Software Design

Software Design

- Deriving a solution which satisfies software requirements

Overview of Design

- What is it? A meaningful engineering representation of something that is to be built.
- Who does it? Software engineers with a variety of skills, ranging from human ergonomics to computer architecture
- Why is it important? A house would never be built without a blueprint. Why should software? Without design the system may fail with small changes, is difficult to test and cannot be assessed for quality
- What is the work product? A design specification

Programmer's Approach to Software Engineering

Skip requirements engineering and design phases;
start writing code

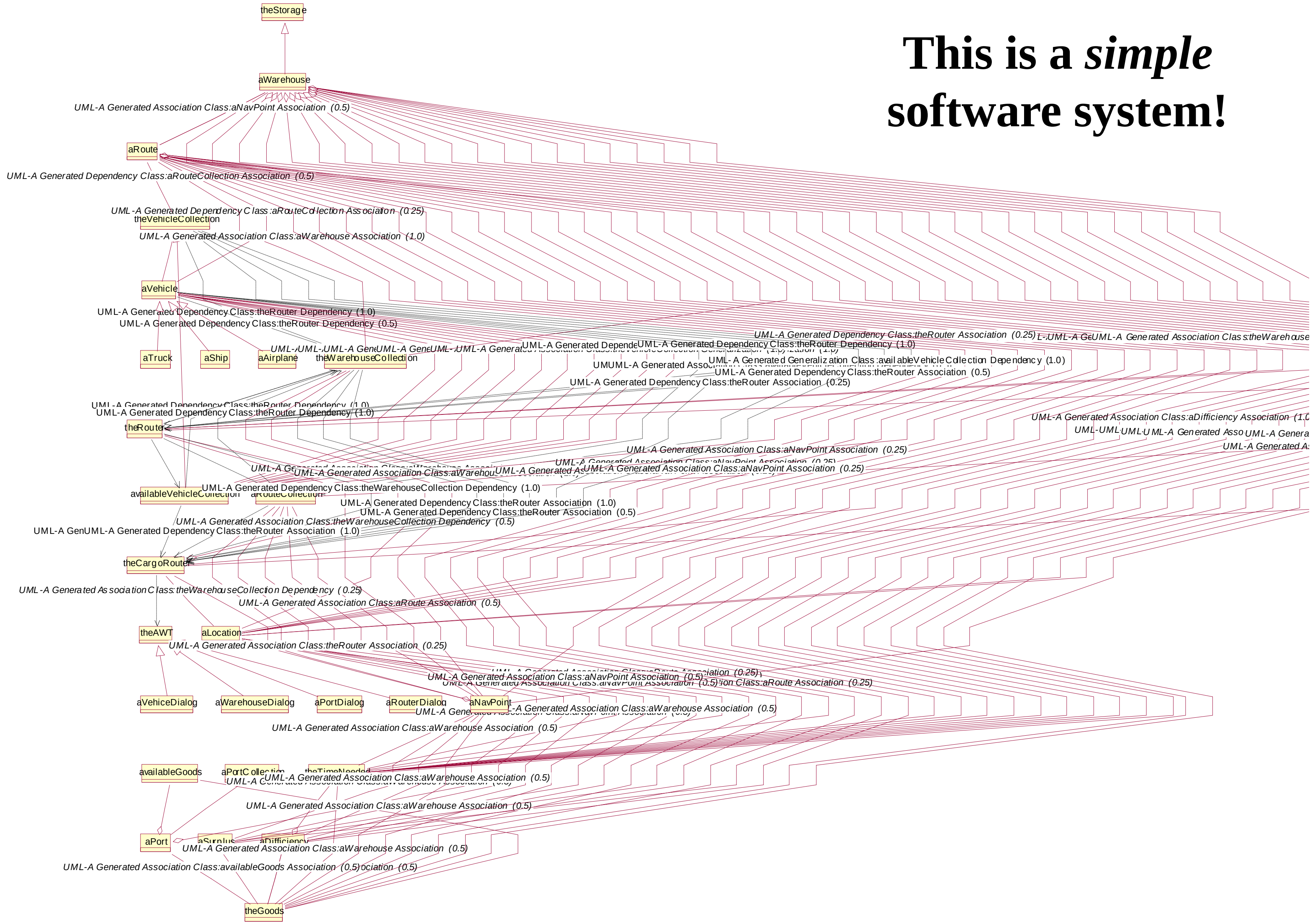
Why this programmer's approach?

