Design Engineering

Purpose of Design

- Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system
- The design model provides detail about the software data structures, architecture, interfaces, and components
- The design model can be assessed for quality and be improved before code is generated and tests are conducted
  - Does the design contain errors, inconsistencies, or omissions?
  - Are there better design alternatives?
  - Can the design be implemented within the constraints, schedule, and cost that have been established?

How to Design

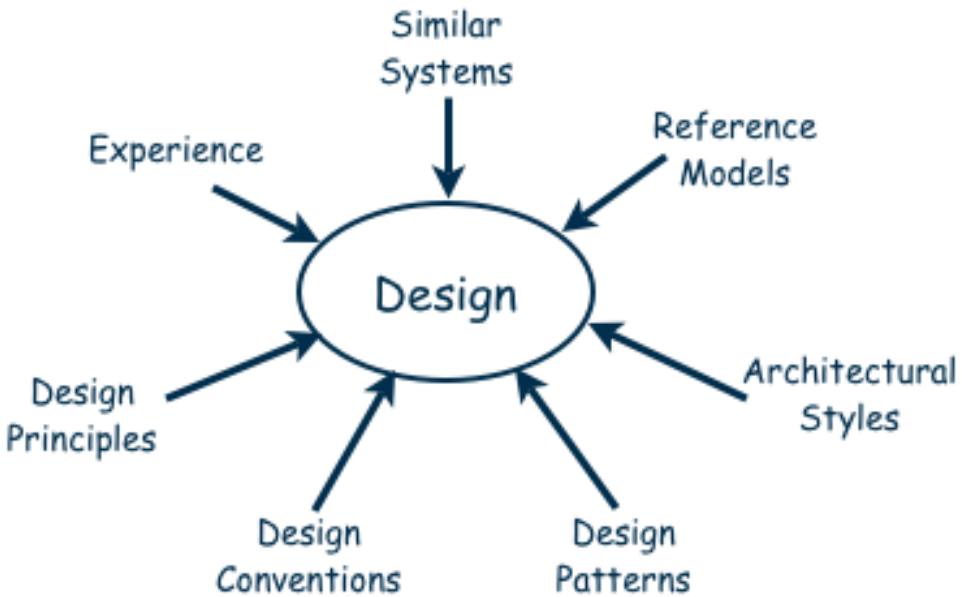
- A designer must practice diversification and convergence -[Belady]
  - The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
  - The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
  - Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen
- Software design is an iterative process through which requirements are translated into a blueprint for constructing the software
  - Design begins at a high level of abstraction that can be directly traced back to the data, functional, and behavioral requirements
  - As design iteration occurs, subsequent refinement leads to design representations at much lower levels of abstraction

The Design Process

- Design is an intellectually challenging task
  - Numerous possibilities the system must accommodate
  - Nonfunctional design goals (e.g., ease of use, ease to maintain)
  - External factors (e.g., standard data formats, government regulations)
- We can improve our design by studying examples of good design. Many ways to leverage existing solutions
  - Cloning: Borrow design/code in its entirety, with minor adjustments
  - Reference models: Generic architecture that suggests how to decompose the system

The Design Process

Design is creative, but so many things influence it

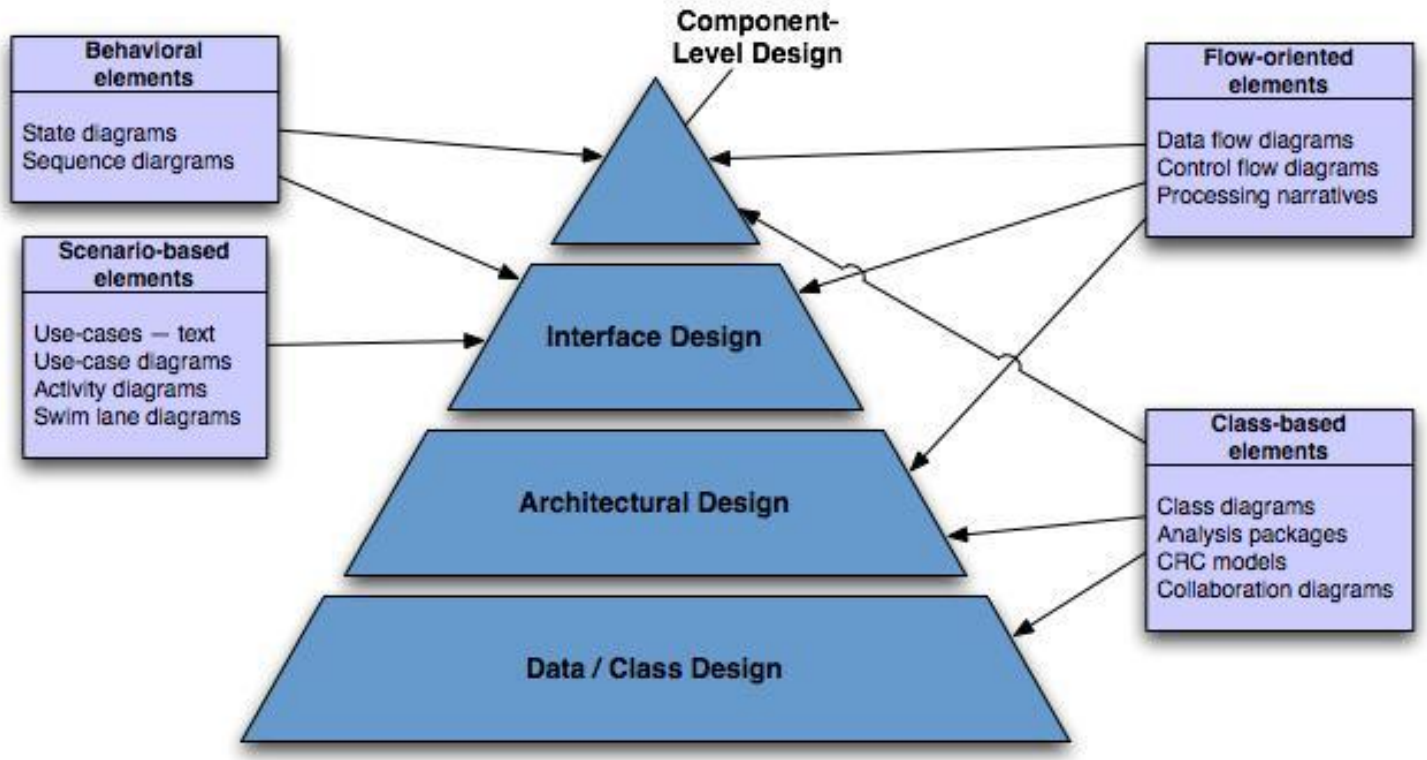


Analysis Model to Design Model

Each element of the analysis model provides information that is necessary to create the four design models

