Chapter 9
Design Engineering

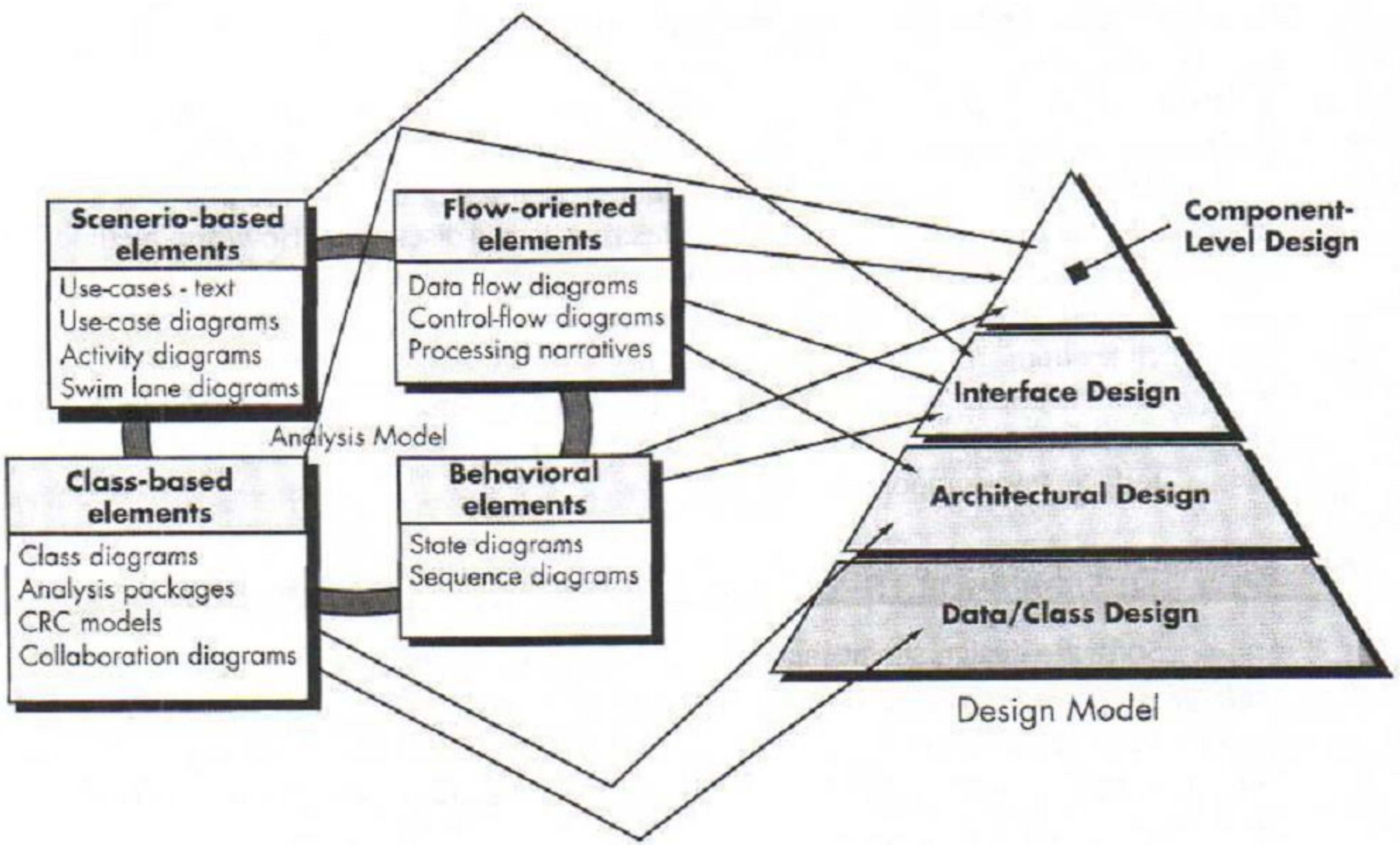
Software Engineering: A Practitioner's Approach, 6th edition
by Roger S. Pressman

