Ch.19 Quality Concepts

April 26, 2015

19.1 What Is Quality

- The ***transcendental (抽象的) view*** argues that quality is something that you immediately recognize, but cannot explicitly define.
- The ***user view*** sees quality in terms of an end-user's specific goals. If a product ***meets those goals***, it exhibits quality.
- The ***manufacturer's view*** defines quality in terms of the original specification of the product. If the product ***conforms to the spec***, it exhibits quality.
- The ***product view*** suggests that quality can be tied to ***inherent*** characteristics (e.g., ***functions and features***) of a product.
- Finally, the ***value-based view*** measures quality based on how much a customer is ***willing to pay*** for a product. In reality, quality encompasses all of these views and more.

19.2 Software Quality

– "Bad software plagues(使痛苦) nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction" --- ***Computer World* [Hil05]**

– "The sorry state of software quality" reporting that the quality problem had not gotten any better--- ***InfoWorld [Fos06]***

– Today, software quality remains an issue, but who is to blame?

---Customers blame developers, arguing that **sloppy** （草率的）practices lead to low-quality software.

---Developers blame customers (and other stakeholders), arguing that **irrational** delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.

19.2 Software Quality

Quality

- The *American Heritage Dictionary* defines *quality* as
  - "a characteristic or attribute of something."
- For software, two kinds of quality may be encountered:
  - ***Quality of design*** encompasses requirements, specifications, and the design of the system.
  - ***Quality of conformance*** is an issue focused primarily on implementation.

  - ***User satisfaction = compliant product + good quality + delivery within budget and schedule***

19.2 Software Quality

- Software quality can be defined as:
  - *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*
- This definition has been adapted from [Bes04] and replaces a more manufacturing-oriented view presented in earlier editions of this book.

19.2 Software Quality

Effective Software Process

- An ***effective software process*** establishes the infrastructure that supports any effort at building a high quality software product.

- The ***management aspects of process*** create the checks and balances that help avoid project chaos—a key contributor to poor quality.

- ***Software engineering practices*** allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.

-

- Finally, umbrella ***activities*** such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

19.2 Software Quality

Useful Product

- A *useful product* delivers the content, functions, and features that the end-user desires
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Adding Value

- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The *software organization* gains added value because high quality software requires *less maintenance effort*, *fewer bug fixes*, and *reduced customer support*.
- The *user community* gains added value because the application provides a *useful capability* in a way that *expedites some business process（例：余额宝）*.
- The end result is:
  - (1) greater software product revenue（收益）,
  - (2) better profitability when an application supports a business process, and/or
  - (3) improved availability of information that is crucial for the business.

19.2 Software Quality

Quality Dimensions

- David Garvin [Gar87]:
  - **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
  - **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
  - **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
  - **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto（实际的） design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

19.2 Software Quality

Quality Dimensions

- **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent（不经意的） generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics.** Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious “presence” that are hard to quantify but evident nonetheless.
- **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality.

19.2 Software Quality

Measuring Quality

- General quality dimensions and factors are not adequate for assessing the quality of an application in concrete terms
- Project teams need to develop a set of targeted questions to assess the degree to which each application quality factor has been satisfied
- Subjective measures of software quality may be viewed as little more than personal opinion
- Software metrics represent indirect measures of some manifestation （表示） of quality and attempt to quantify the assessment of software quality

19.3 The Software Quality Dilemma（困境）

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete. **[Ven03]**

19.3 The Software Quality Dilemma

"Good Enough" Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure(某种模糊) or specialized functions and features that contain known bugs.

- Arguments *against* "good enough."
  - It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in. (Ex. MS Windows's Releases)
  - If you work for a small company be wary of this philosophy. If you deliver a "good enough" (buggy) product, you risk permanent damage to your company's reputation.
  - You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet(垂直落下) and your company to fold.
  - If you work in certain application domains (e.g., real time embedded software, application software that is integrated with hardware can be negligent and open your company to expensive litigation(打官司).