- The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
- The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
- The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

Analysis → Design



Wisdom of Design

- "Questions about whether design is necessary or affordable are quite beside the point; design is inevitable. The alternative to good design is bad design, [rather than] no design at all." **Douglas Martin (Author)**
- "You can use an eraser on the drafting table or a sledge hammer on the construction site." **Frank Lloyd Wright (Architect, Structural Designer)**
- "The public is more familiar with bad design than good design. If is, in effect, conditioned to prefer bad design, because that is what it lives with; the new [design] becomes threatening, the old reassuring." **Paul Rand (Graphic Designer)**
- "A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools." **Douglas Adams (Author Science Fiction)**
- "Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen." **Leonardo DaVinci (Genius)**

“Software Design Manifesto”

Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal.* He applied ideas of Roman architecture critic Vitruvius over software design. According to him, 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.

Task Set for Software Design

1) Examine the information domain model and design appropriate data structures for data objects and their attributes
2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software
3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture
   a) Design the subsystem interfaces
   b) Allocate analysis classes or functions to each subsystem
4) Create a set of design classes or components
   a) Translate each analysis class description into a design class
   b) Check each design class against design criteria; consider inheritance issues
   c) Define methods associated with each design class
   d) Evaluate and select design patterns for a design class or subsystem

Task Set for Software Design (continued)

5) Design any interface required with external systems or devices

6) Design the user interface

7) Conduct component-level design
   a) Specify all algorithms at a relatively low level of abstraction
   b) Refine the interface of each component
   c) Define component-level data structures
   d) Review each component and correct all errors uncovered

8) Develop a deployment model
   - Show a physical layout of the system, revealing which components will be located where in the physical computing environment

Fundamental Design Concepts

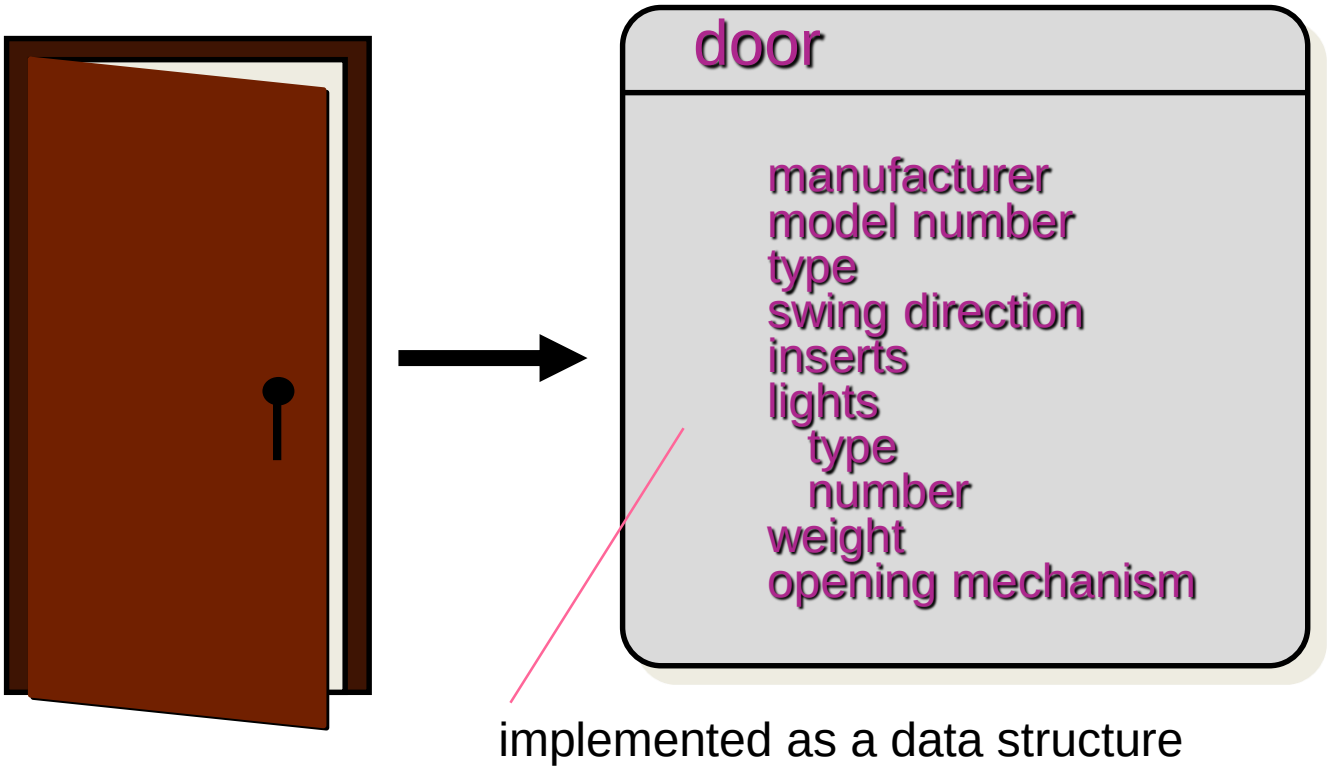
- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Information 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

The beginning of wisdom (for a software engineer) is to recognize the difference between getting program to work, and getting it right *– M A Jackson*