- Design is a waste of time
- We need to show something to the customer real quick
- We are judged by the amount of LOC/month
- We expect or know that the schedule is too tight

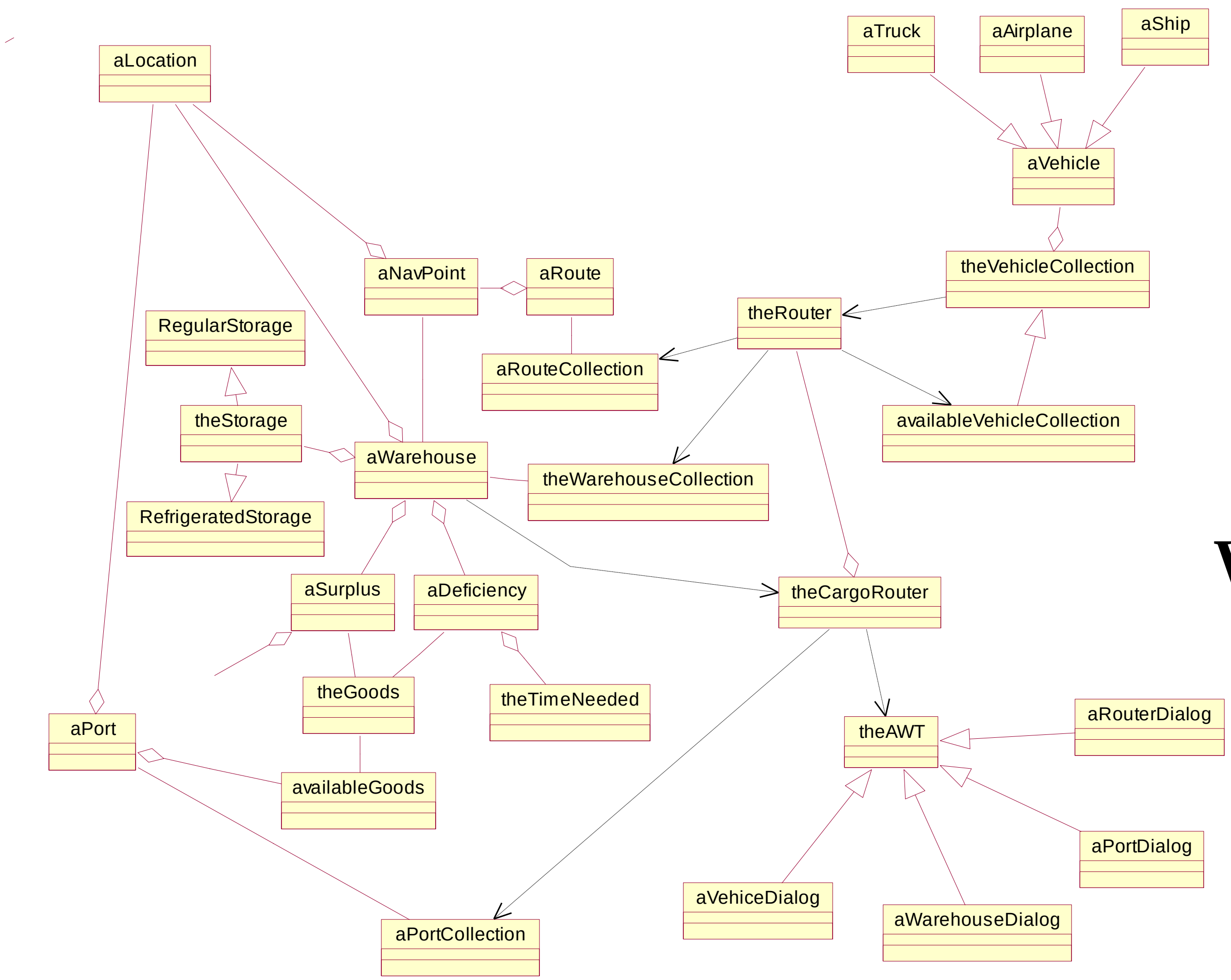
Design of small and large systems

What is the Problem?

This is a *simple* software system!



The Usual Tool: Design Abstraction



We have to do better!

