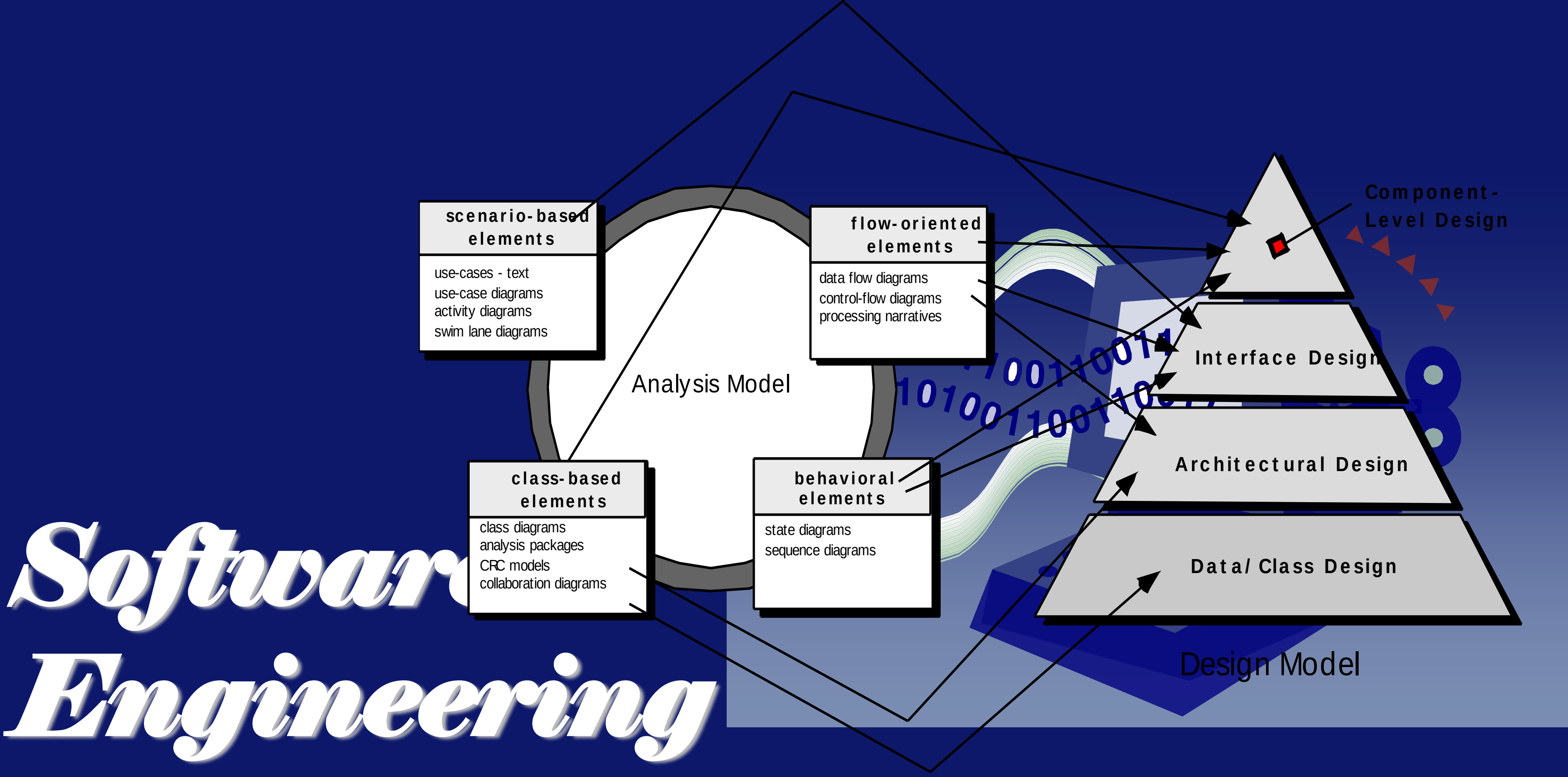
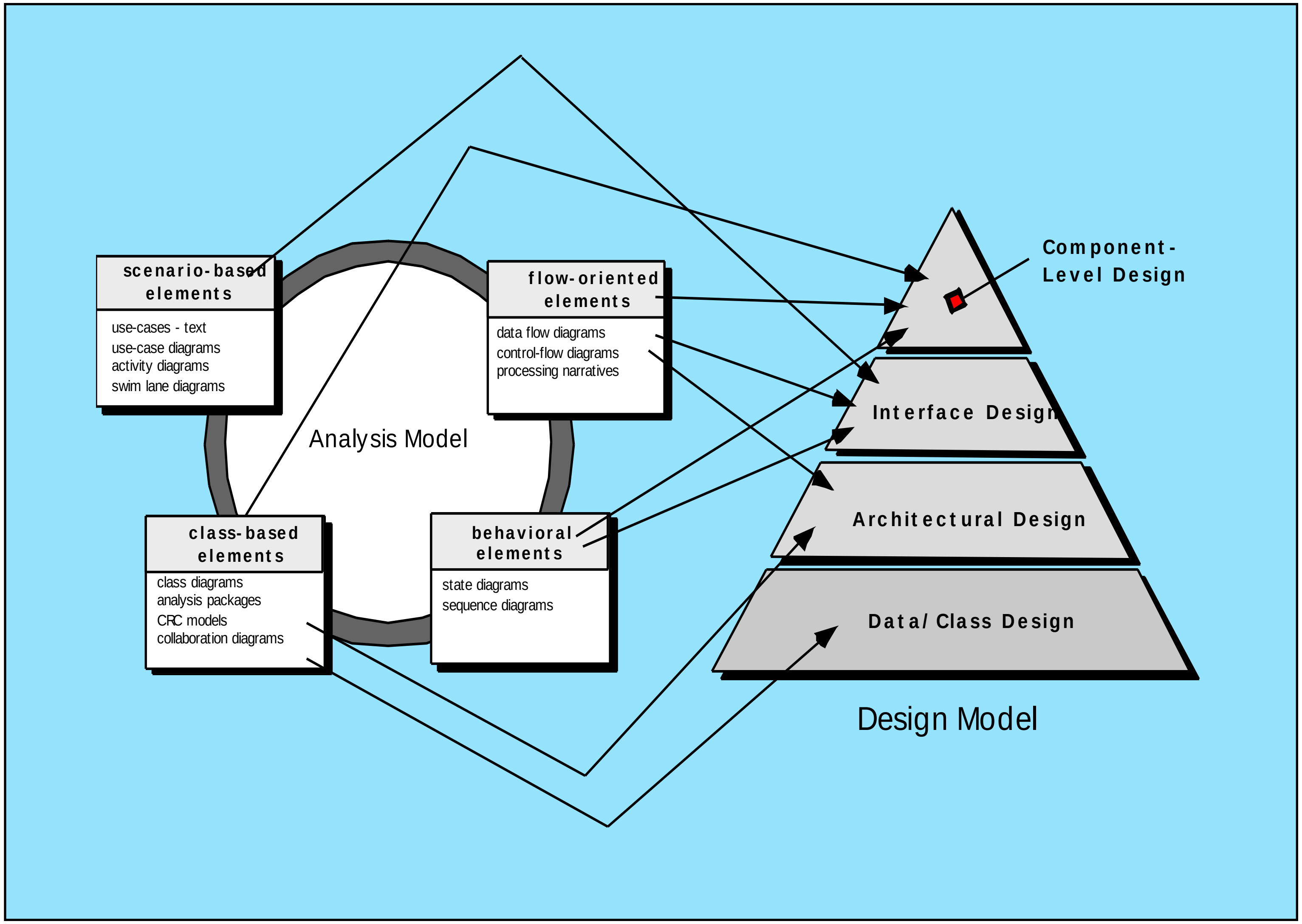