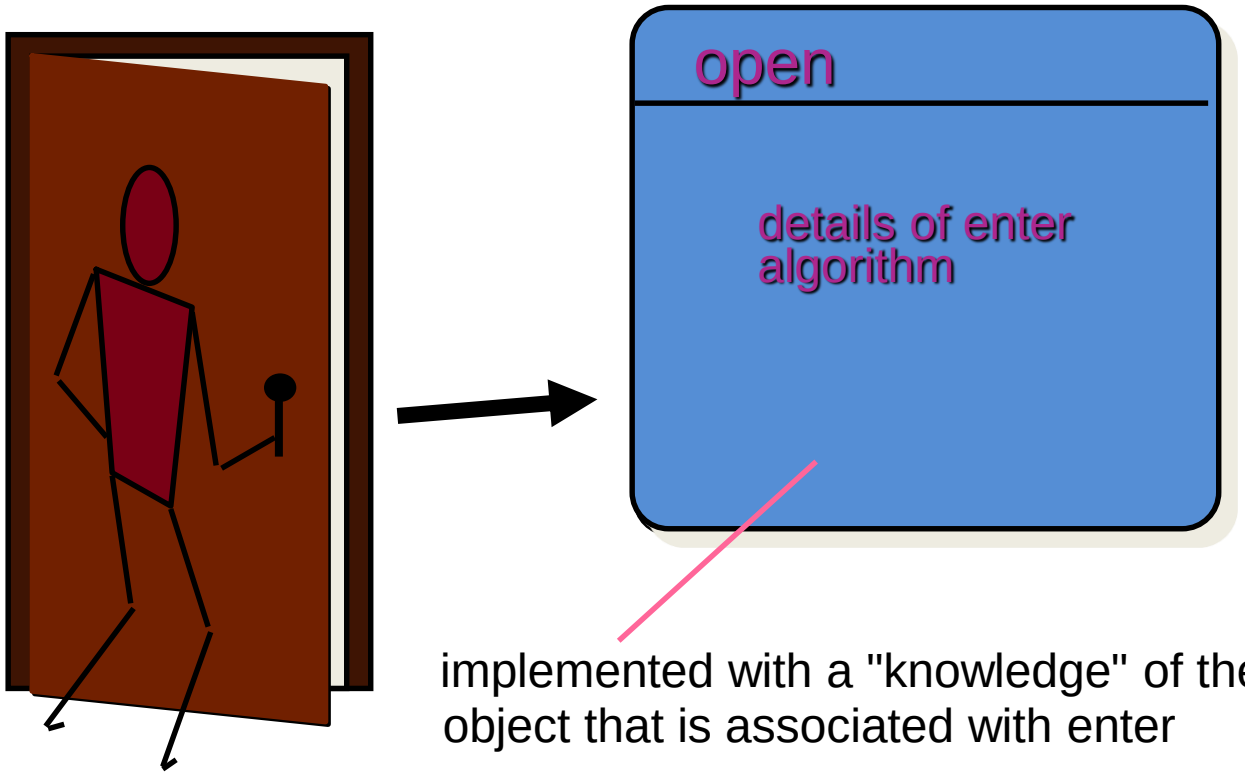
Abstraction et al.

- Abstraction
  - process – extracting essential details
  - entity – a model or focused representation
  - Enablers
    - Information hiding
      - the suppression of inessential information
    - Encapsulation
      - process – enclosing items in a container
      - entity – enclosure that holds the items

Data Abstraction



Procedural Abstraction



Architectural Elements

- The architectural model 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 and styles.

– Shaw & Garlan

Architecture

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system." - Shaw & Garlan

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

- An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- A **design pattern** provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- An **idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

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.

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

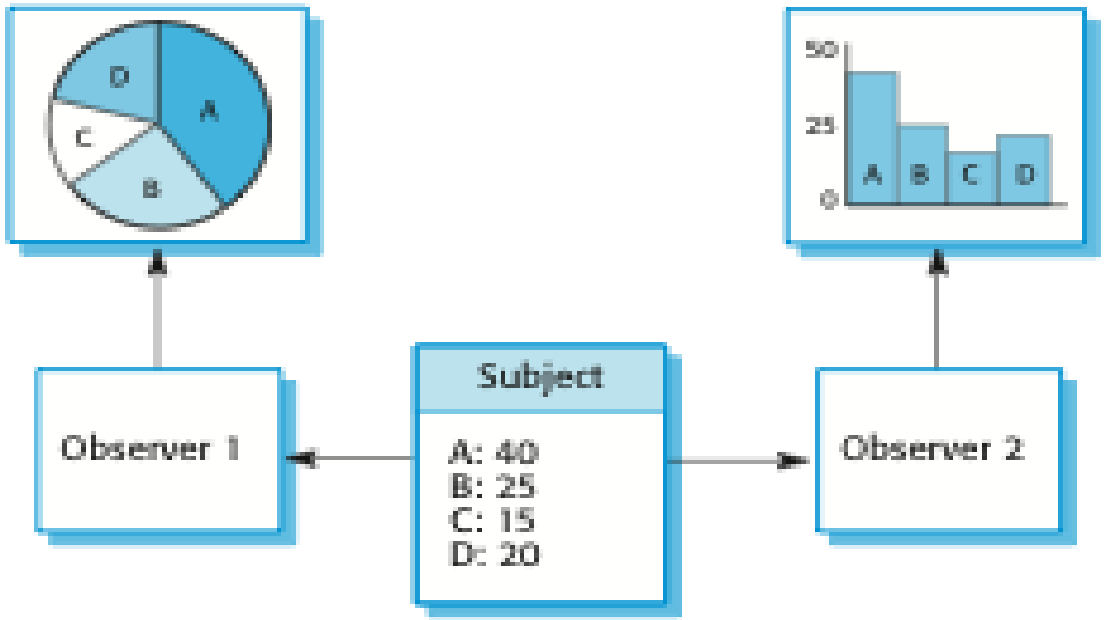
Consequences **—** describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