Introduction

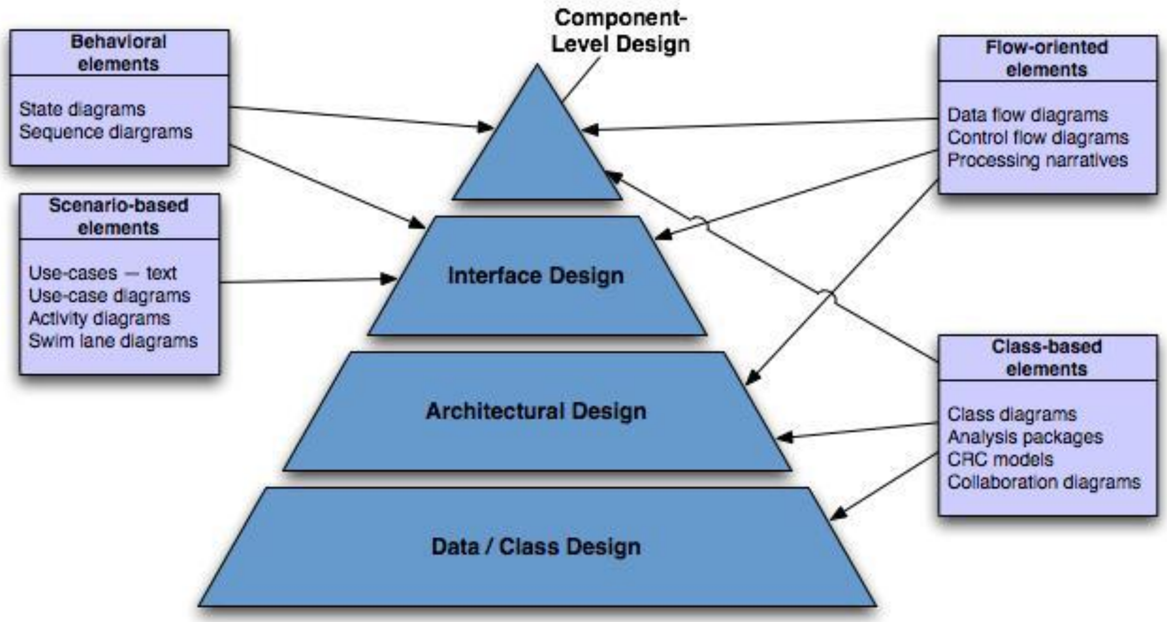
- Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high—quality system or product.
- Design is a core engineering activity.
- A good software
  - **Firmness:** A program should not have any bugs that inhibit its function.
  - **Commodity:** A program should be suitable for the purposes for which it was intended.
  - **Delight:** The experience of using the program should be a pleasurable one.

Design with the Context of Software Engineering

- The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight.
- Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve.
- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).


Scenerio-based elements
Use-cases - text
Use-case diagrams
Activity diagrams
Swim lane diagrams
Flow-oriented elements
Data flow diagrams
Control-flow diagrams
Processing narratives
Class-based elements
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Behavioral elements
State diagrams
Sequence diagrams
Analysis Model
Component-Level Design
Interface Design
Architectural Design
Data/Class Design
Design Model

Analysis → Design



Design Process and Design Quality

- Three characteristics that serve as a guide for the 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.

Quality guidelines for Good Design

1. A design should exhibit an architecture that
   - (a) has been created using recognizable architectural styles or patterns,
   - (b) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and
   - (c) can be implemented in an evolutionary fashion} thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

Quality guidelines for Good Design

5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality attributes (FURPS)

- Functionality is assessed by evaluating the **feature set** and **capabilities** of the program, the generality of the functions that are delivered, and the security of the overall system.
- Usability: is assessed by considering **human factors** , overall aesthetics, consistency, and documentation.
- Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to—failure (**MTTF**), the ability to **recover** from failure, and the **predictability** of the program.
- Performance is measured by processing speed, **response time**, resource consumption, throughput, and efficiency.
- Supportability combines the ability to extend the program (**extensibility**), adaptability, serviceability