Chapter 9 design Engineering

Moon kun Lee
Division of Electronics & Information Engineering
Chonbuk National University

Contents

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
  - For smaller systems, design can sometimes be developed linearly.
- 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

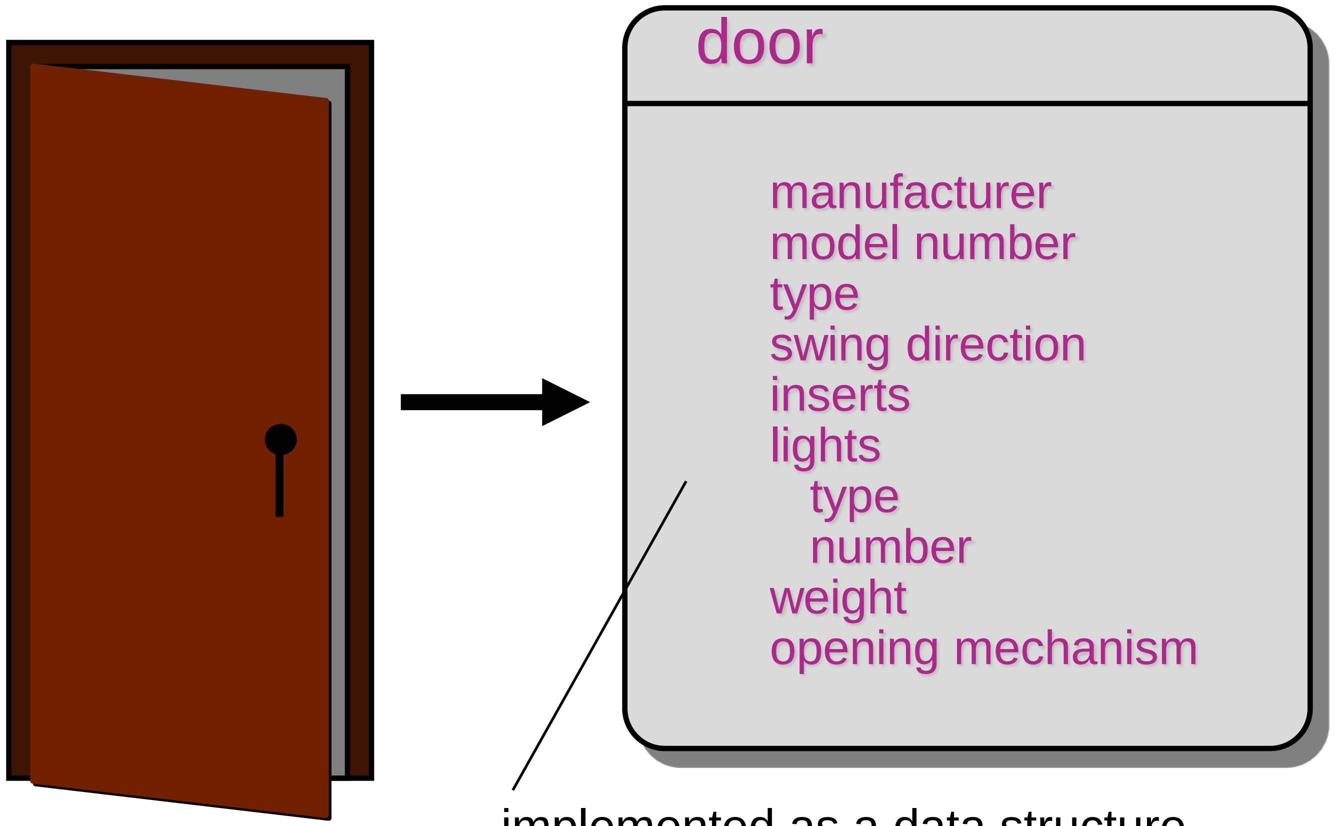
- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The 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

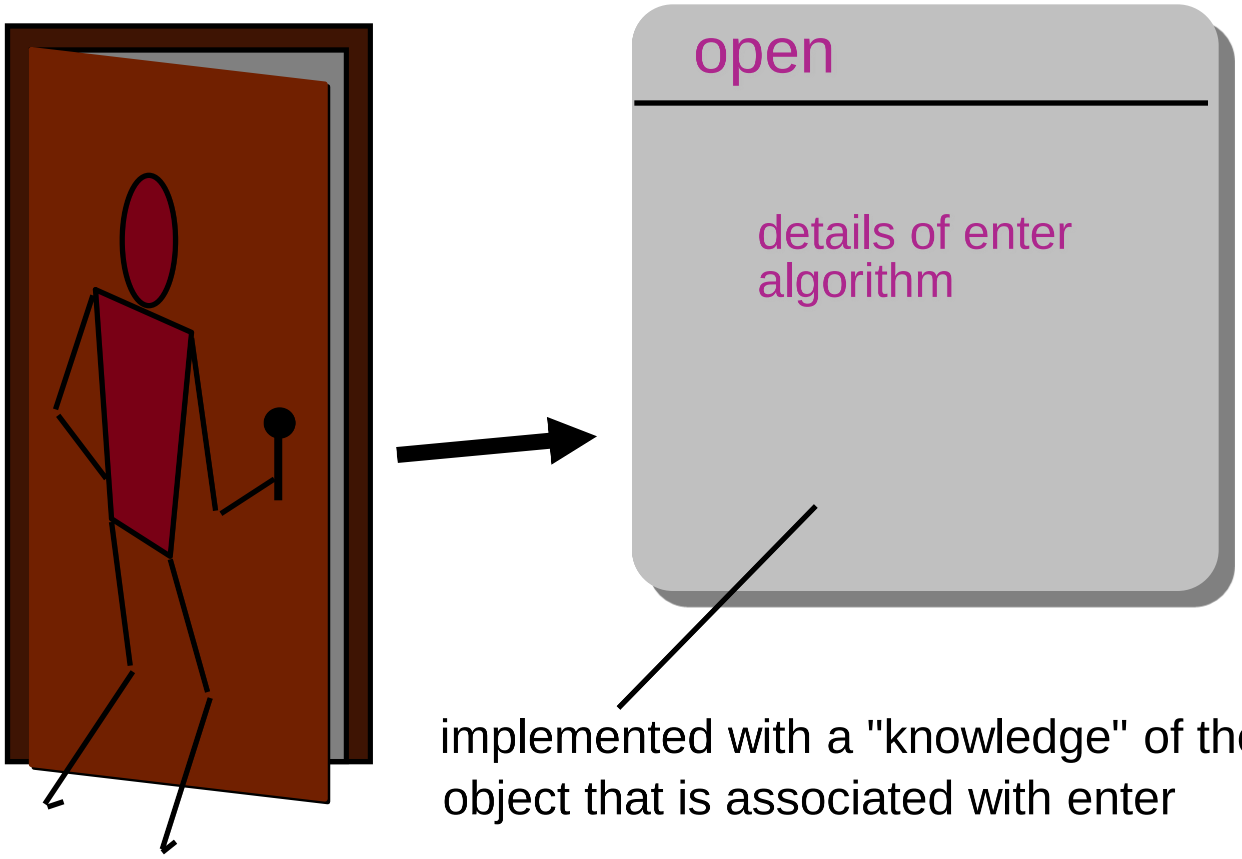
- abstraction—data, procedure, control
- architecture—the overall structure of the software
- patterns—"conveys the essence" of a proven design solution
- modularity—compartmentalization of data and function
- hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- refinement—elaboration of detail for all abstractions
- Refactoring—a reorganization technique that simplifies the design