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

The Observer pattern

Pattern name	Observer
Intent	Separates the display of the state of an object from the object itself and allows alternative displays to be provided.
Motivation	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified
Applicability	This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Structure	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations.
Participants & Collaborations	The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Multiple displays using the Observer pattern



Separation of Concerns - Dijkstra

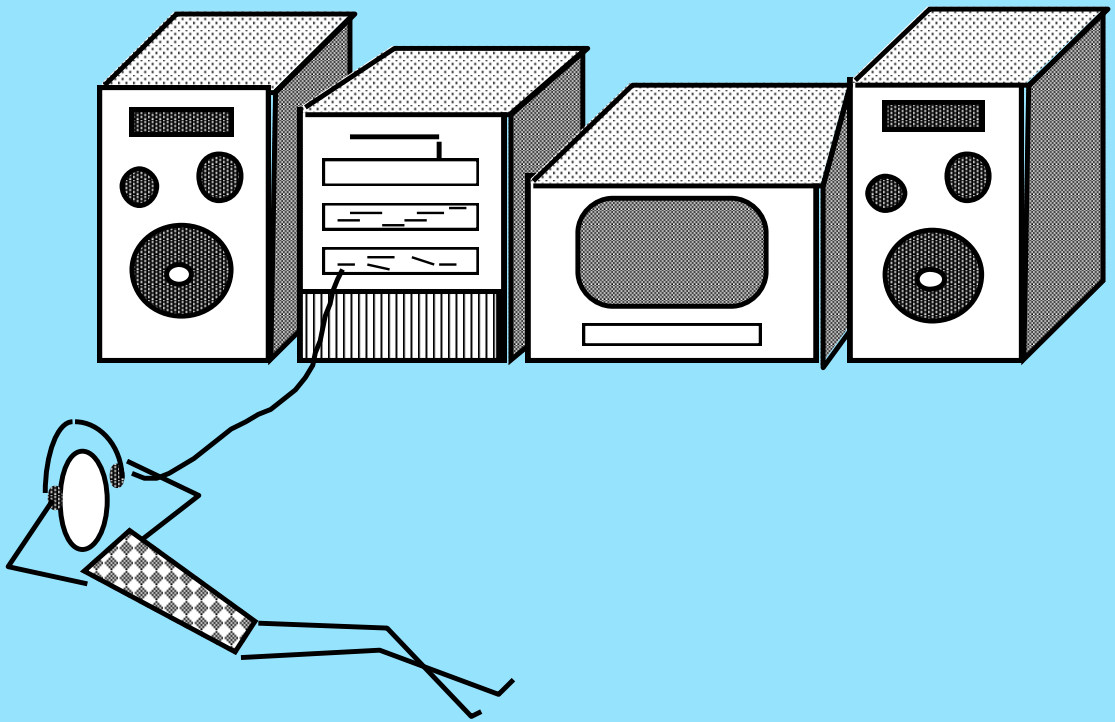
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more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