Design Concepts

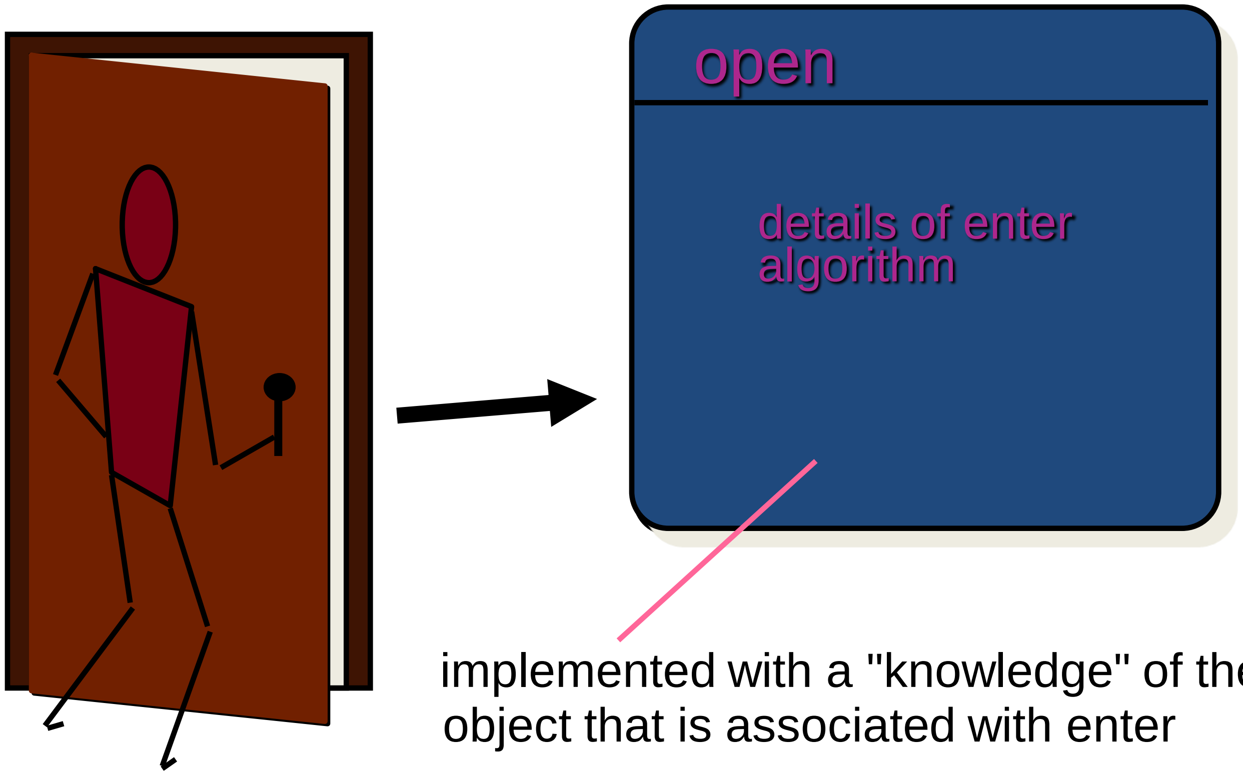
Getting A Program Work or Getting a Program Right

- 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
- information hiding — controlled interfaces
- functional independence — high cohesion and low coupling
- refinement — elaboration of detail for all abstractions
- refactoring — improve design without effecting behavior

Abstraction et al.