Data Abstraction



implemented as a data structure

Procedural Abstraction



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

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

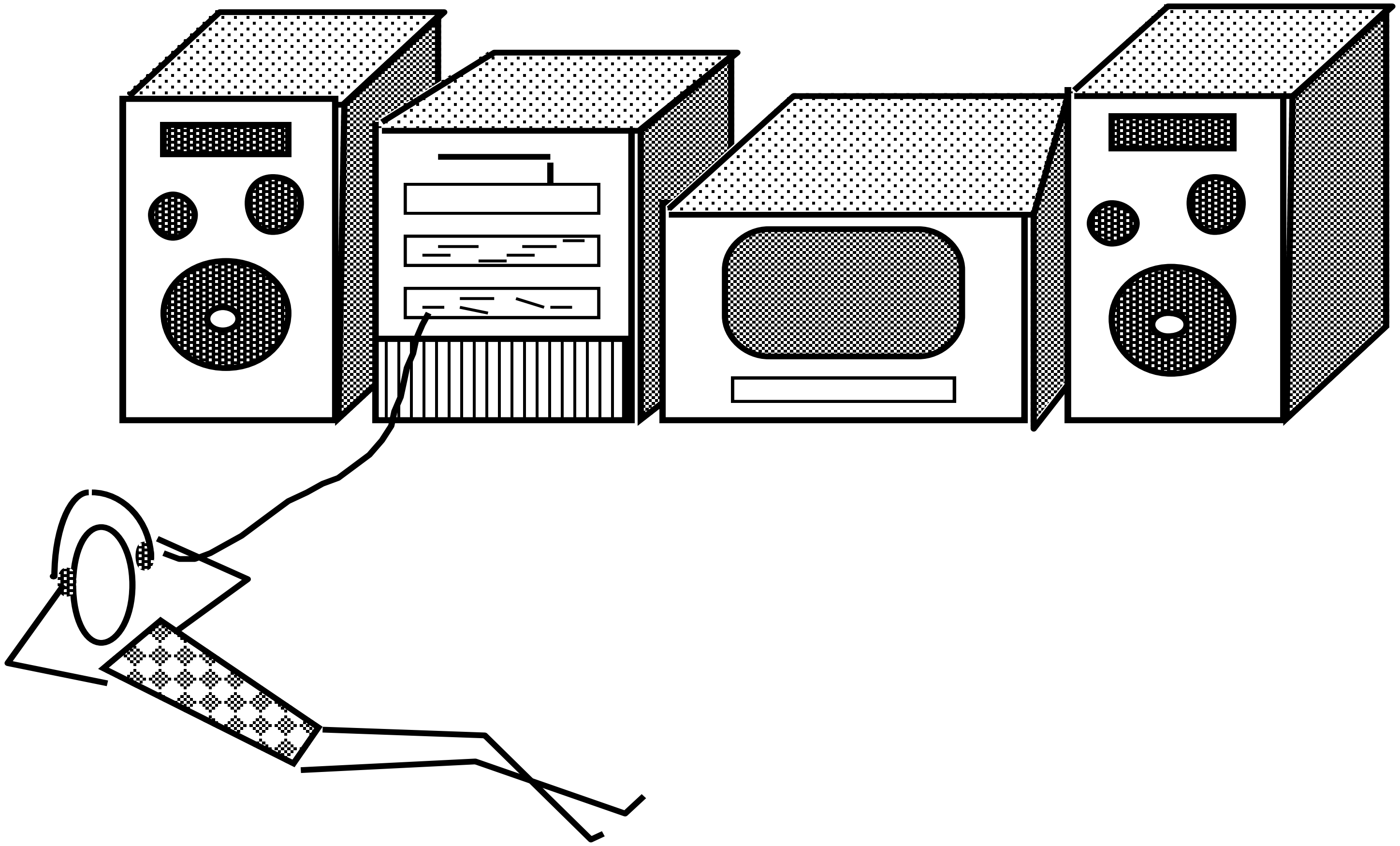
Collaborations : describes how the participants collaborate to carry out their responsibilities