19.3 The Software Quality Dilemma

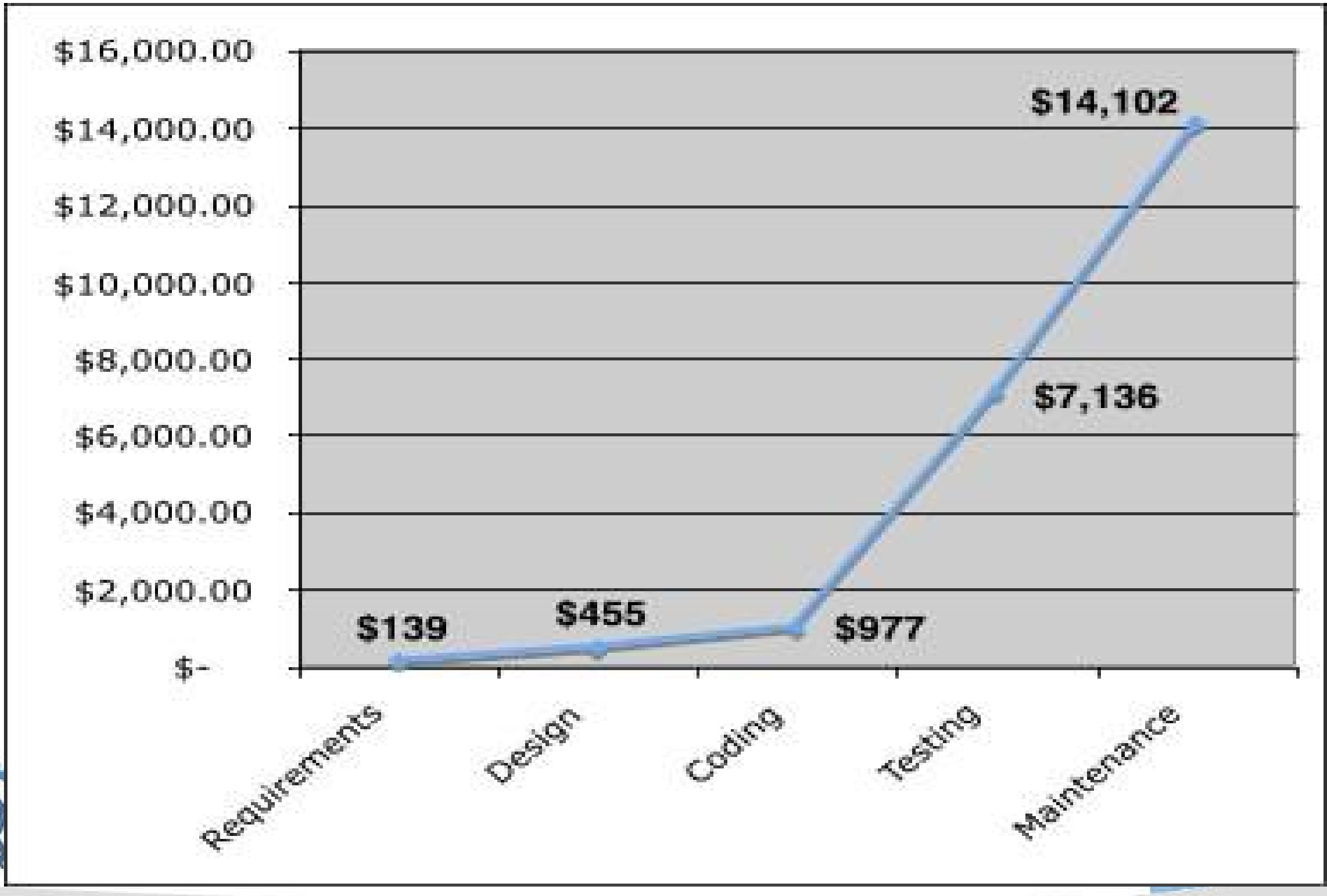
Cost of Quality

- *Prevention costs* include
  - quality planning
  - formal technical reviews
  - test equipment
  - Training
- *Internal failure costs* include
  - rework
  - repair
  - failure mode analysis
- *External failure costs* are
  - complaint resolution
  - product return and replacement
  - help line support
  - warranty work

19.3 The Software Quality Dilemma

Cost

- The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.



19.3 The Software Quality Dilemma

Quality and Risk

- *"People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."* ---by **SEPA**

- **Example**:
  - *Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.*

19.3 The Software Quality Dilemma

Negligence and Liability(责任)

- The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity.
  – The system might support a major corporate function (e.g., pension(退休金) management) or some governmental function (e.g., healthcare administration or homeland security).

- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.

- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.

- Litigation ensues.

19.3 The Software Quality Dilemma

Low Quality Software

- Low quality software increases risks for both developers and end-users
- When systems are delivered late, fail to deliver functionality, and does not meet customer expectations litigation ensues
- Low quality software is easier to hack and can increase the security risks for the application once deployed
- A secure system cannot be built without focusing on quality (security, reliability, dependability) during the design phase
- Low quality software is liable(有责任的) to contain architectural flaws as well as implementation problems (bugs)

19.3 The Software Quality Dilemma

Impact of Management Decisions

- Estimation decisions – irrational delivery date estimates cause teams to take short-cuts that can lead to reduced product quality
- Scheduling decisions – failing to pay attention to task dependencies when creating the project schedule
- Risk-oriented decisions – reacting to each crisis as it arises rather than building in mechanisms to monitor risks may result in products having reduced quality

19.4 Achieving Software Quality

- Software quality is the **result** of **good project management** and **solid engineering practice.**
- To build high quality software you must **understand the problem** to be solved and be capable of **creating a quality design** the conforms to the problem requirements.
- **Eliminating architectural flaws** during design can improve quality.
- **Project management** – project plan includes explicit techniques for quality and change management.
- **Quality control** - series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications.
- **Quality assurance** - consists of the auditing and reporting procedures used to provide management with data needed to make proactive decisions.

Ch.12 Design Concepts

- **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.

