

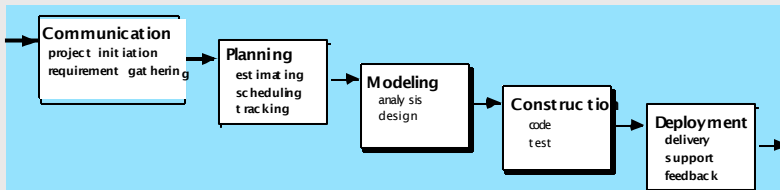
Design Concepts (Ch. 8)



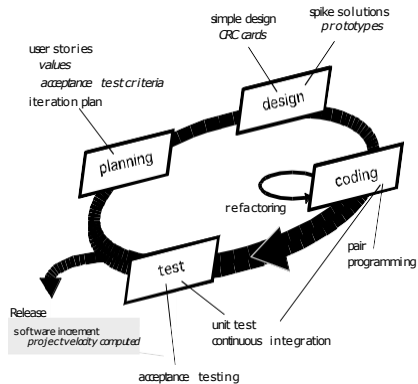
Copyright © 2002 United Feature Syndicate, Inc.

A good design should make the coding much easier!

The Waterfall Model



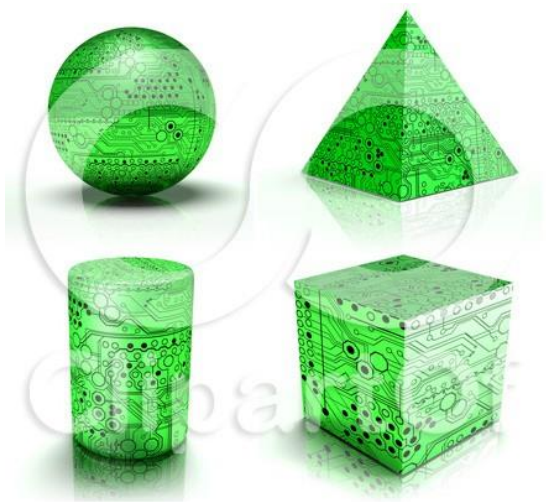
Extreme Programming (XP)



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7th* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Design

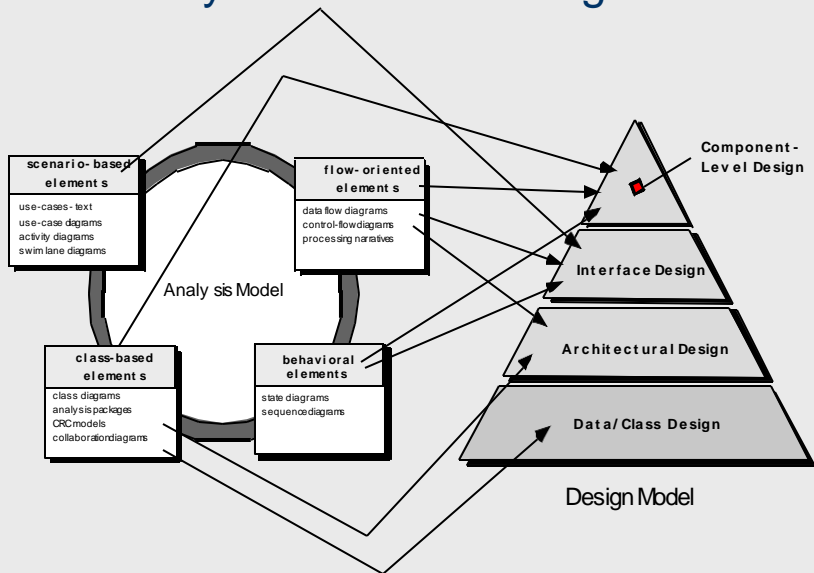
- Mitch Kapor, the creator of **Lotus 1-2-3**, presented a “software design manifesto” in *Dr. Dobbs Journal*. He said:
 - Good software design should exhibit:
 - *Firmness*: A program should not have any bugs that inhibit its function.
 - *Commodity*: A program should be suitable for the purposes for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



www.clipartof.com · 29258

(c) clipartof.com

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]