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

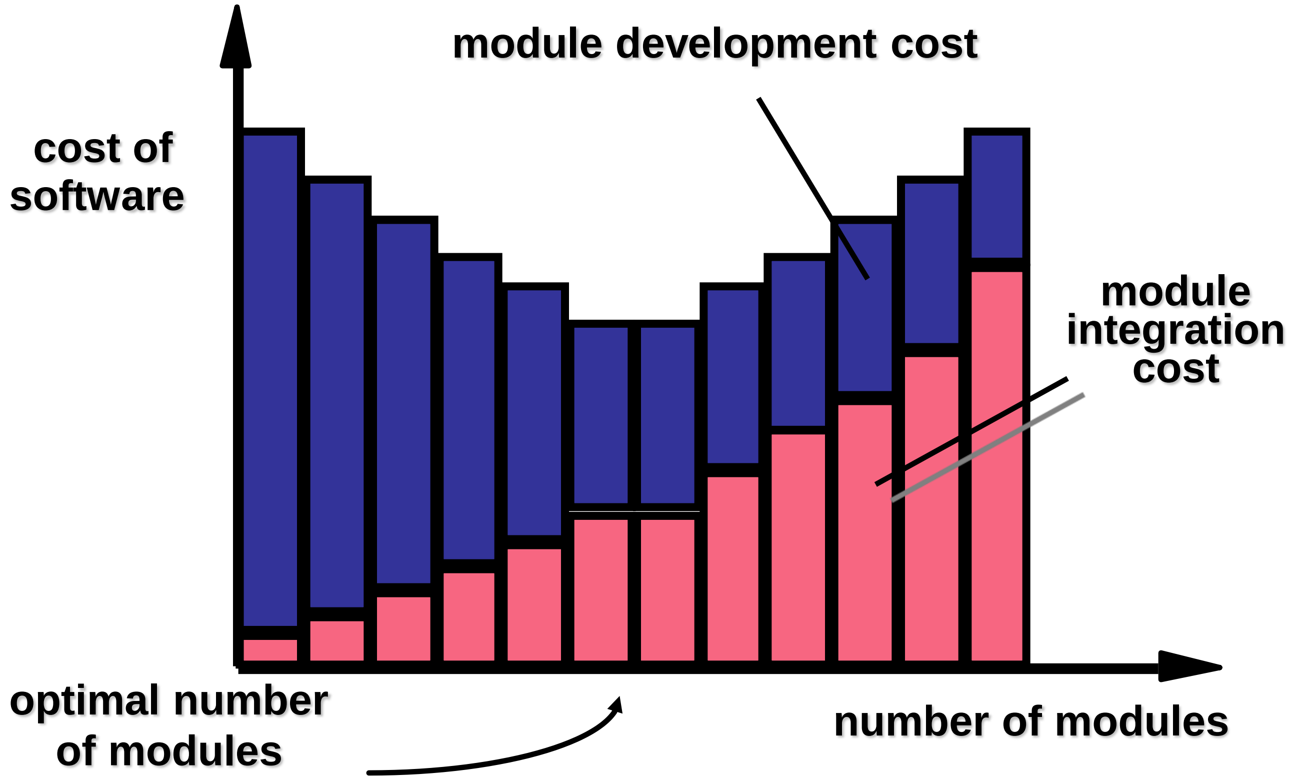
Modular Design

easier to build, easier to change, easier to fix ...