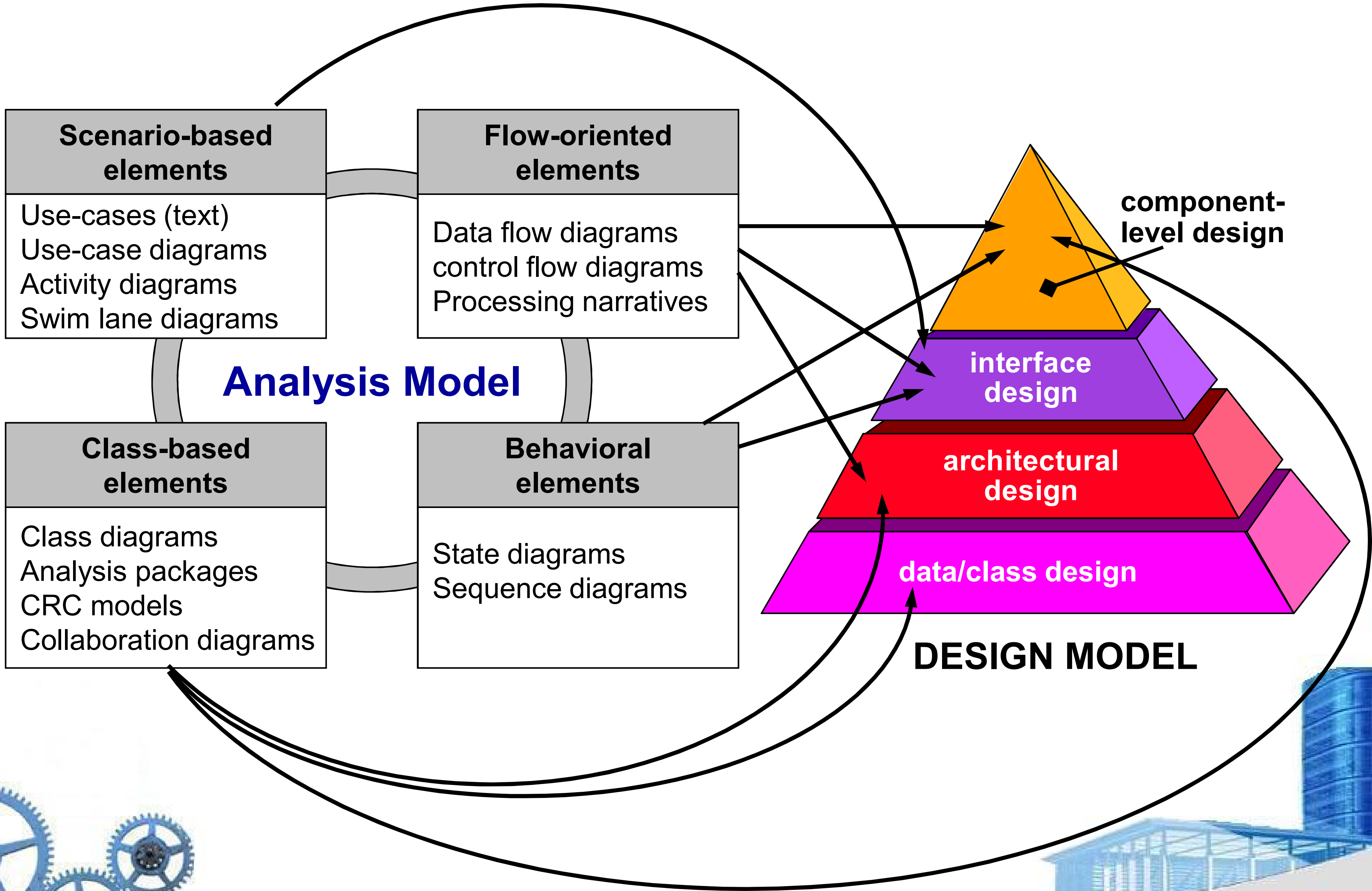
- **Software Design**

  - 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(覆盖) 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(重蹈覆辙) 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---**Ex.**双**11**节
  - 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
- *O-O design concepts*—Appendix II
- *Design Classes*—provide design detail that will enable analysis classes to be implemented

- **Data Abstraction**

door

manufacturer
model number
type
swing direction
inserts
lights
  type
  number
weight
opening mechanism