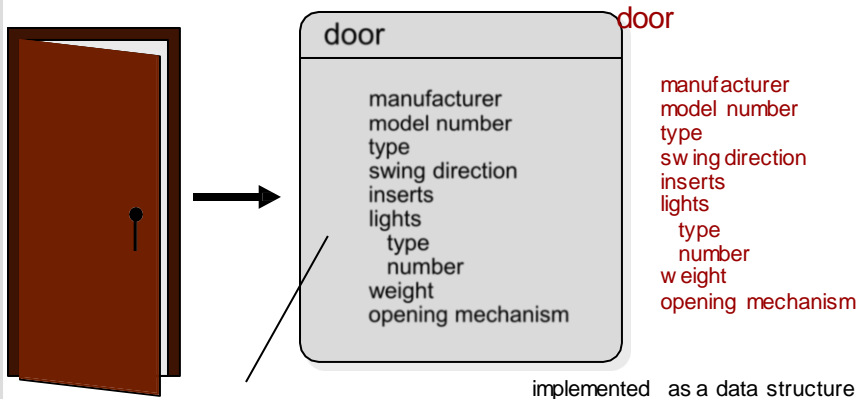
Fundamental Concepts

- **Abstraction**—data, procedure, control
- **Architecture**—the overall structure of the software
- **Patterns**—“conveys the essence” of a proven design solution
- **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

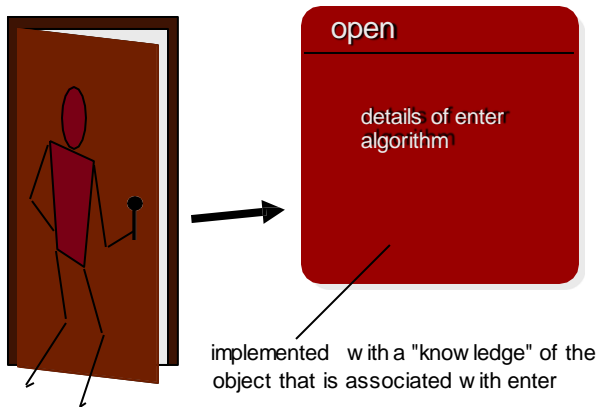


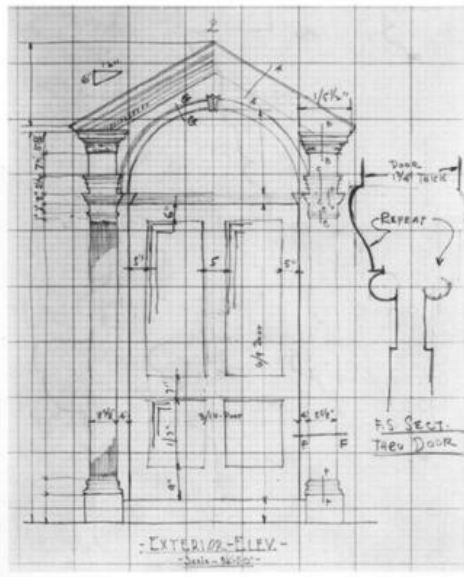
(c) dictionary.com

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

Simple Pattern Template

Pattern name

Intent

Also-known-as

Motivation

Applicability

Structure

Participants

Collaborations

Consequences

Related 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

These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 7/e

What is Stack Exchange?

Stack Exchange is a [growing network](#) of individual communities, each dedicated to serving experts in a specific field. We build libraries of high-quality questions and answers, focused on each community's area of expertise.

From programmers sharing answers on parsing HTML, to researchers seeking solutions to combinatorial problems, to photographers exposing lighting techniques, our communities are built by and for those best able to define them: the experts and enthusiasts.