What is not Design

- Design is not programming.
- Design is not modeling. Modeling is part of the architectural design.
- Design is not part of requirements.
- Where requirements finishes and where design starts ?.
- Requirements = What the system is supposed to do.
- Design = How the system is built.

What is Design (or Architecture?)

- A high-level model of a software system
  - Describes the structure, functionality and characteristics of the software system.
  - Understandable to many stakeholders
  - Allows evaluation of the system’s properties before it is built
  - Provides well understood tools and techniques for constructing the thing from its blueprint
- A software system’s blueprint
  - Its components
  - Their interactions
  - Their interconnections
- Which aspects of a software system are architecturally relevant?

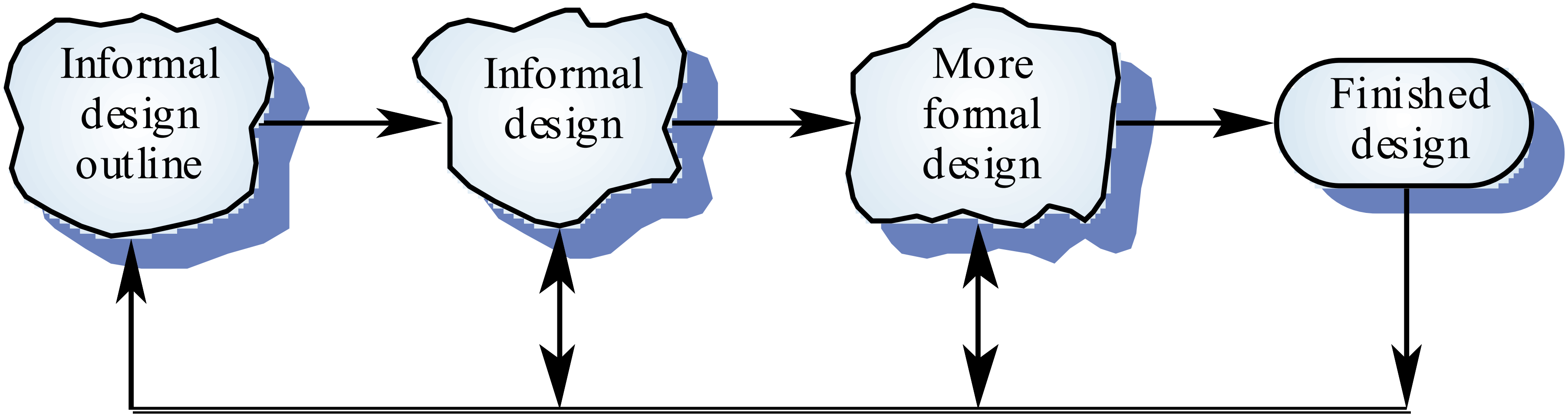
What is Design (or Architecture?)

- How should they be represented most effectively to enable stakeholders to understand, reason, and communicate about a system before it is built?
- What tools and techniques are useful for implementing an architecture in a manner that preserves its properties?
- We design the software but we must consider the hardware (and the environment).
- Design must reflect requirements, and we must be able to relate each requirements with parts of the design.
- How can we include non-functional requirements into the design?

Stages of design

- Problem understanding
  - Look at the problem from different angles to discover the design requirements
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design
- Repeat process for each identified abstraction until the design is expressed in primitive terms