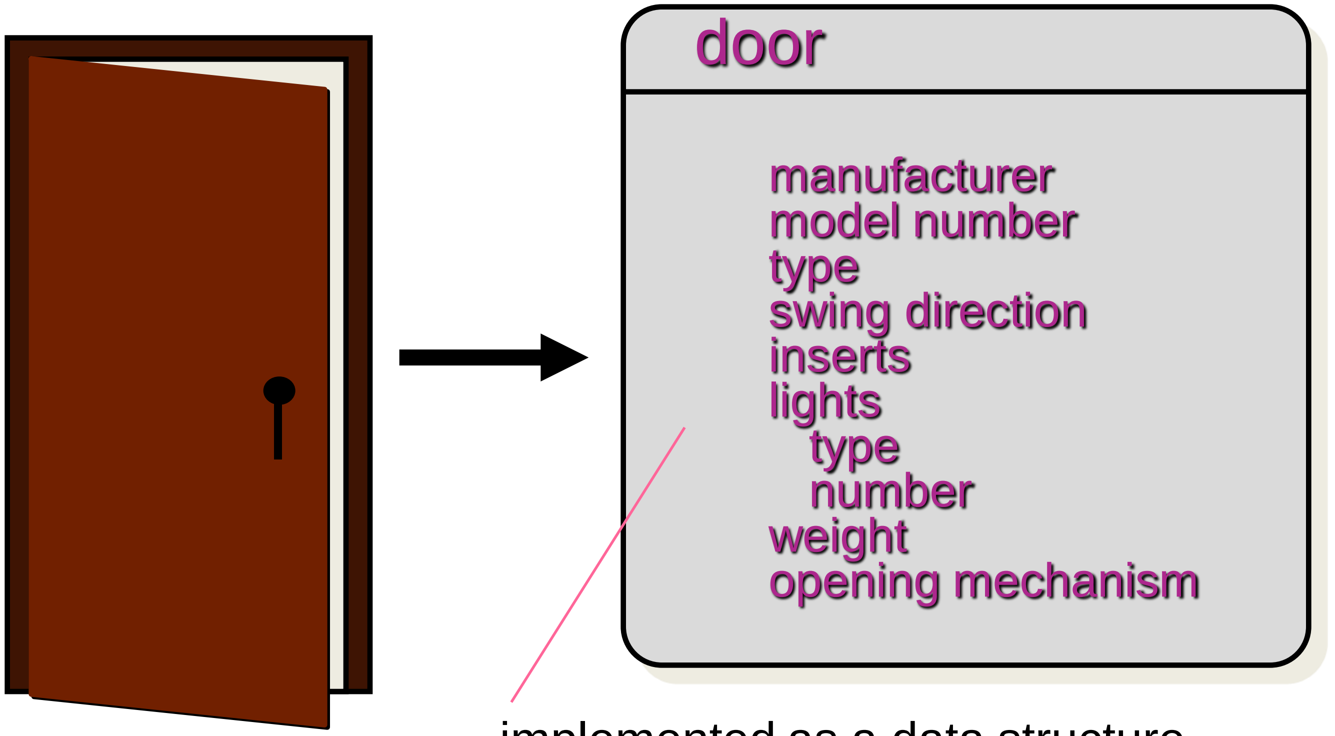
- Abstraction
  - process – extracting essential details
  - entity – a model or focused representation
- Information hiding
  - the suppression of inessential information
- Encapsulation
  - process – enclosing items in a container
  - entity – enclosure that holds the items

Procedural Abstraction



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

Data Abstraction



implemented as a data structure

Architecture

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

Architectural Models

Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of the business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