implemented as a data structure

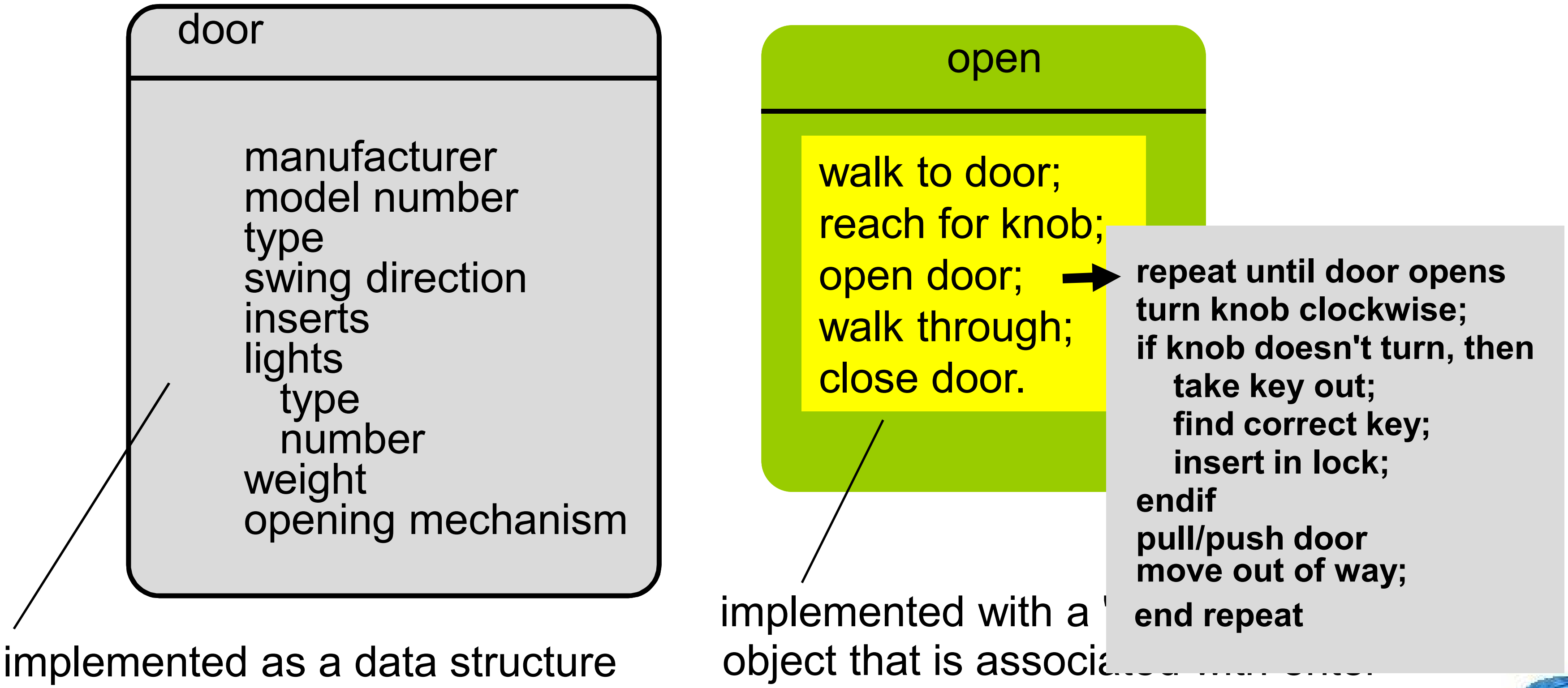
open

walk to door;
reach for knob;
open door;
walk through;
close door.

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

- **Refinement — top-down design strategy**

- # Data Abstraction



- **Refinement — top-down design strategy**

- **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.
- *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.

Design Concepts

Design forces **describe non-functional requirements (e.g., ease of maintainability, portability)**

- **Patterns**

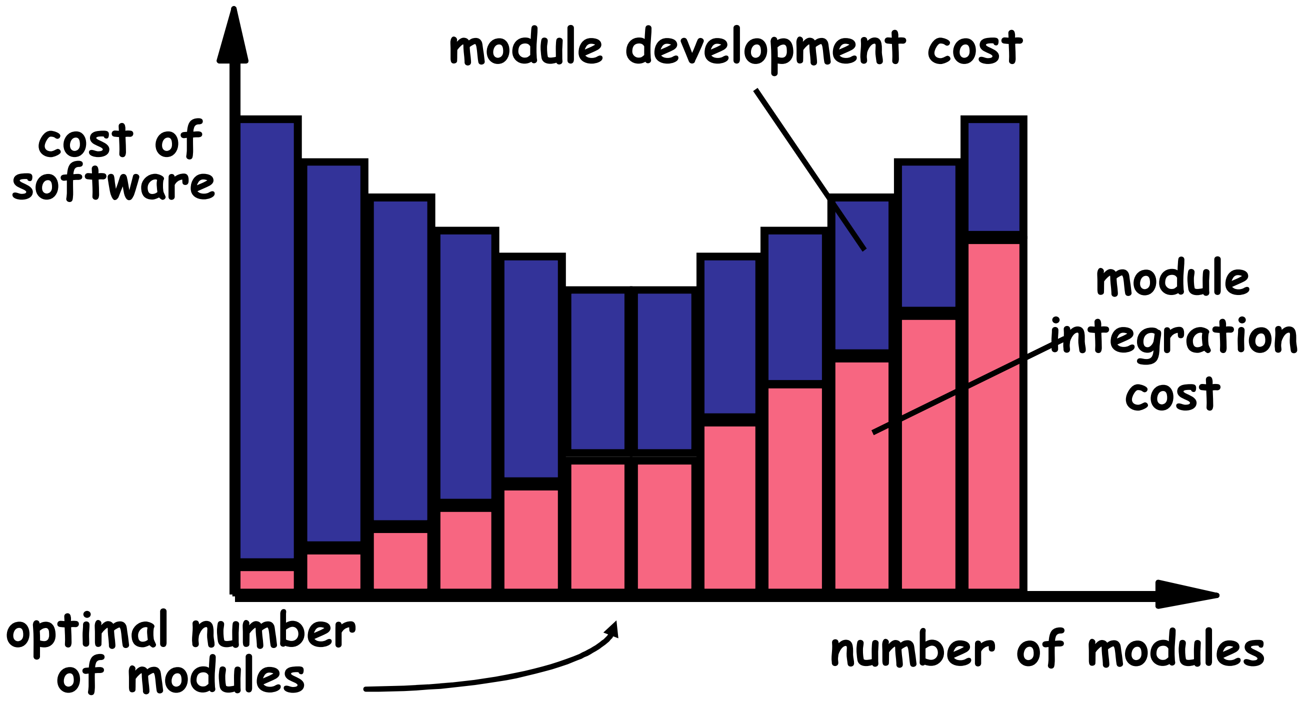
Design Pattern Template

- **Pattern name** describes the essence of the pattern
- **Intent** describes the pattern and what it does
- **Also-known-as** synonyms（同义词） for the pattern
- **Motivation** provides an example of the problem
- **Applicability** specific design situations in which the pattern is applicable
- **Structure** classes that are required to implement the pattern
- **Participants** responsibilities of the classes
- **Collaborations** how the participants collaborate to carry out their responsibilities
- **Consequences** describes the "design forces" that affect the pattern and the potential trade-offs
- **Related patterns** cross-references related design patterns