Network Stats

107 Q&A sites

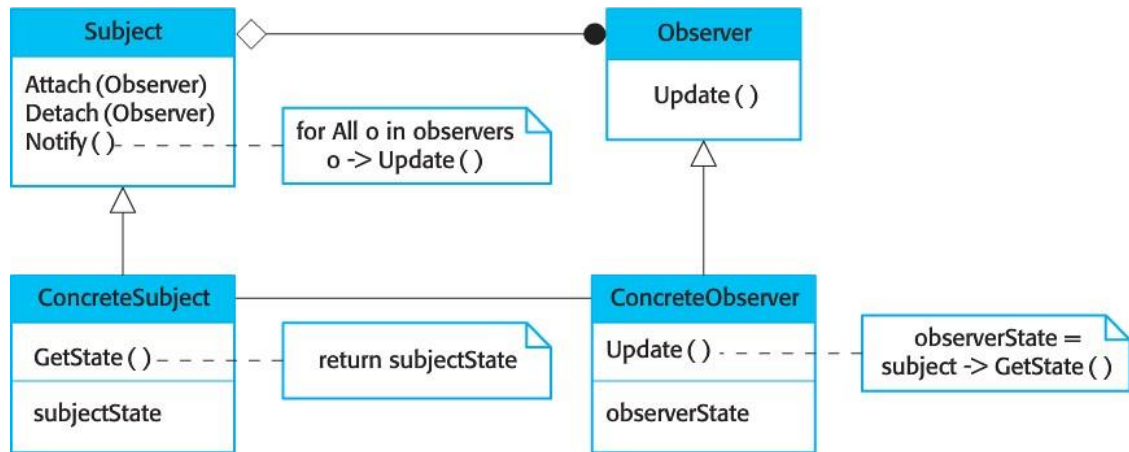
4,113,796 users

7.3 million questions

13.1 million answers

8.5 million visits per day

An Observer Programming Pattern





(c) bp.blogspot.com

Separation of Concerns

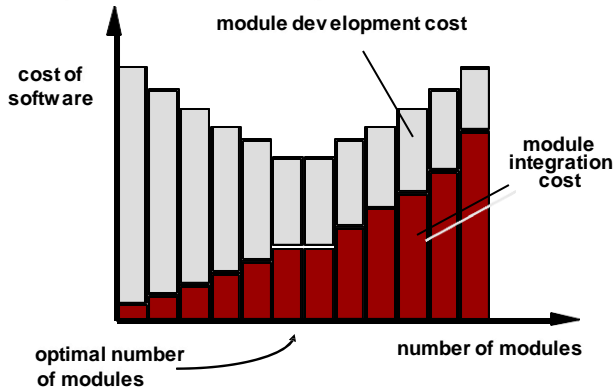
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more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

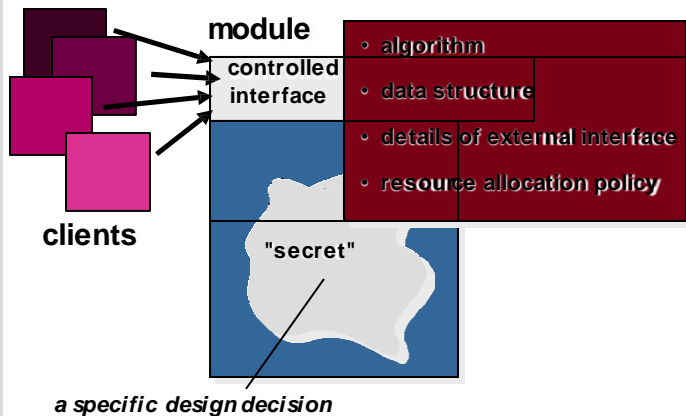


These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.