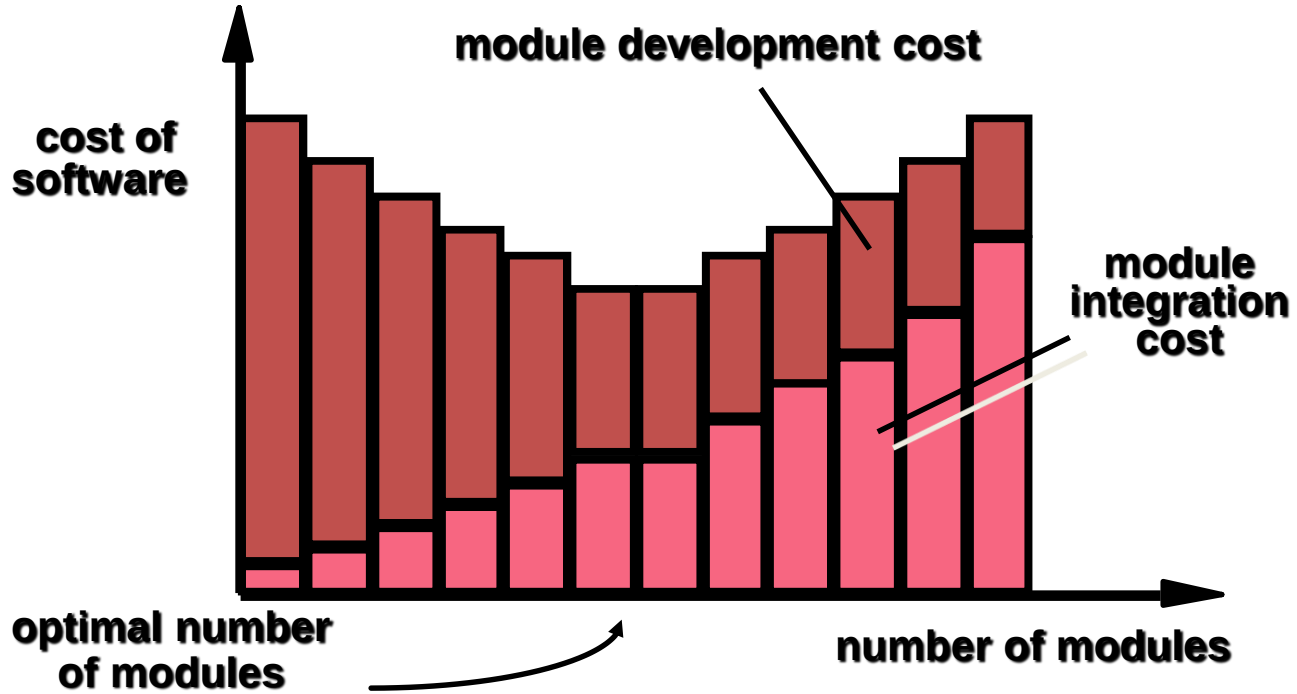
Modular Design

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



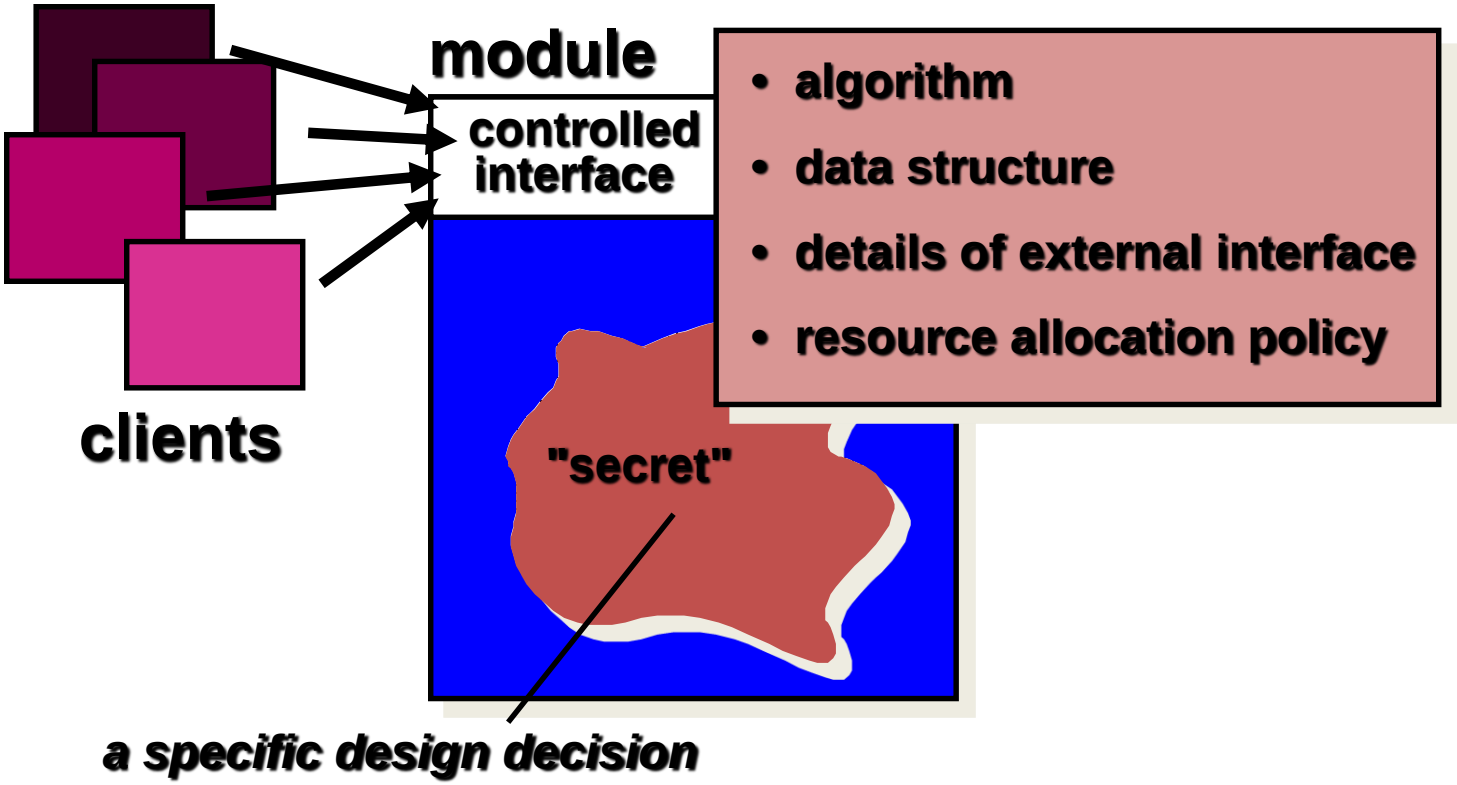
Modularity: Trade-offs

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



Information Hiding

Responds to “How do I decompose software solution to obtain best set of modules



Modules be “characterized by design decisions that hides from all others” - Parnas

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

Functional Independence

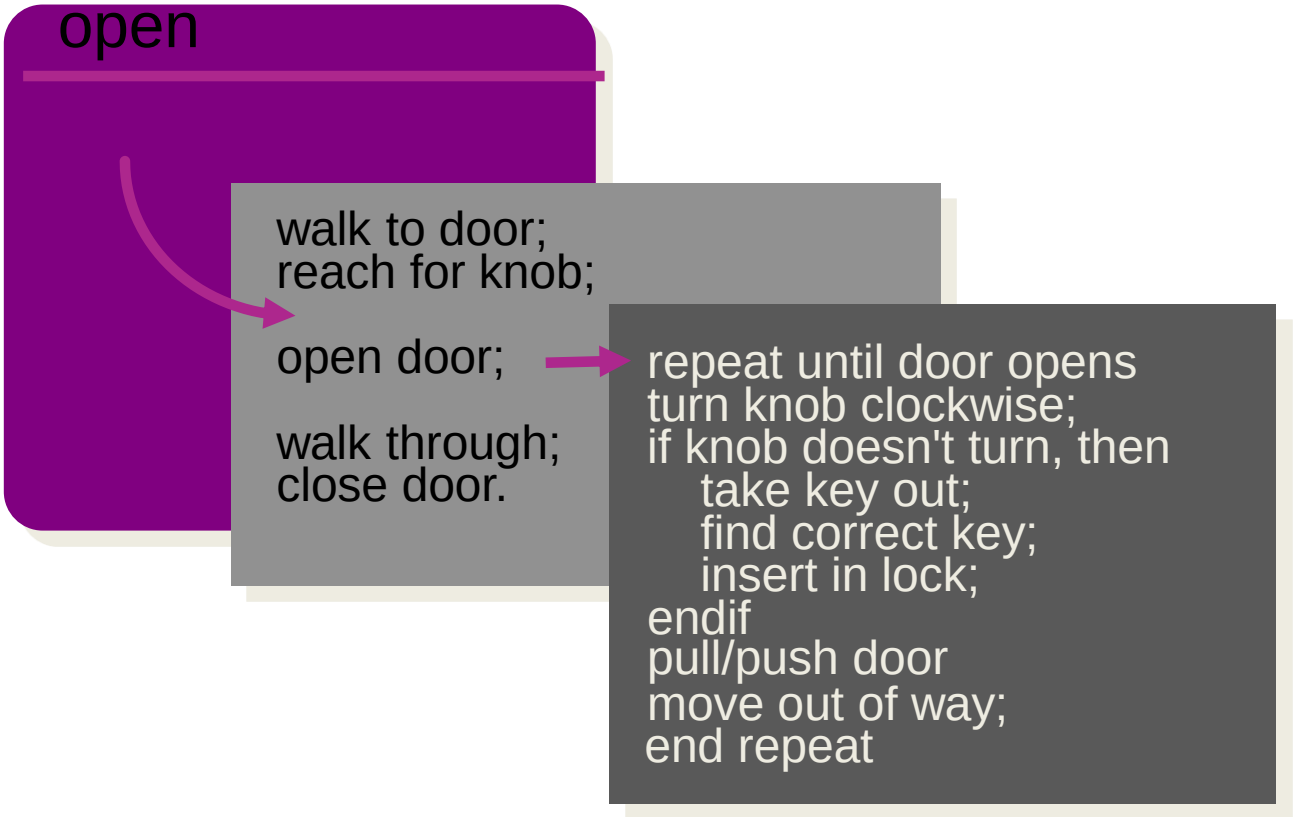
- Functional independence is achieved by developing modules with "**single-minded**" function and an "**aversion**" to excessive interaction with other modules.
- ***Cohesion*** is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little 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.