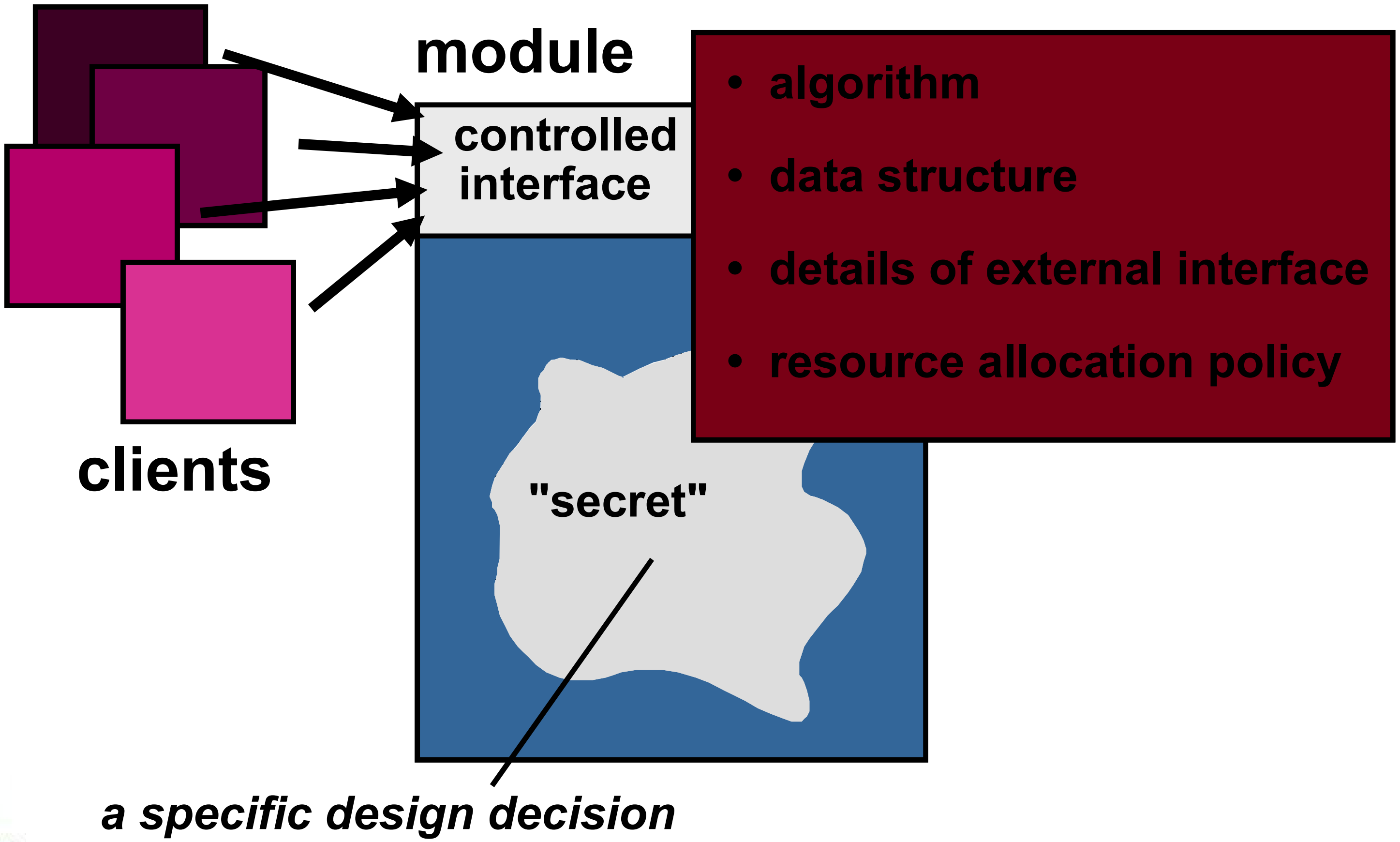
- **Separation of Concerns**
  - Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
  - A *concern* is a feature or behavior that is specified as part of the requirements model for the software
  - By separating concerns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