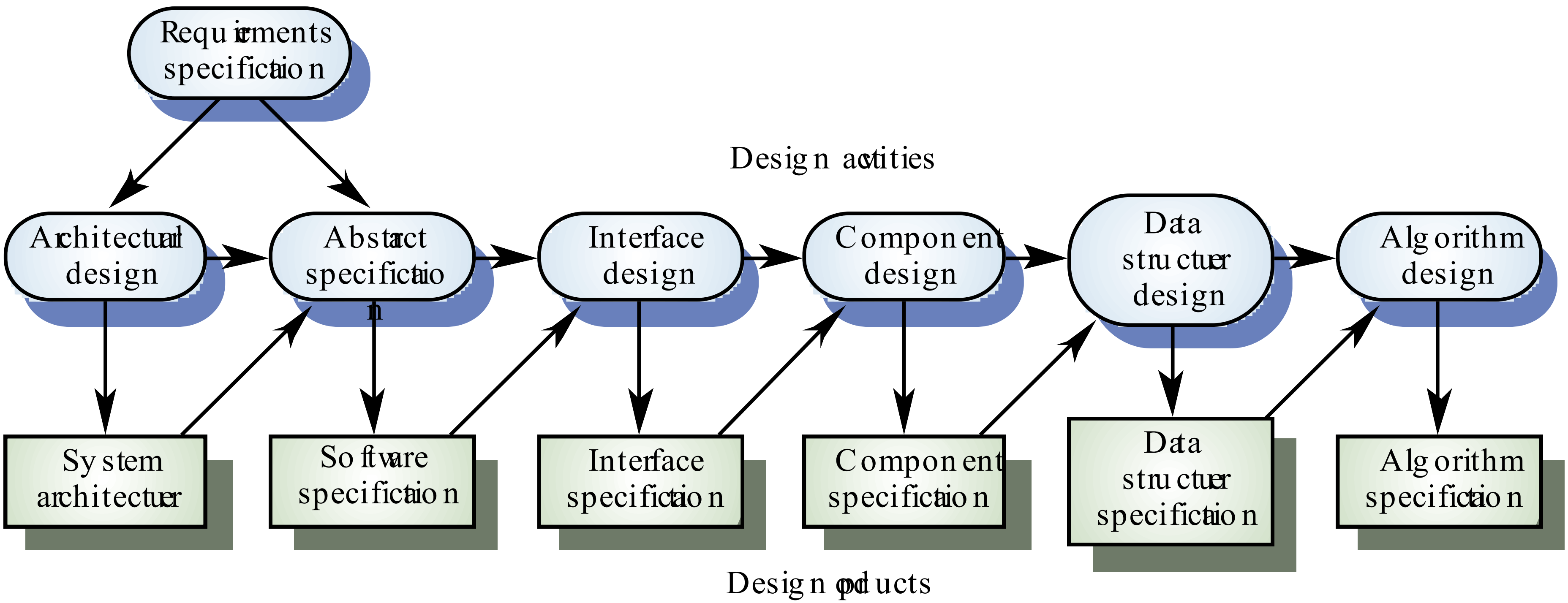
From informal to formal design



The design process

- The system should be described at several different levels of abstraction
- Design takes place in overlapping stages. It is artificial to separate it into distinct phases but some separation is usually necessary