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.

Stepwise Refinement



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”

  - Rosenhainer

- 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

- Analysis classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass are 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

Analysis/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.

A Taxonomy of Design Classes

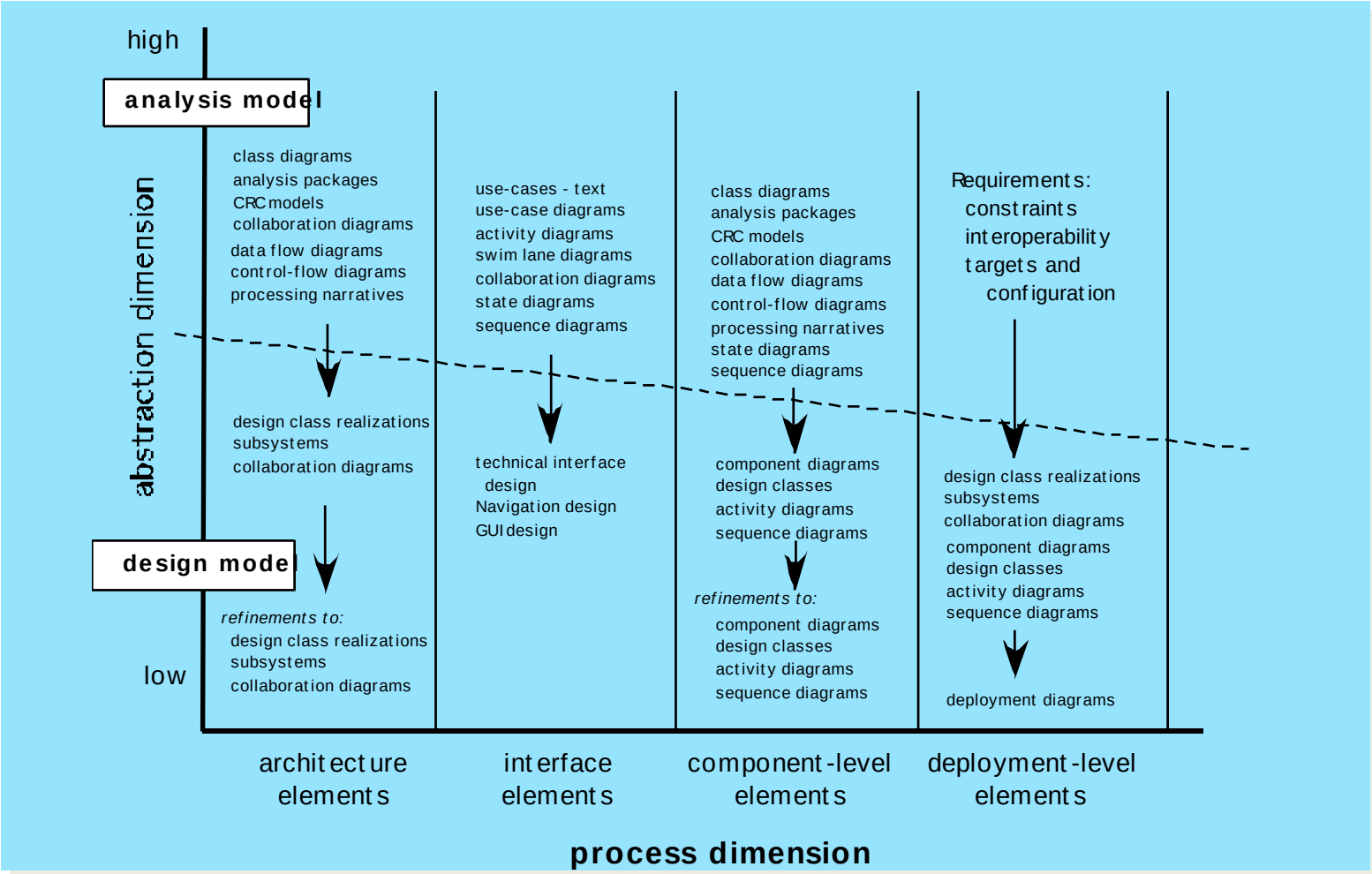
Scott Ambler classifies design classes as per the layers of architecture

- User interface classes – define abstractions necessary for Human-Computer interface.
- Business domain classes – refinements of analysis classes.
- Process classes – lower-level business abstractions that manage business domain classes.
- Persistent classes – data stores (databases) that persist beyond execution of the software.
- System classes – management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Well-formed Design Class

- **Complete and sufficient** – class should be a complete and sufficient encapsulation of reasonable attributes and methods.
- **Primitiveness** – each method should be focused on one thing, e.g setStartPoint()
- **High cohesion** – class should be focused on one kind of thing.
- **Low coupling** – collaboration should be kept to an acceptable minimum.