Design Concepts

- **Modularity---Trade-offs**

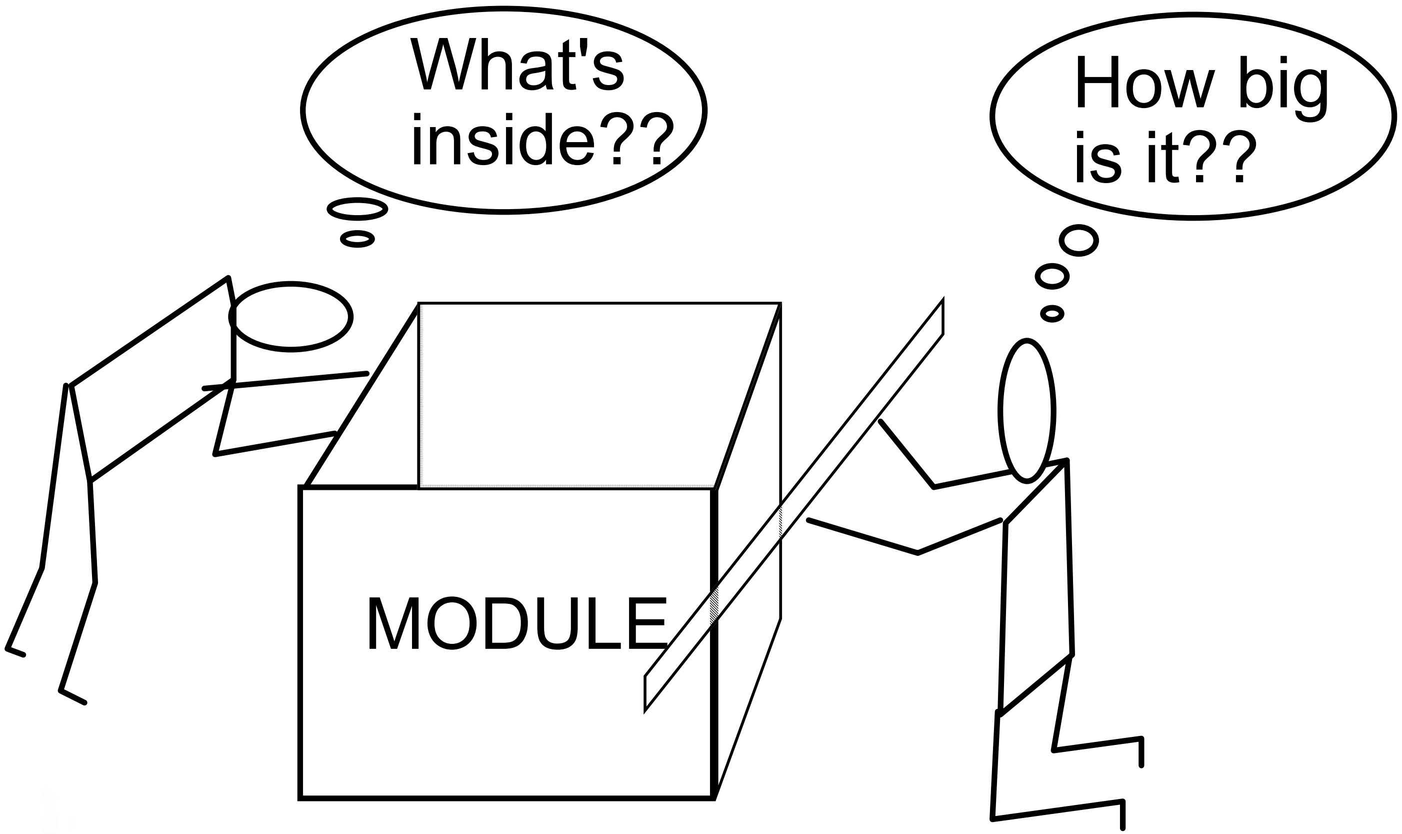


- **Information Hiding**



- **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

- **Sizing Modules: Two Views**



- **Functional Independence**
  - Functional independence is achieved by developing modules with "single-minded" function and an "aversion(厌恶)" 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. [Ex. Safehome]
  - A: access video; B: generic security
  - A* and B* are the design representations of A and B

  →B*crosscuts (横切) A*

  - *Aspect* is a representation of a cross-cutting concern.

- **Refactoring**

---change 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
- ✓ any other design failure that can be corrected to yield a better design.

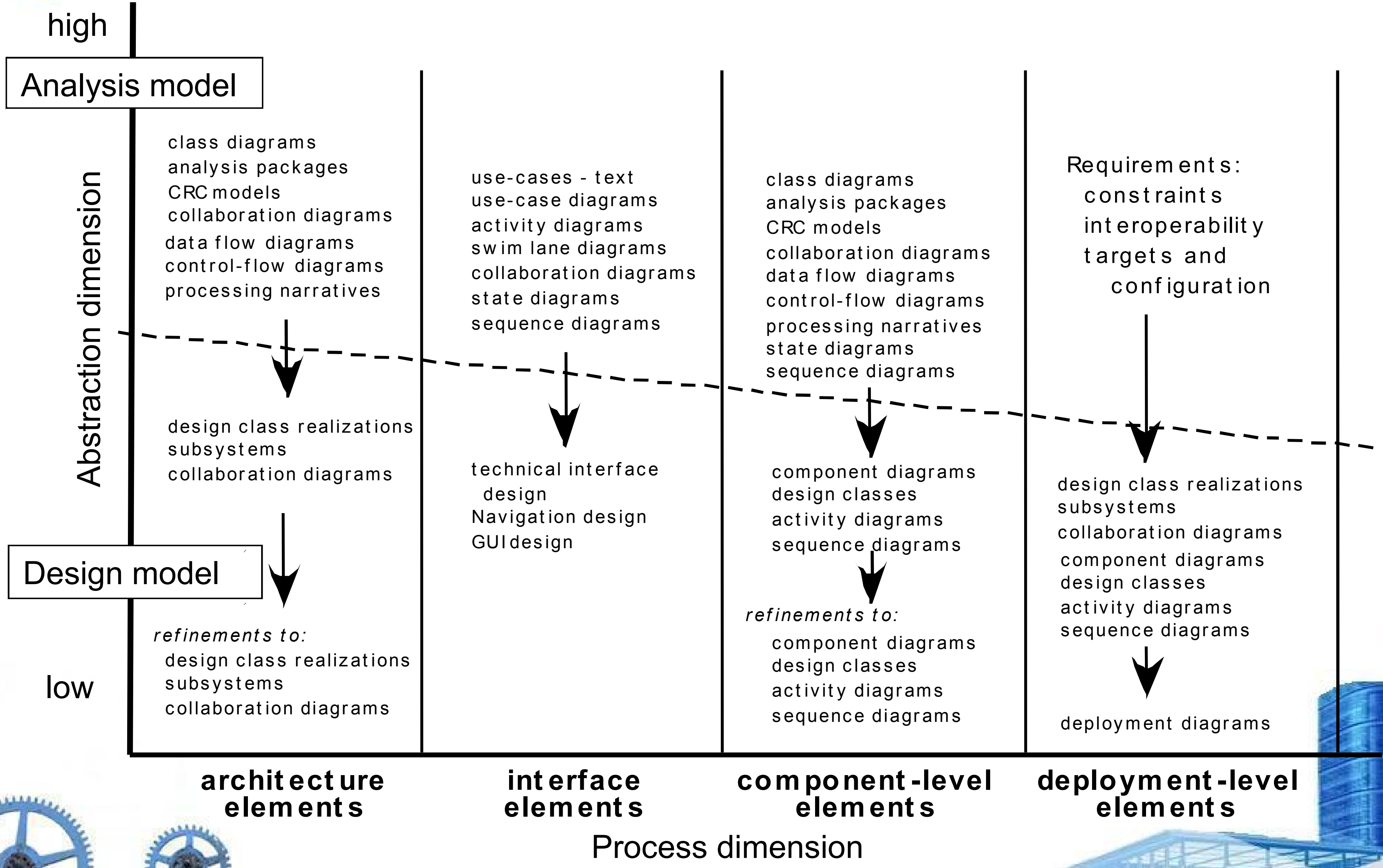
- **O-O Design Concepts**
  - *Design classes*
    - Entity classes
    - Boundary classes
    - Controller classes
  - *Inheritance*—all responsibilities of a superclass is immediately inherited by all subclasses
  - *Messages*—stimulate some behavior to occur in the receiving object
  - *Polymorphism*—a characteristic that greatly reduces the effort required to extend the design