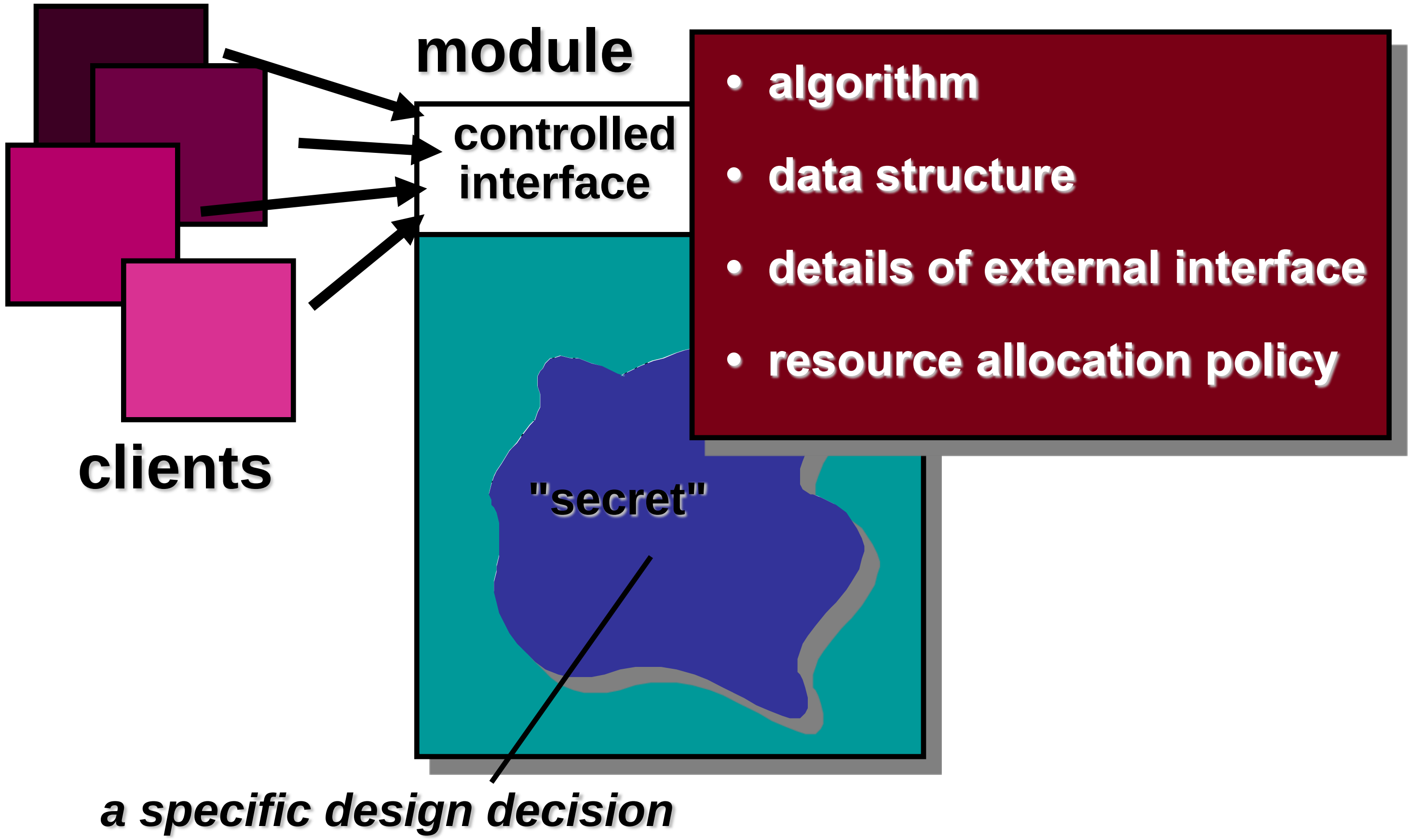


Modularity: Trade-offs

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



Information Hiding

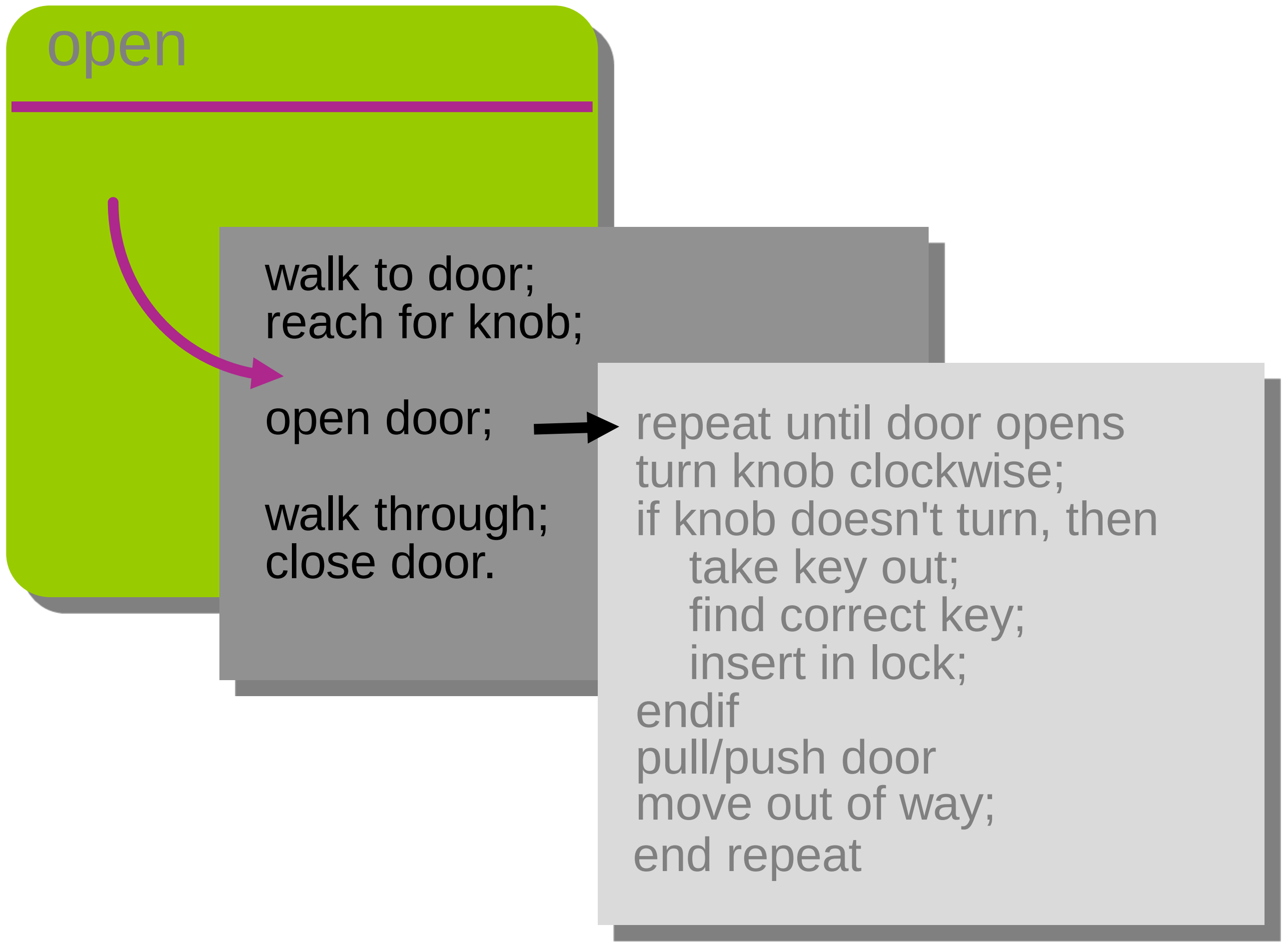


- algorithm
- data structure
- details of external interface
- resource allocation policy

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



```
walk to door;
reach for knob;

open door;

walk through;
close door.
```