(c) bp.blogspot.com

Information Hiding



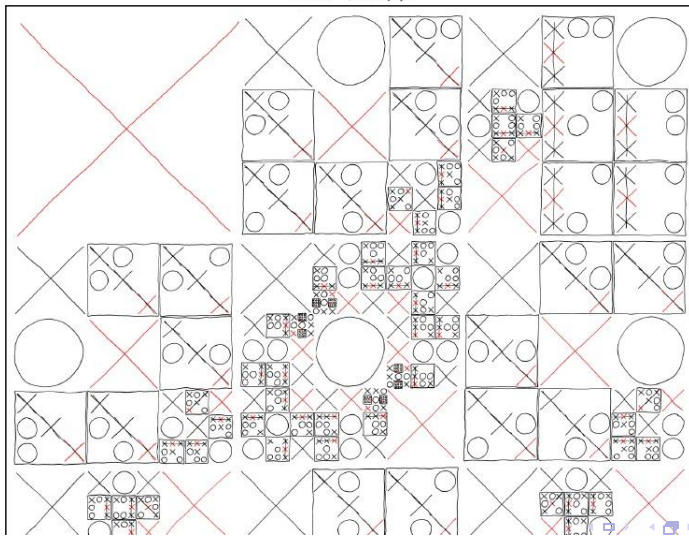
Why Information Hiding?

- 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

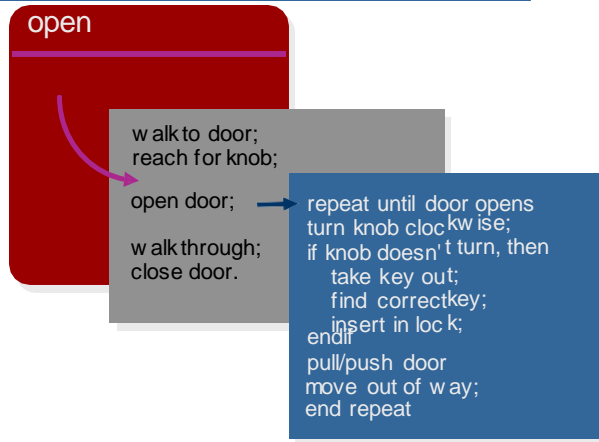
COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

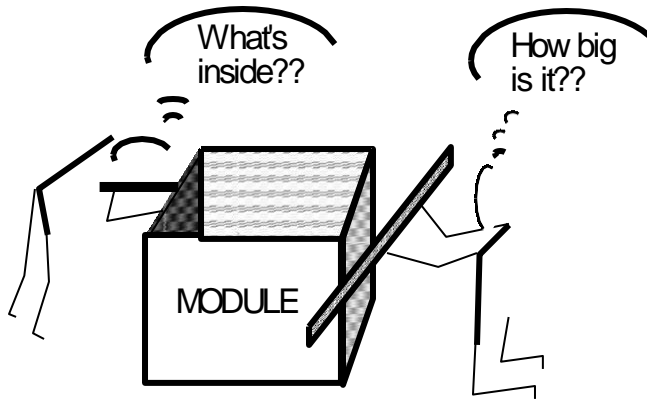
MAP FOR X:



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.

00!