---EX: Class: Animal, Subclass: Dog and Chicken

- **Design Class Characteristics**

  - *Complete* - includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
  - *Primitiveness(原始性)* – 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*

- **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** (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**

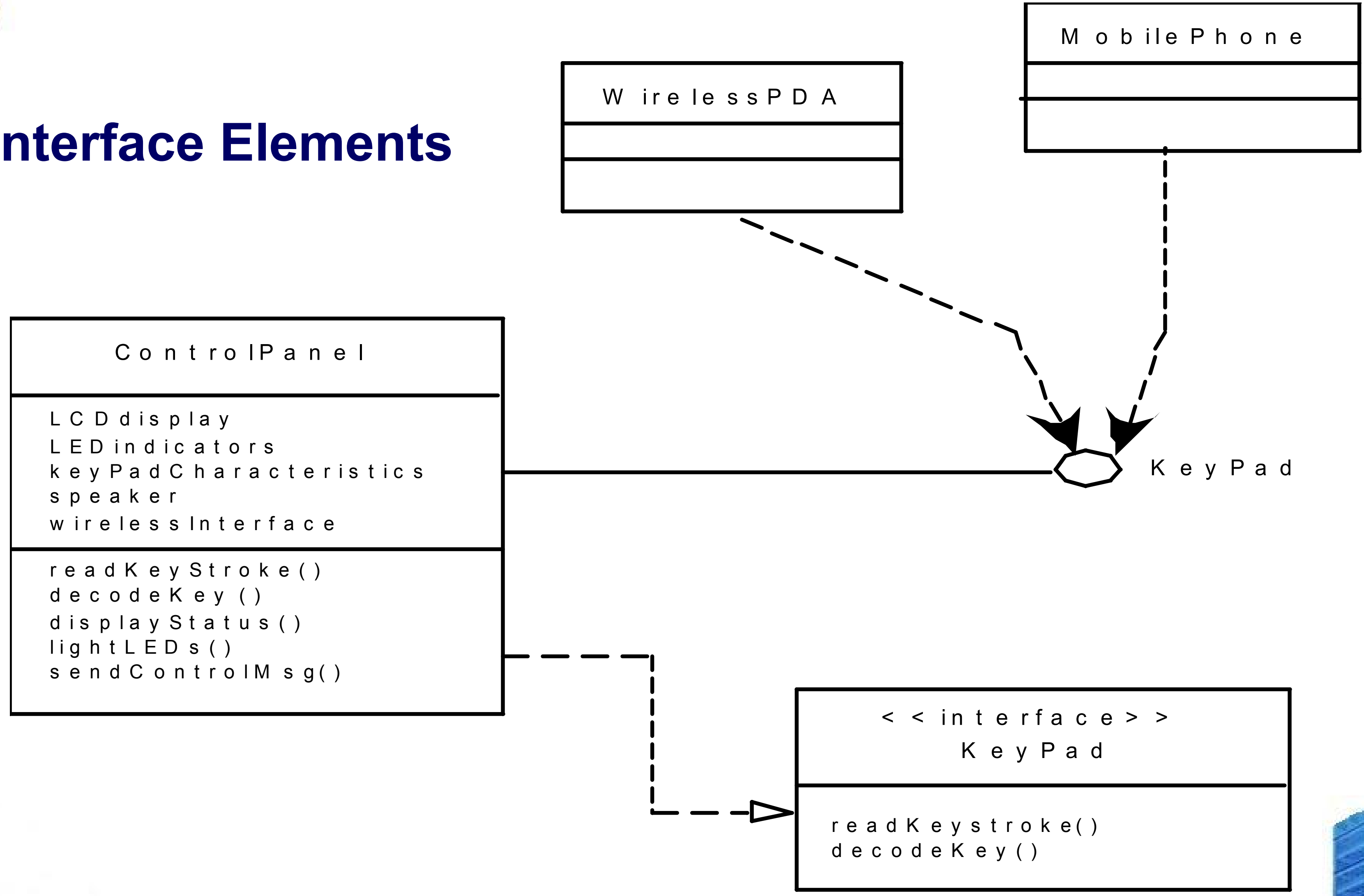
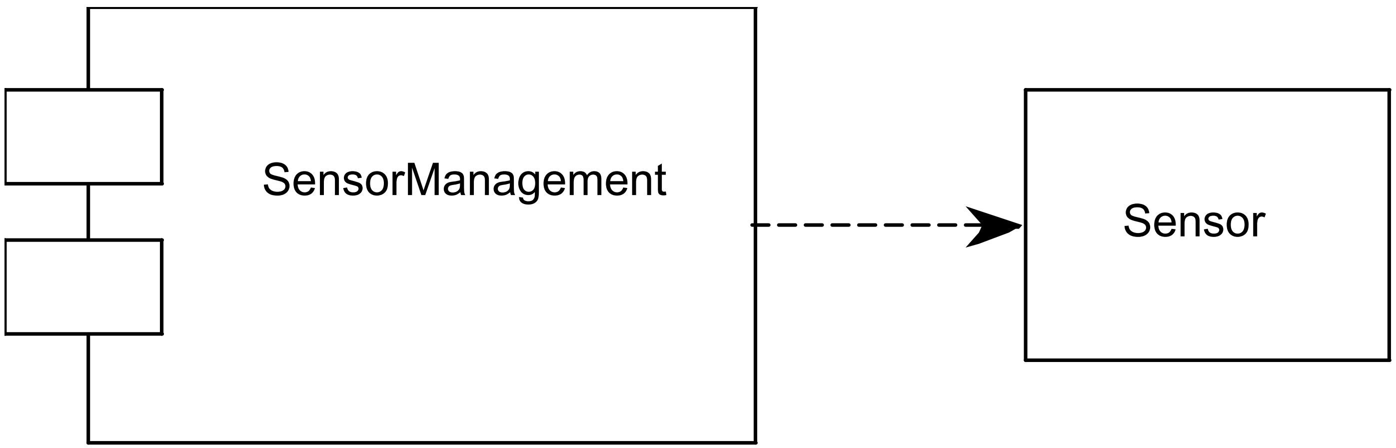