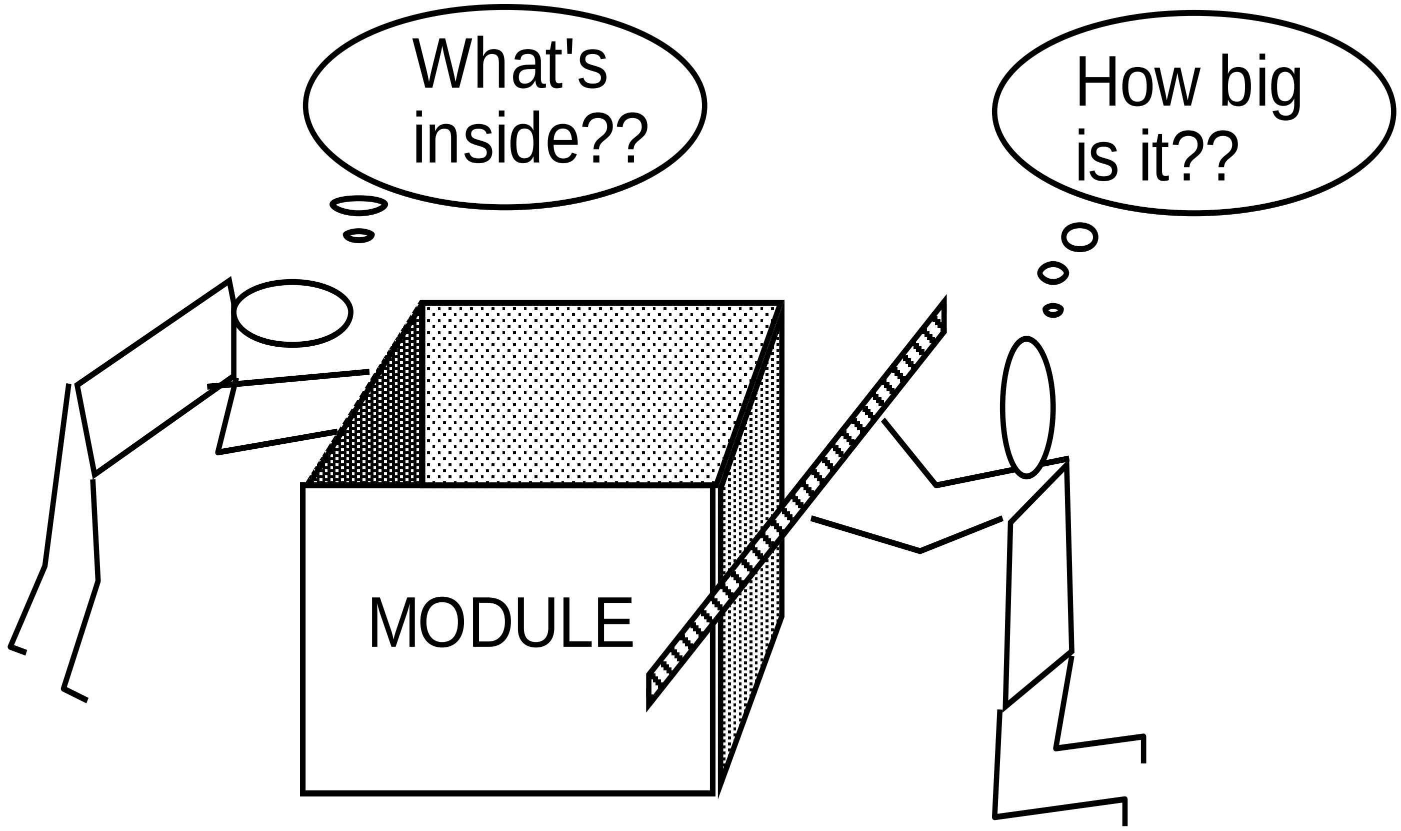
```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Sizing Modules: Two Views



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

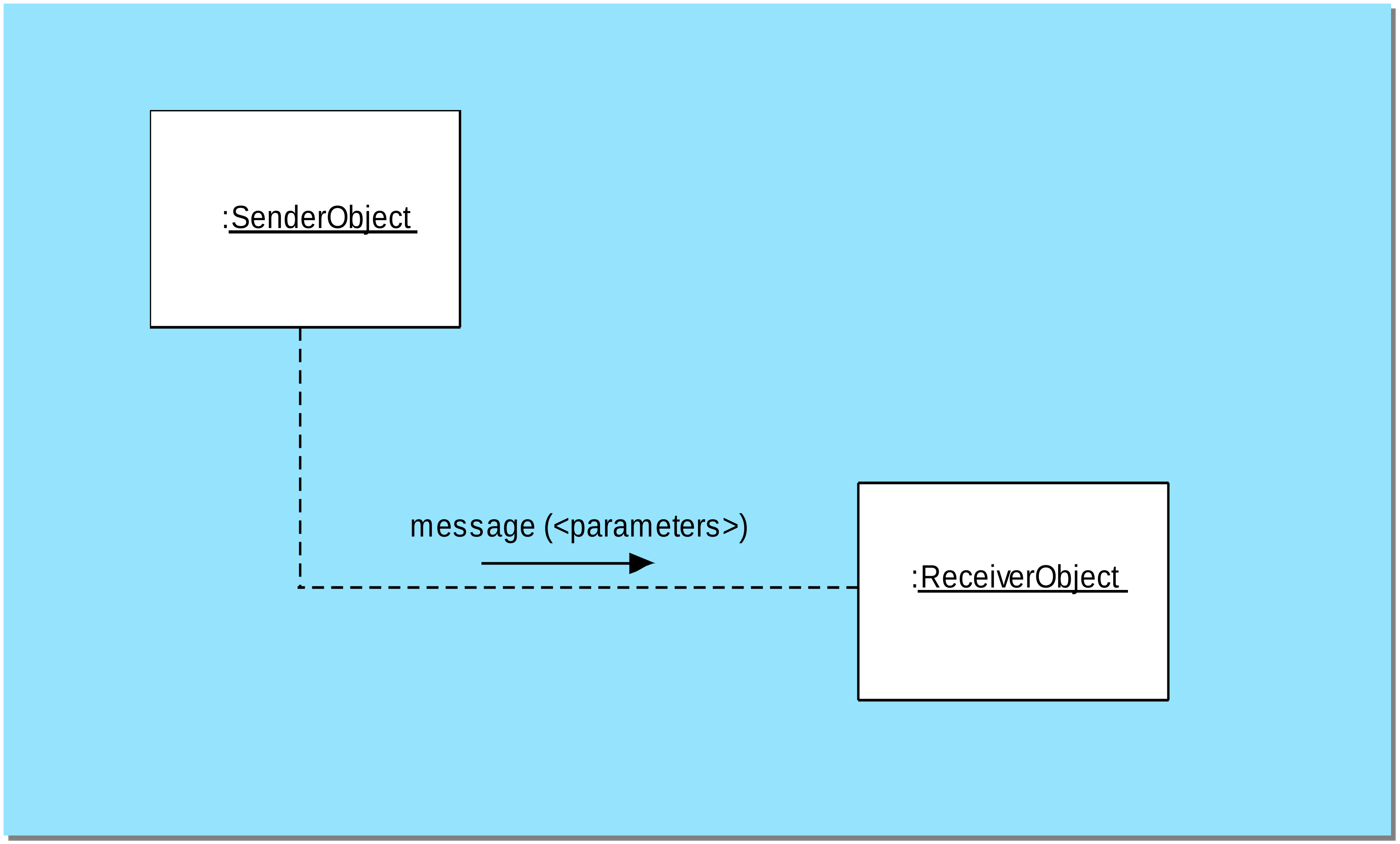
Design Classes

- 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.

Inheritance

- Design options:
  - The class can be designed and built from scratch. That is, inheritance is not used.
