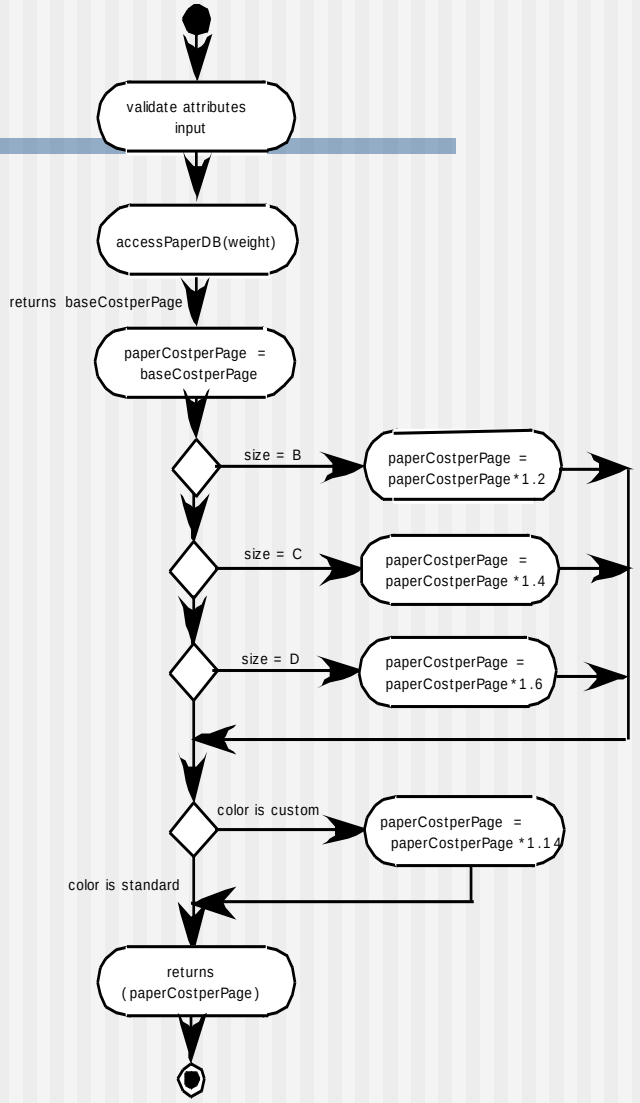
Algorithm Design Model

- Represents the algorithm at a level of detail that can be reviewed for quality.
- Options:
  - **Graphical Design Notation** (e.g., flowchart, box diagram, activity diagram)
  - **Tabular Design Notation** (e.g., decision table)
  - **Program Design Language** (e.g., pseudocode)

A Structured Procedural Design



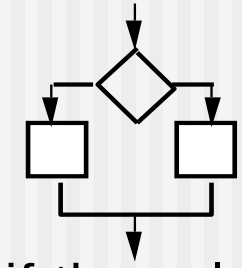
Activity Diagram



Decision Table

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
regular customer	T	T				
silver customer			T	T		
gold customer					T	T
special discount	F	T	F	T	F	T
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

Program Design Language (PDL)



if-then-else

```
if condition x
   then process a;
   else process b;
endif
```

PDL

- easy to combine with source code
- machine readable, no need for graphics input
- graphics can be generated from PDL
- enables declaration of data as well as procedure
- easier to maintain

- **Deployment Design**

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