Figure 9.6 UML interface representation for **ControlPanel**

- **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

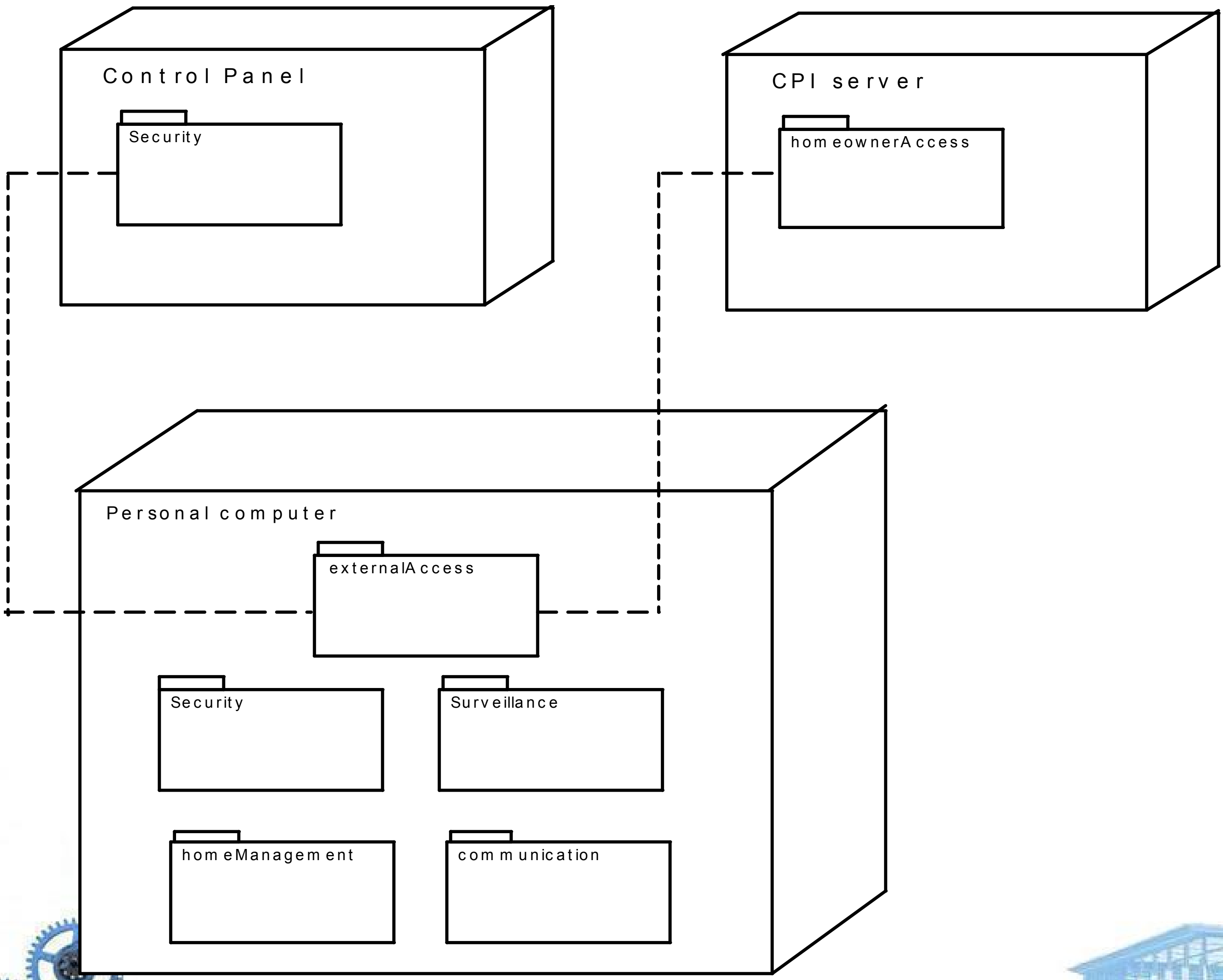


Figure 9.8 UML deployment diagram for *SafeHome*

Tasks

- **Review** Ch.12，19
- **Finish** “Problems and points to ponder” in Ch. 12，19
- **Preview** Ch.13, 17
- **Submit Requirement Gathering Report due April 27 !**