  - The class hierarchy can be searched to determine if a class higher in the hierarchy (a superclass)contains most of the required attributes and operations. The new class inherits from the superclass and additions may then be added, as required.
  - The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
  - Characteristics of an existing class can be overridden and different versions of attributes or operations are implemented for the new class.

Messages



Polymorphism

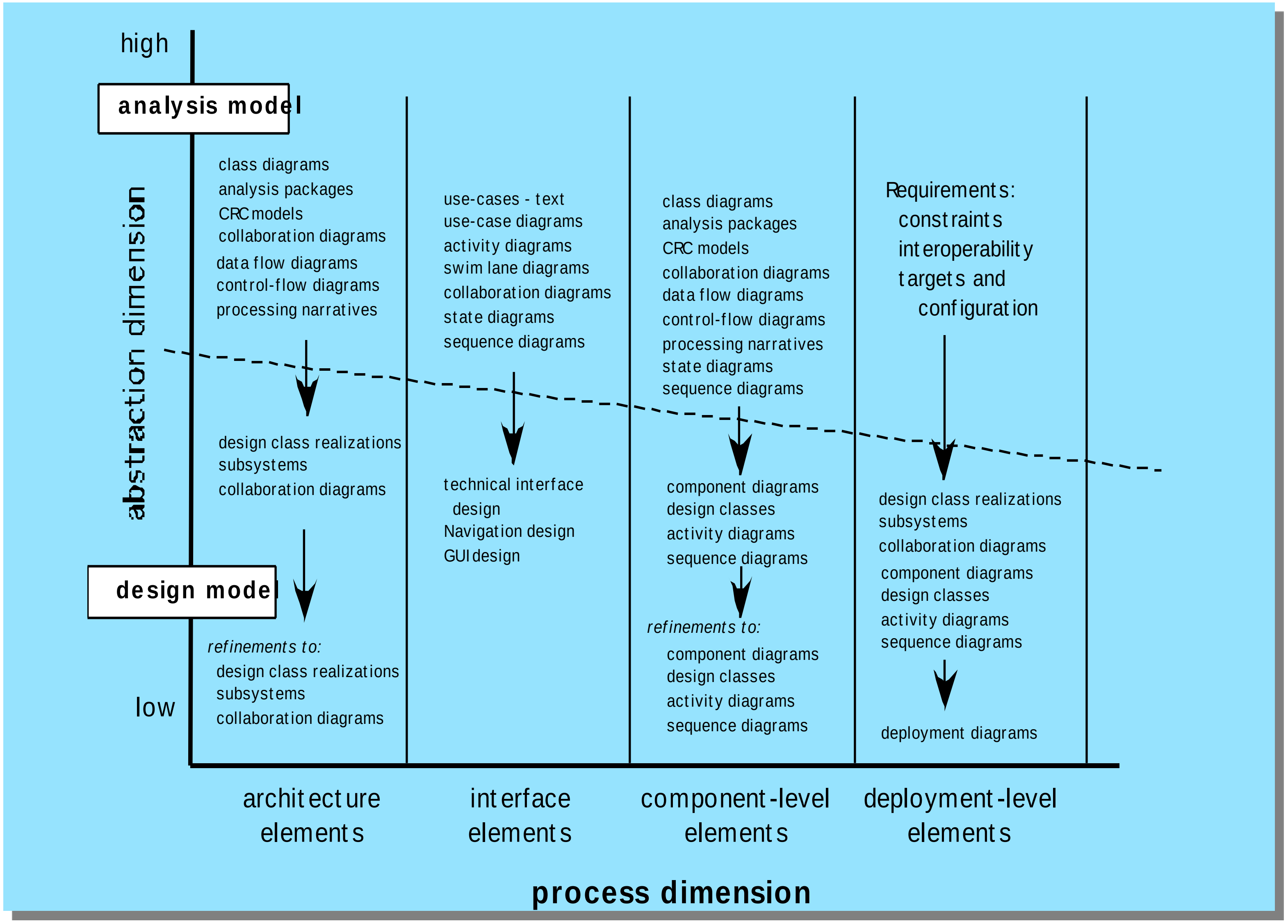
Conventional approach …