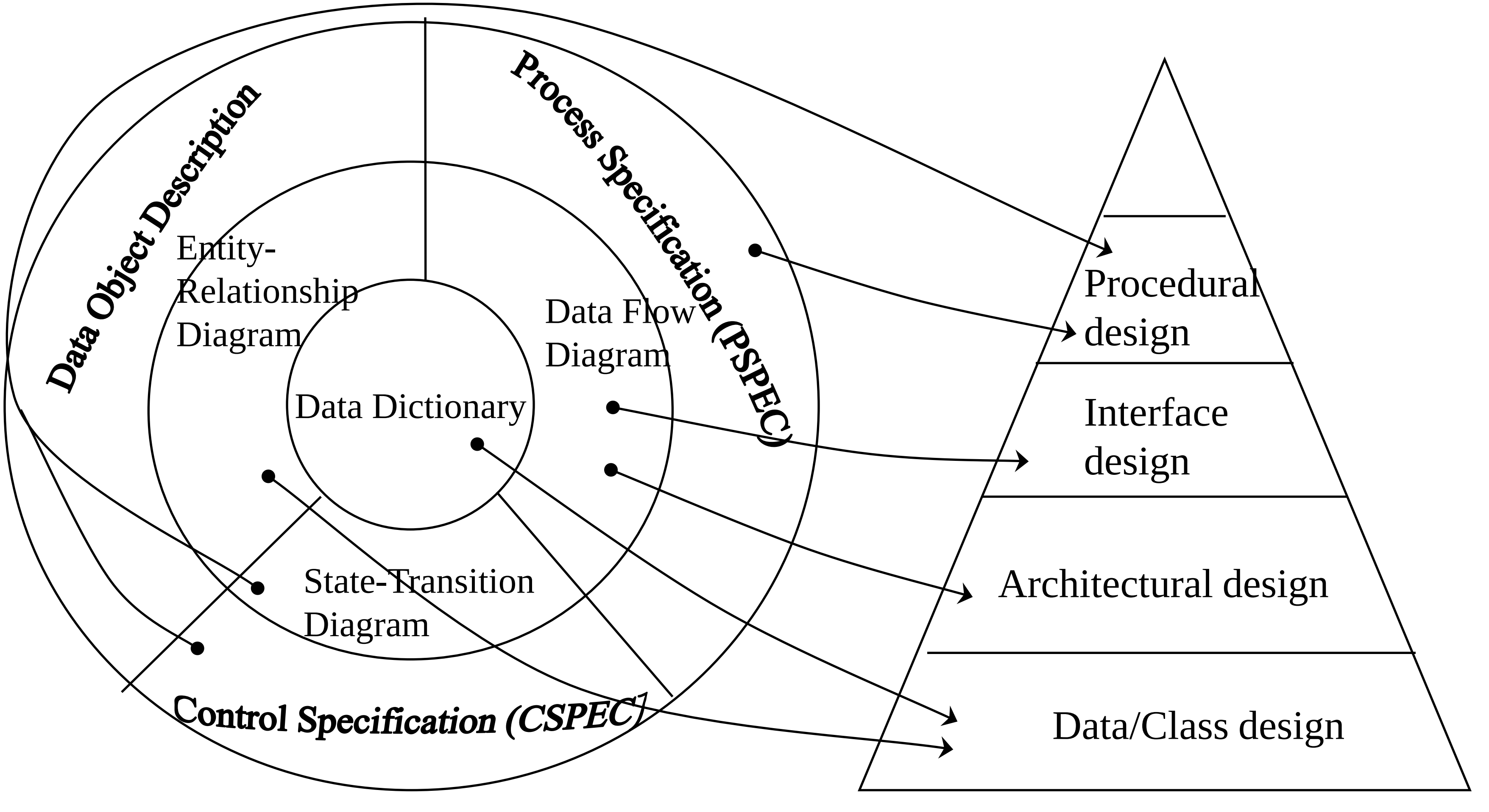
Phases in the design process



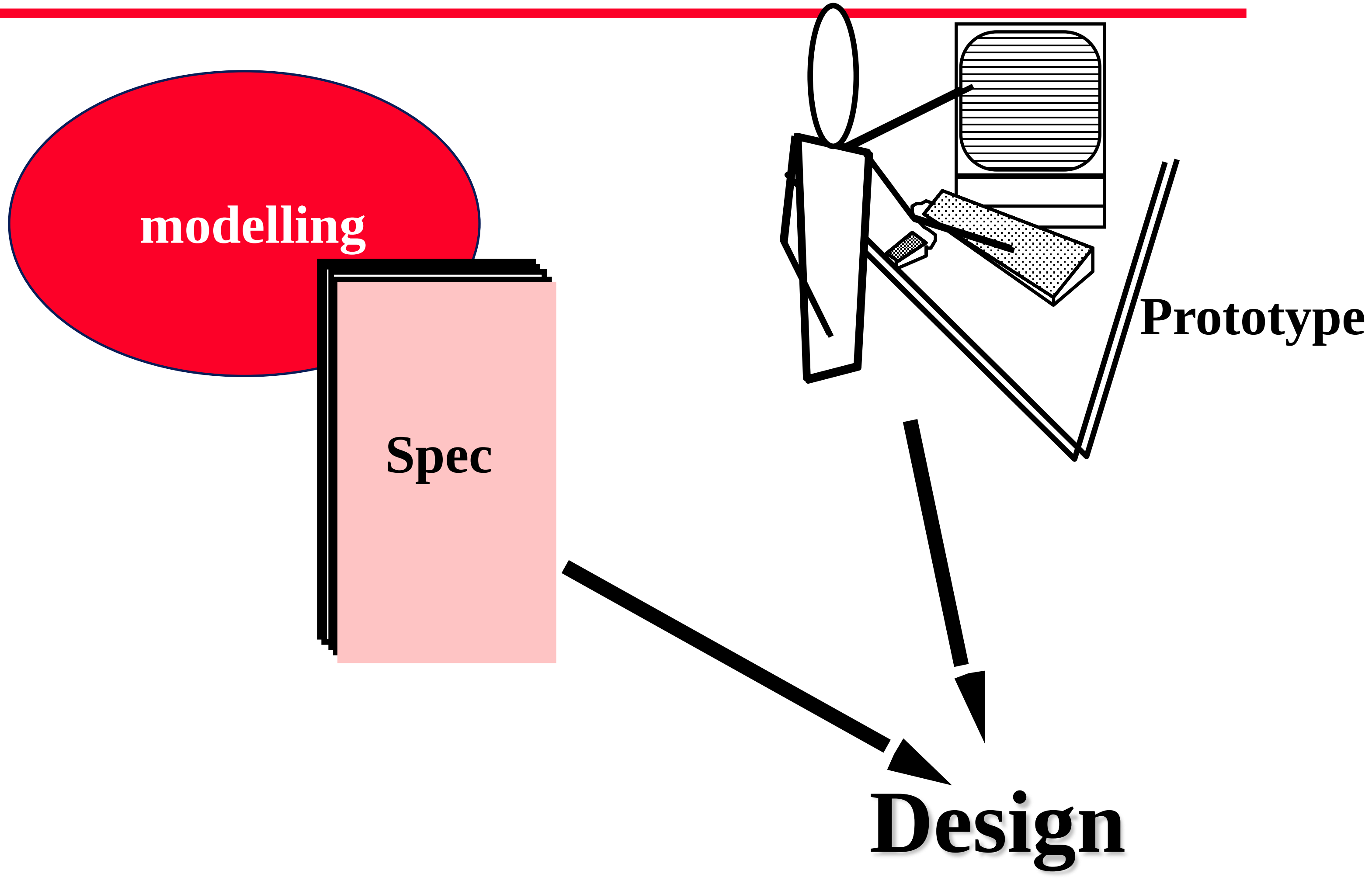
Design phases

- *Architectural design* Identify sub-systems
- *Abstract specification* Specify sub-systems
- *Interface design* Describe sub-system interfaces
- *Component design* Decompose sub-systems into components
- *Data structure design* Design data structures to hold problem data
- *Algorithm design* Design algorithms for problem functions

Relation between analysis and design



Where Do We Begin?



to experiment
to clarify
to understand
to analyse
to evaluate

Reason for modelling

What to model

structure
transformations
inputs and outputs
state

How to model

textual
graphical
mathematical

The Design Model

Component-level Design

(Class-based model, Flow-oriented model
Behavioral model)

Interface Design

(Scenario-based model, Flow-oriented model
Behavioral model)