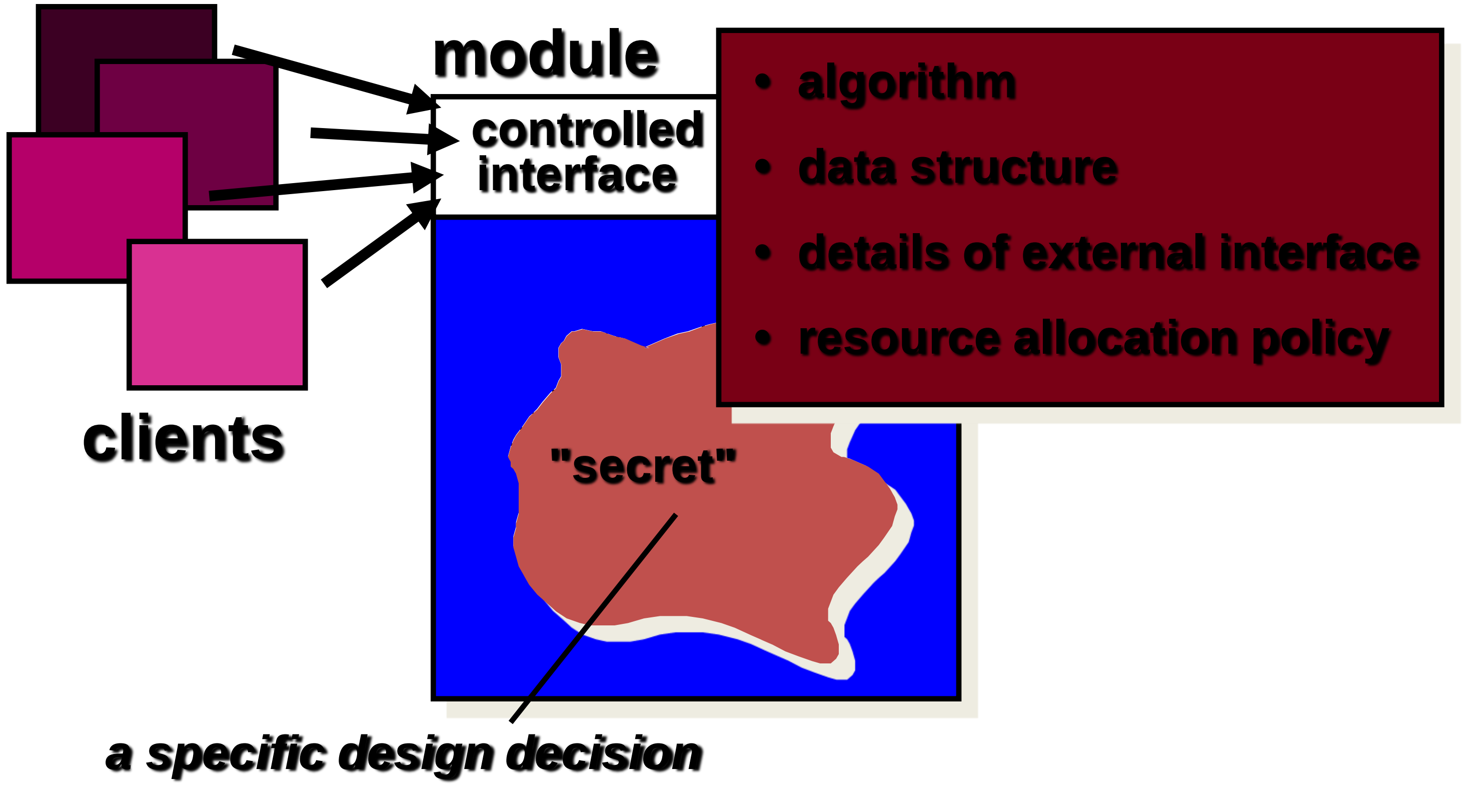
Modular Design

- Software is divided into separately named and addressable components. sometimes called modules, that are integrated to satisfy problem requirements
- Modularity is the single attribute of software that allows a program to be intellectually manageable

Information Hiding

- How do we decompose a software solution to obtain the best set of modules?