The Design Model



Design Model Elements

- **Data elements**
  - Architectural level → databases and files
  - Component level → data structures
- **Architectural elements**
  - An architectural model is derived from:
    - Application domain
    - Analysis model
    - Available styles and patterns
- **Interface elements**
  - There are three parts to the interface design element:
  - The user interface (UI)
  - Interfaces to external systems
  - Interfaces to components within the application
- **Component elements**
- **Deployment elements**

Interface Elements

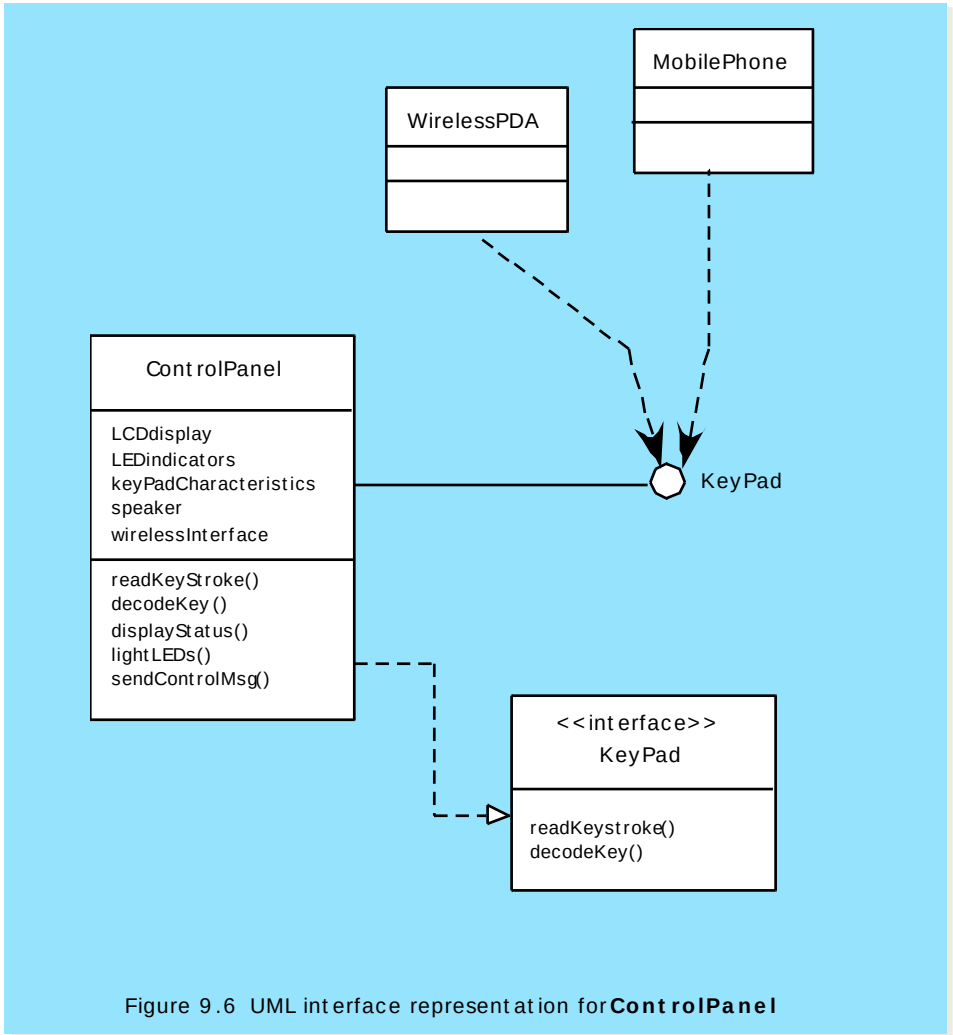
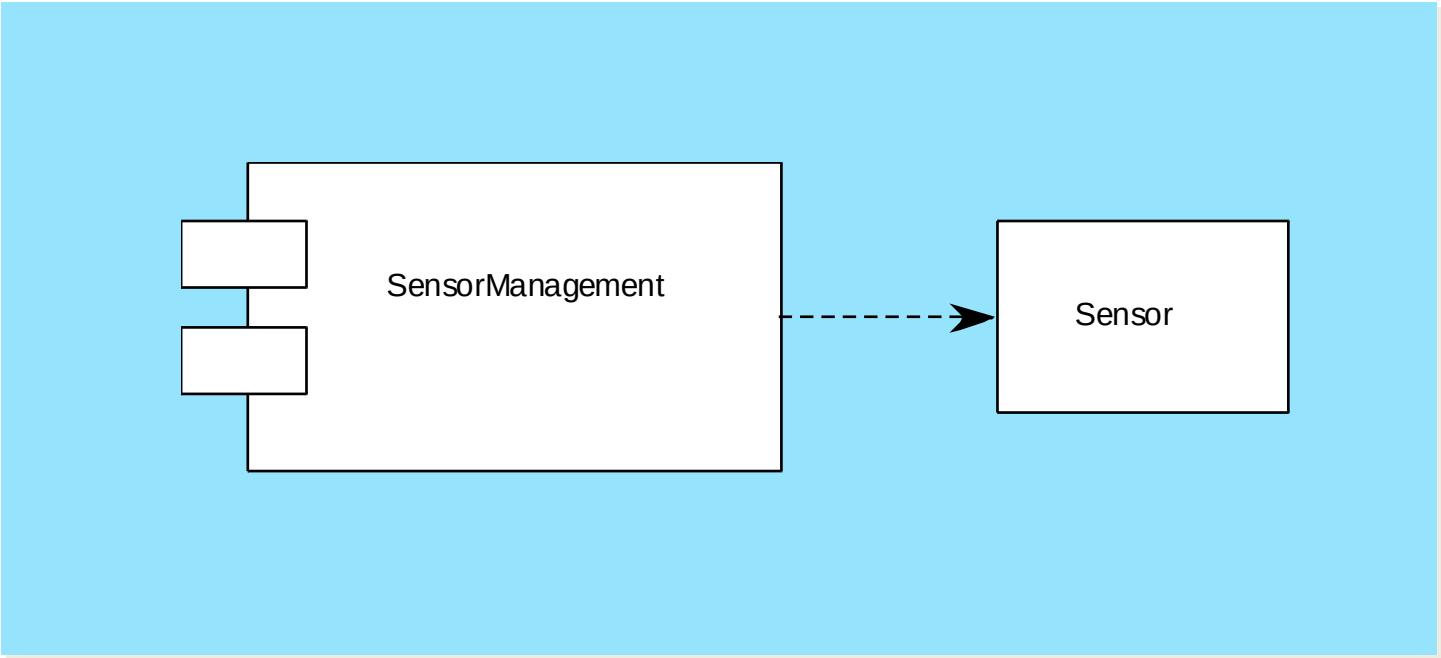
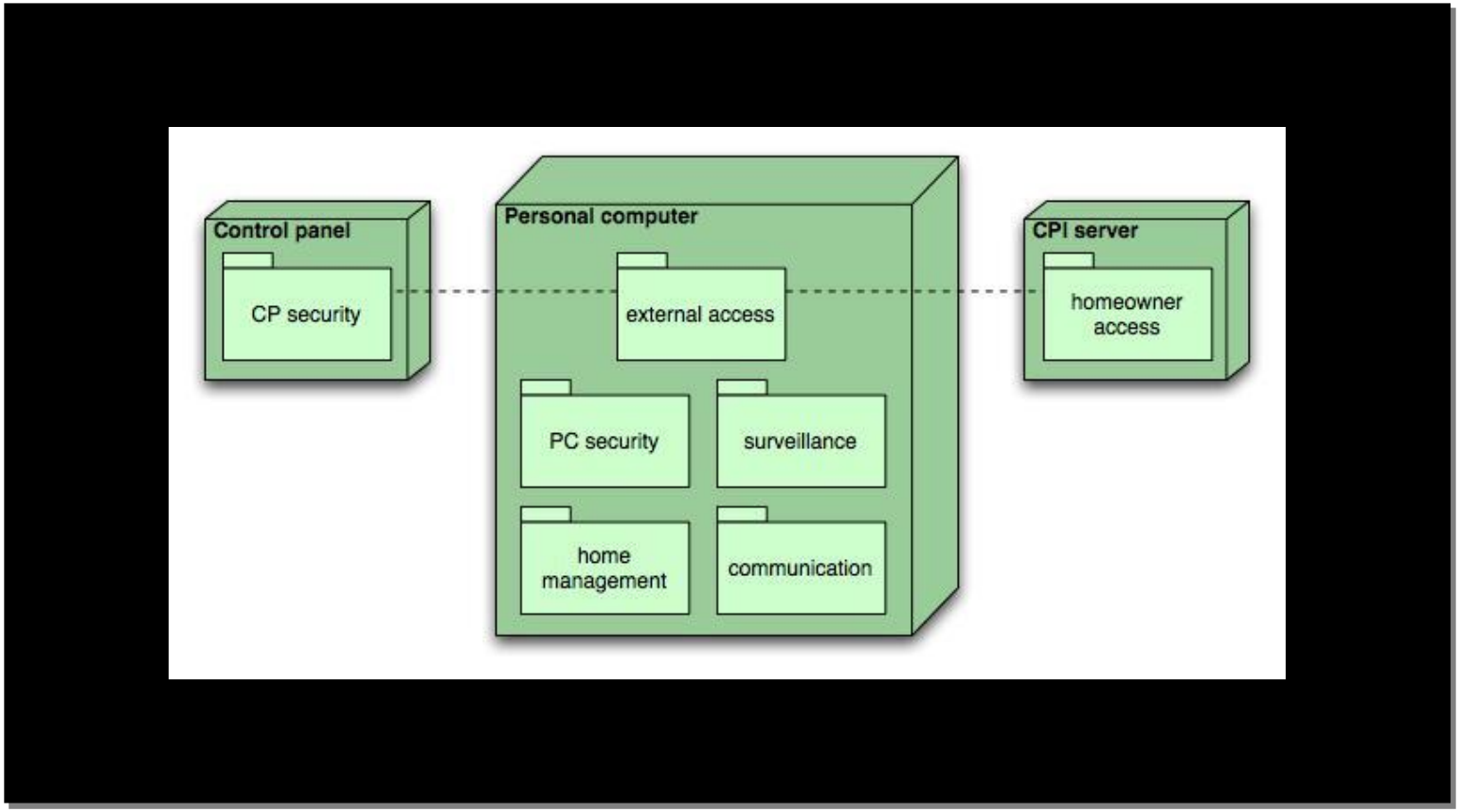


Figure 9.6 UML interface representation for **ControlPanel**

Component Elements



Deployment Diagram



Another Dimension of Design Challenge

According to Capers Jones (of Software Productivity Research), as of 2009, a large organization with over 250 applications would have used more than 50 different design languages and methodologies such as

- —Various flavors of UML
- —Flowcharts
- —Decision tables,
- —Data-flow diagrams
- —HIPO (hierarchical Input-Process-Output) diagrams
- —LePus3(Language for Pattern Uniform Specification)
- —Express (data modeling language)
- —Nassi-Schneiderman charts
- —Jackson design
- —Etc.

Due to incompatible design representations there is no easy way

- —to pick out common features or patterns
- —to identify library of reusable materials.

Design and Quality

McGlaughlin[91] suggests three characteristics that serve as a guide for evaluation of a good design:

- 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.

Guidelines to achieve quality design

- 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 *- Davis*

- 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” 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.

Summary

- A software design creates meaningful engineering representation (or model) of some software product that is to be built.
- Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model.
- A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, behavior) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system.
- Each design product is reviewed for quality
  - —identify and correct errors, inconsistencies, or omissions,
  - —whether better alternatives exist, and
  - —whether the design model can be implemented within the project constraints