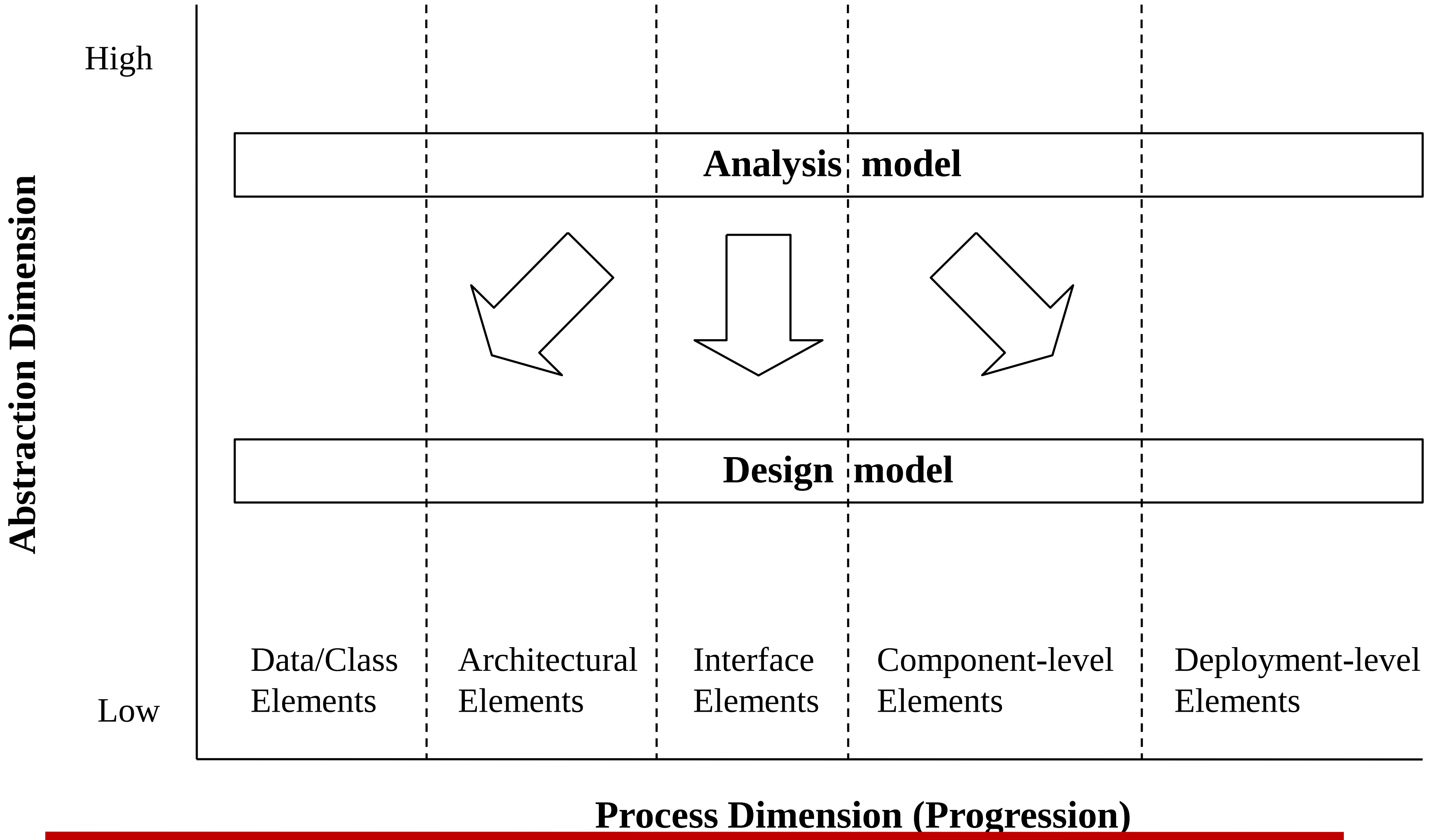
Architectural Design

(Class-based model, Flow-oriented model)

Data/Class Design

(Class-based model, Behavioral model)

Dimensions of the Design Model

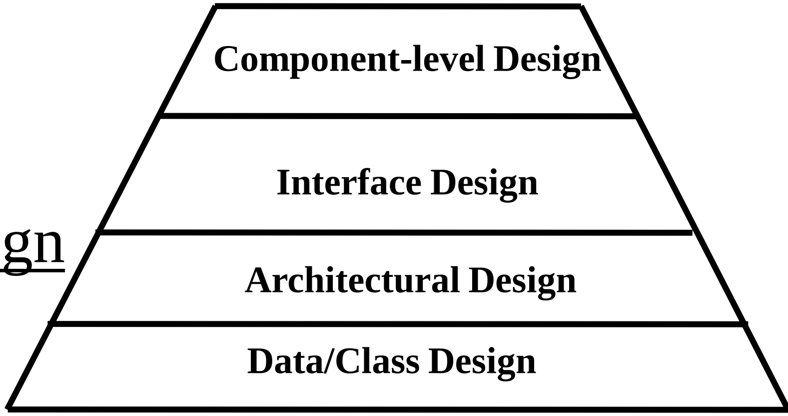


Introduction

- The design model can be viewed in two different dimensions
  - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed
  - (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Elements of the design model use many of the same UML diagrams used in the analysis model
  - The diagrams are refined and elaborated as part of the design
  - More implementation-specific detail is provided
  - Emphasis is placed on
    - » Architectural structure and style
    - » Interfaces between components and the outside world
    - » Components that reside within the architecture

Introduction (continued)

- Design model elements are <u>not always</u> developed in a <u>sequential</u> fashion
  - Preliminary architectural design sets the stage
  - It is followed by interface design and component-level design, which often occur <u>in parallel</u>
- The design model has the following layered elements
  - Data/class design
  - Architectural design
  - Interface design
  - Component-level design
- A fifth element that follows all of the others is <u>deployment-level design</u>



Design Elements

- Data/class design
  - Creates a model of data and objects that is represented at a high level of abstraction
- Architectural design
  - Depicts the overall layout of the software
- Interface design
  - Tells how information flows into and out of the system and how it is communicated among the components defined as part of the architecture
  - Includes the user interface, external interfaces, and internal interfaces
- Component-level design elements
  - Describes the internal detail of each software component by way of data structure definitions, algorithms, and interface specifications
- Deployment-level design elements
  - Indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software