```
case of graphtype:
        if graphtype = linegraph then DrawLineGraph (data);
        if graphtype = piechart then DrawPieChart (data);
        if graphtype = histogram then DrawHisto (data);
        if graphtype = kiviat then DrawKiviat (data);
end case;
```

All of the graphs become subclasses of a general class called graph. Using a concept called overloading [TAY90], each subclass defines an operation called *draw*. An object can send a *draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph.

```
graphtype draw
```

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” (Chapter 10)
- 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

Interface Elements

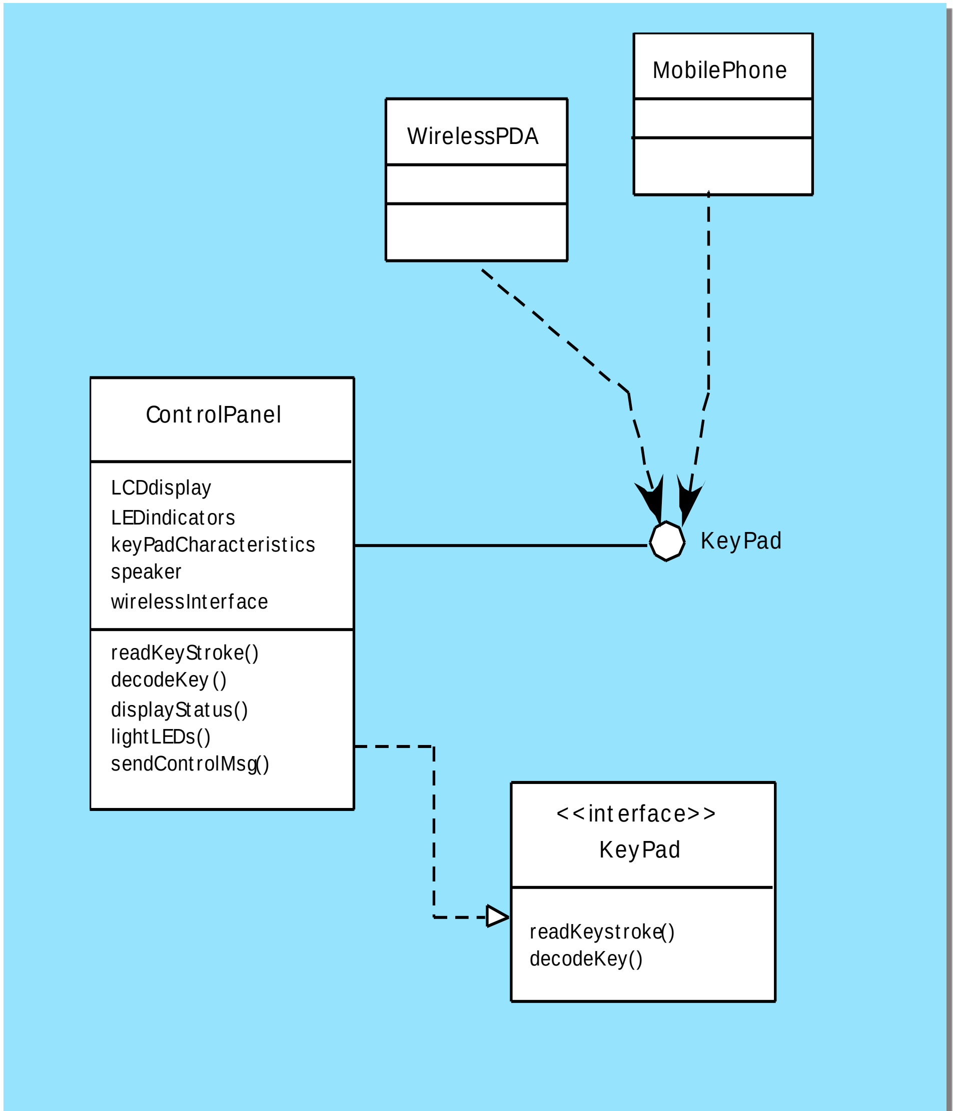
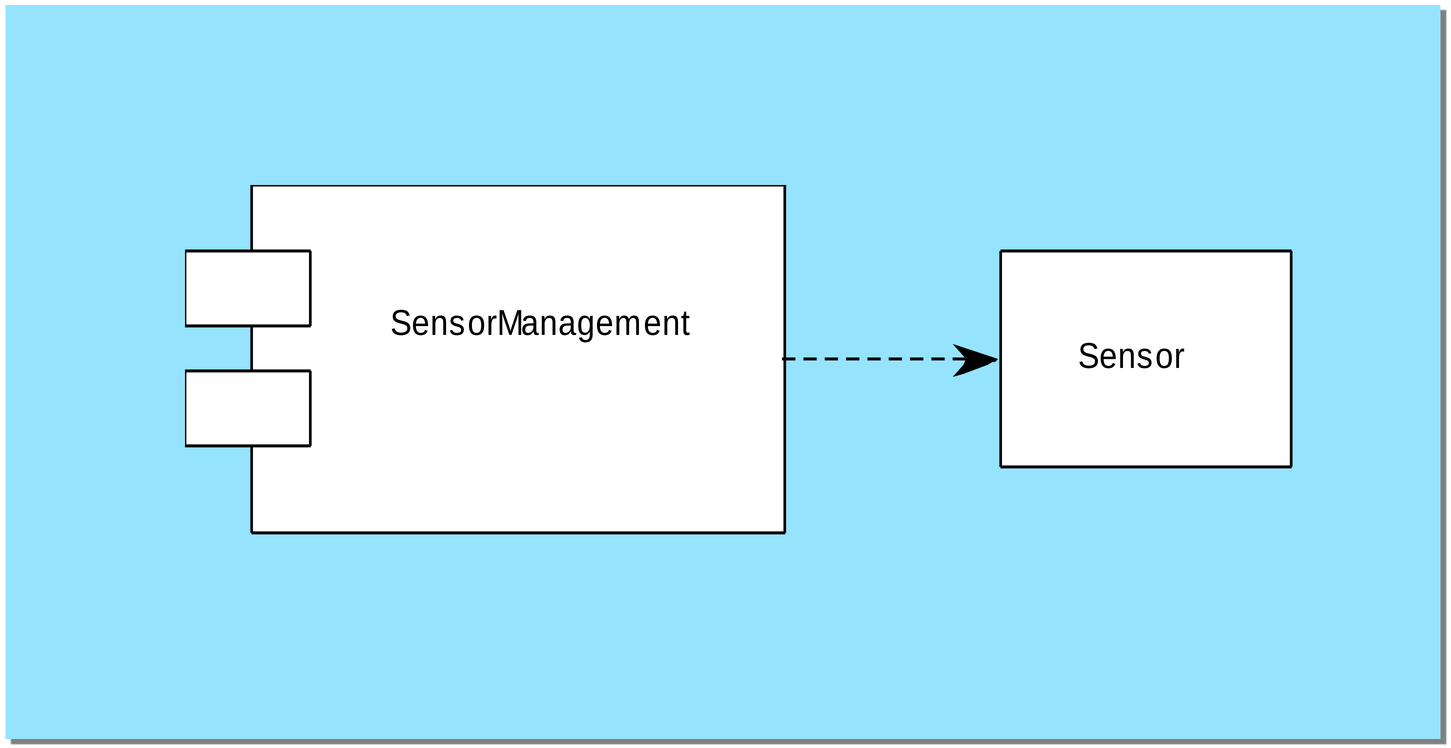


Figure 9.6 UML interface representation for **ControlPanel**

Component Elements



Deployment Elements

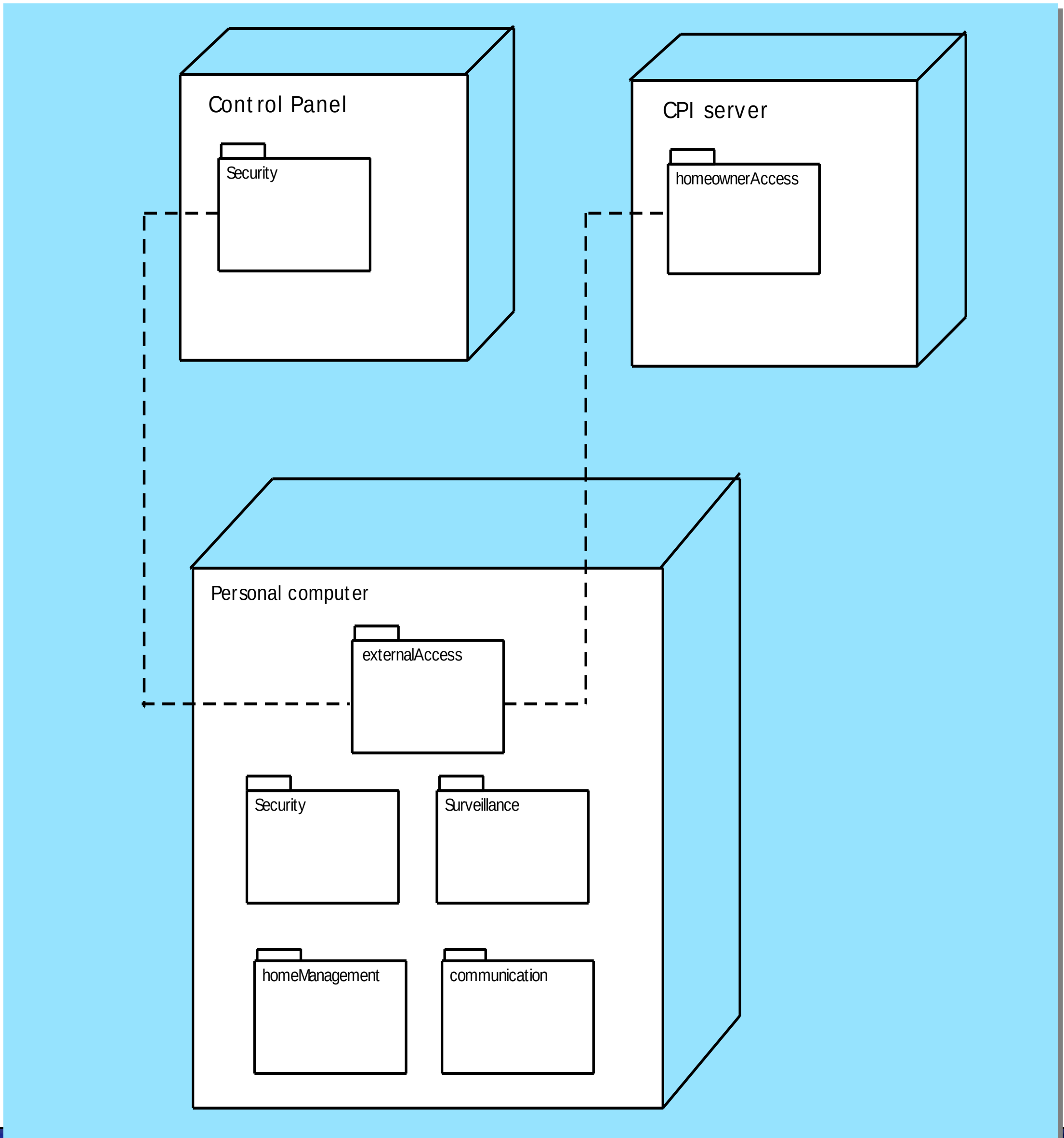


Figure 9.8 UML deployment diagram for *SafeHome*

Design Patterns

- The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution
- A description of a design pattern may also consider a set of design forces.
  - *Design forces* describe non-functional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied.
- The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.

Frameworks

- 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.
- Gamma, et. al. note that:
  - Design patterns are more abstract than frameworks.
  - Design patterns are smaller architectural elements than frameworks
  - Design patterns are less specialized than frameworks