- Information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

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

Functional Independence

- Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

Functional Independence

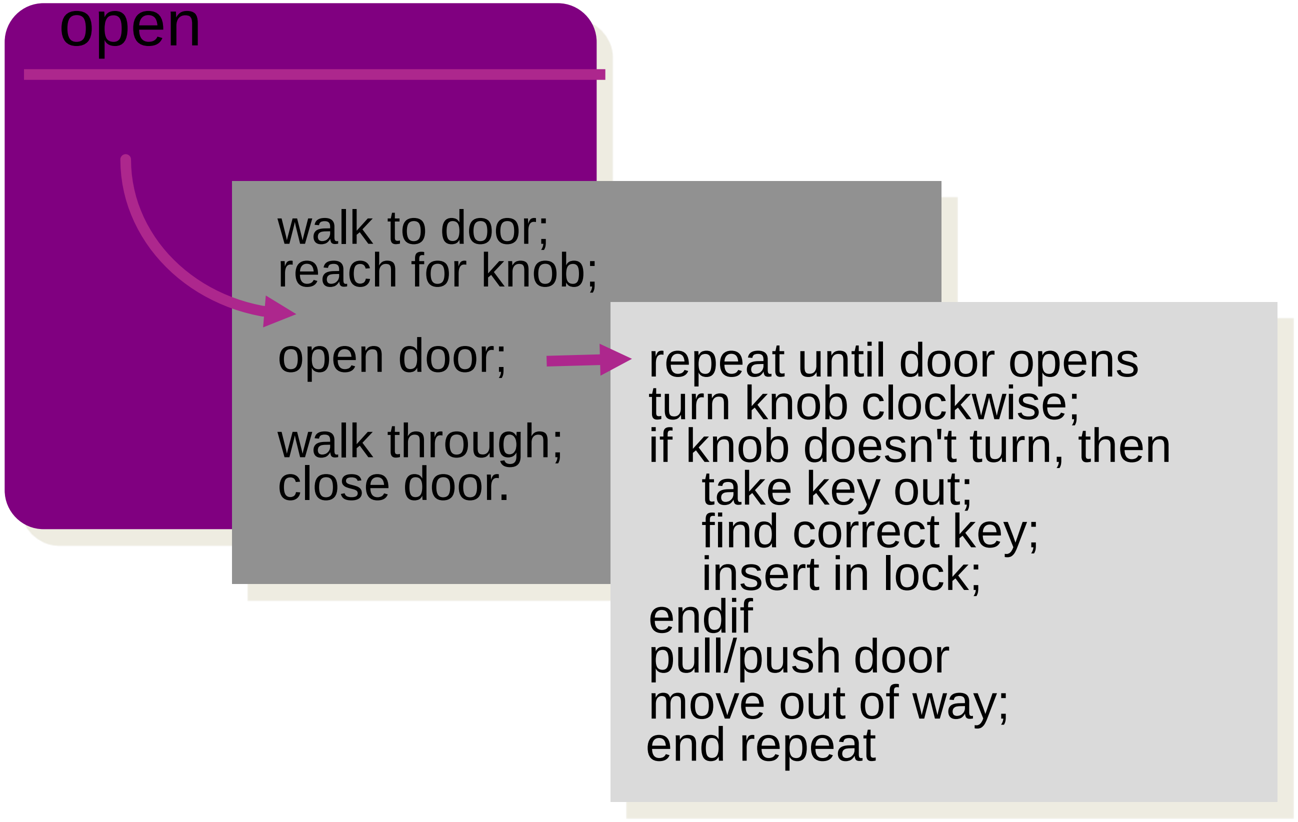
COHESION - the degree to which a module performs one and only one function.