(c) jacarandafm.com

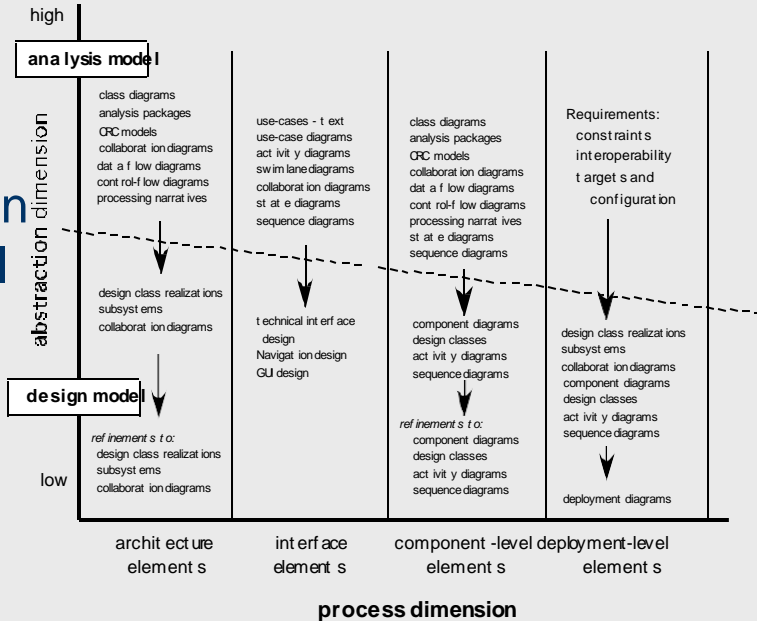
OO Design Concepts

- **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.

The Design Model



Design Model Elements

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

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 12) **and styles** (Chapter 9).

Interface Elements

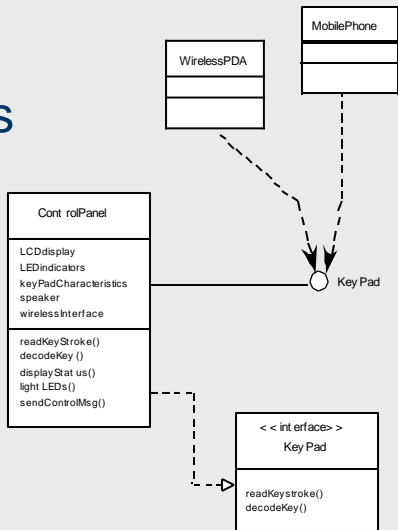
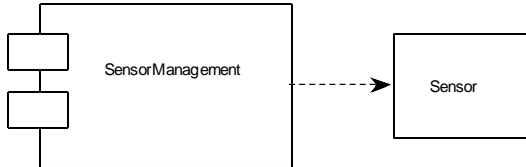


Figure 9.6 UML interface representation for **ControlPanel**

Component Elements





(c) military.com

Deployment Elements

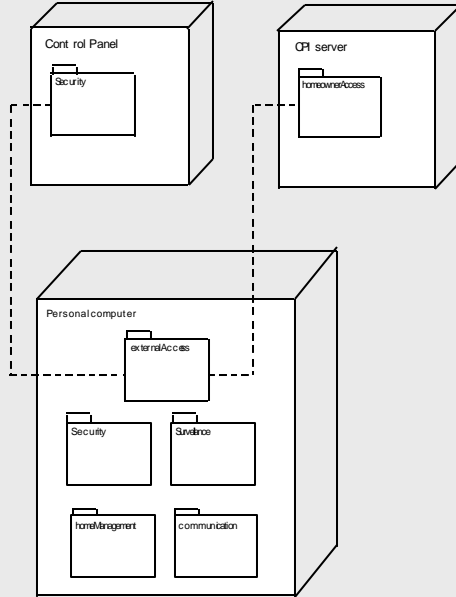


Figure 9.8 UML deployment diagram for SafeHome

Sustainability (Maintenance)

- Staffing changes over time
 - Original designer may leave
 - Original programmer may move on to other projects
 - Maintenance may go on for many years beyond first deployment
- Design, Architecture, Components
 - Often maintenance is not considered
 - Focus is “ideal” design or architecture
 - Consequences of complex implementation reverberate for many years
- Example: Voyages Project
 - Hibernate, AJAX
 - Simple changes require programmer intervention
 - Even simple map label modifications are a complex, time-consuming event

Fact/Fallacy Tidbit

- Fact 41
Maintenance typically consumes 40% to 80% of software costs
- Discussion
 - Original software build cost is about 20%-60% of total
 - Maintenance is dominant phase: never ends
 - “Old hardware becomes obsolete; old software goes into production every night”

From Robert Glass, “Facts & Fallacies of Software Engineering”