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

Stepwise Refinement

- Stepwise refinement is a top—down design strategy
- A program is developed by successively refining levels of procedural detail.
- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually a process of elaboration

Stepwise Refinement



```
open
```

```
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
```

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.

example

```java
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            return "Don";
        }
        if (people[i].equals ("John")){
            return "John";
        }
        if (people[i].equals ("Kent")){
            eturn "Kent";
        }
    }
    return "";
}
```

```java
String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});
    for (int i=0; i<people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return "";
}
```

Design Classes

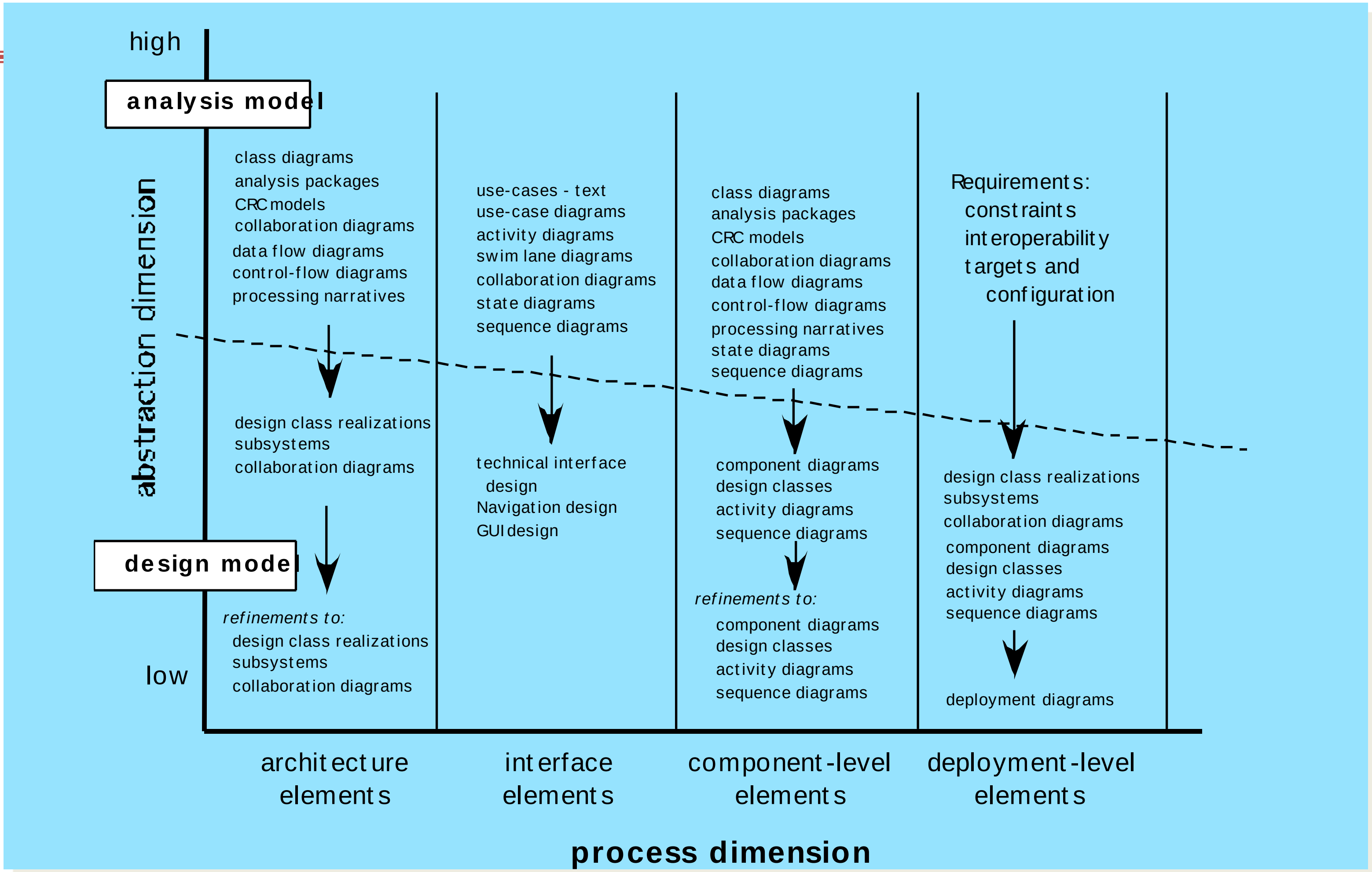
Types of Design Classes

- User interface classes
- 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.
- High cohesion – class should be focused on one kind of thing.
- Low coupling – collaboration should be kept to an acceptable minimum.

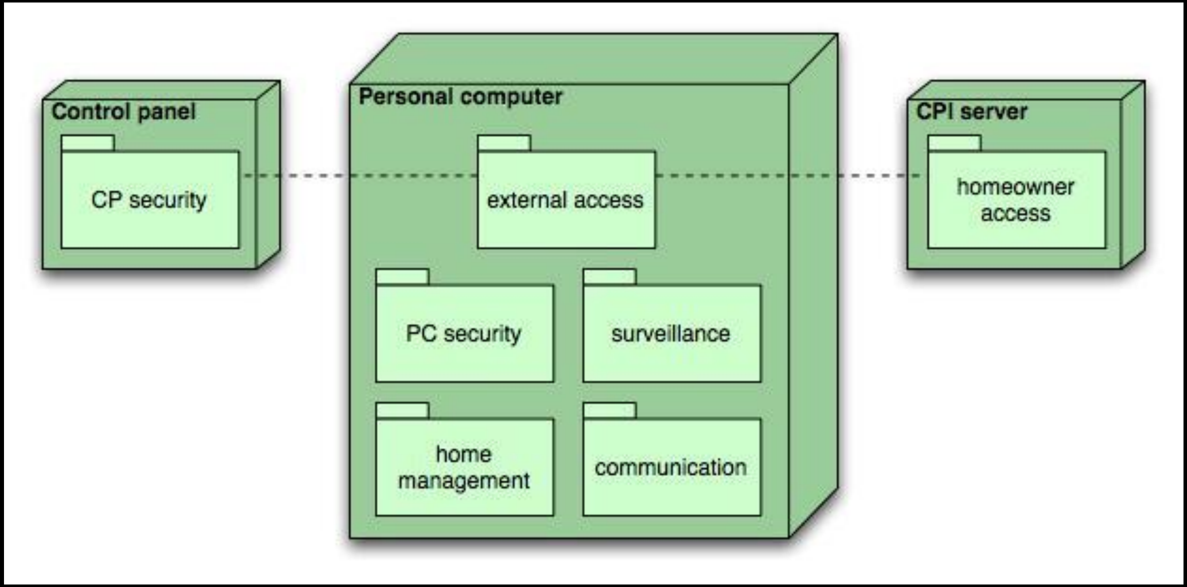
The Design Model



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

Deployment Diagram


Control panel
CP security
Personal computer
external access
PC security
surveillance
home management
communication
CPI server
homeowner access