Design Principles

1. Design process should not suffer from tunnel vision
2. Design should be traceable to the analysis model
3. Design should not reinvent the wheel
4. Design should exhibit uniformity and integration
5. Design should be structured to accommodate change
6. Design is not coding and coding is not design
7. Design should be reviewed to minimize conceptual errors

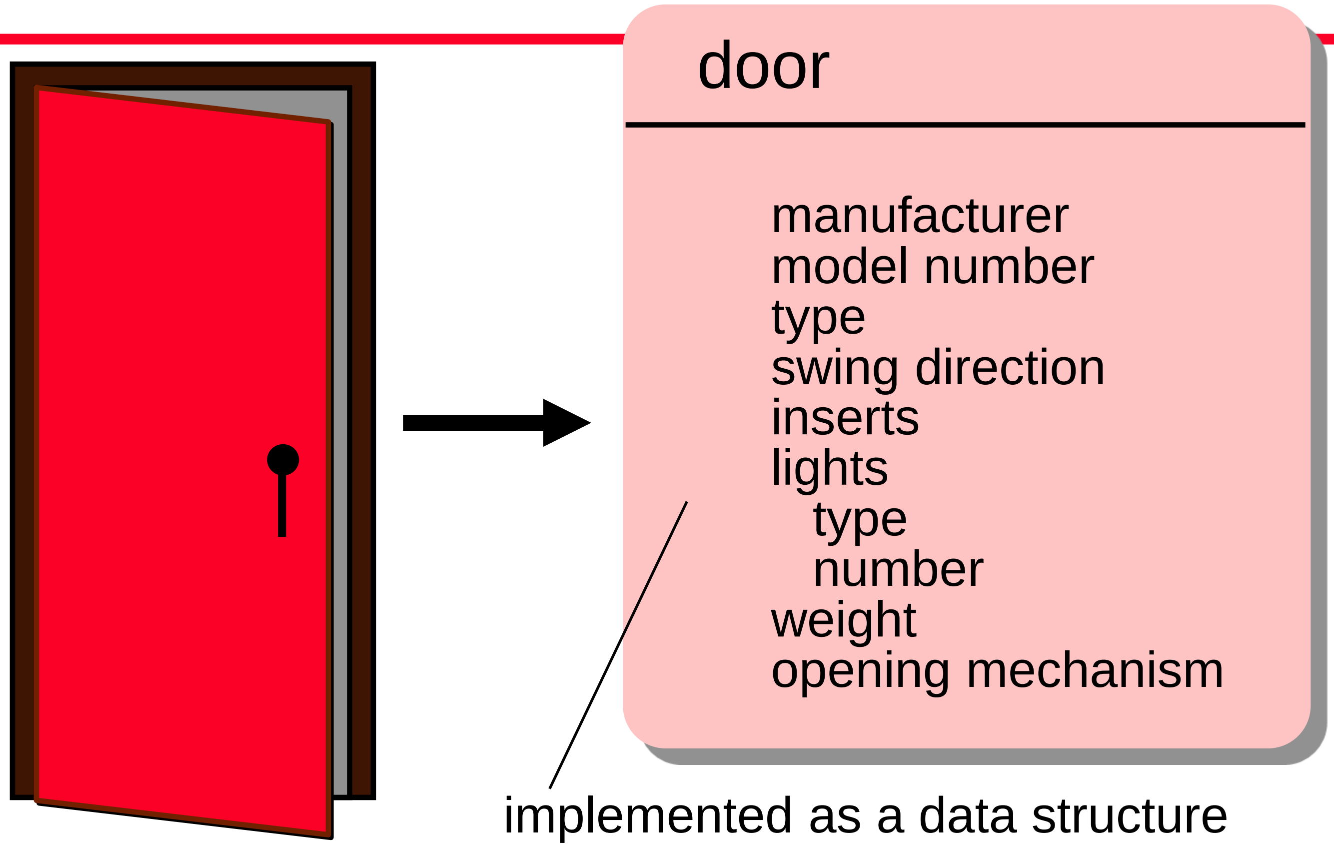
Design Concepts

- Abstraction
- Refinement
- Modularity
- Information hiding
- Functional Independence
  - coupling and cohesion
- Hierarchical structure
- Understandability
- Adaptability

Abstraction

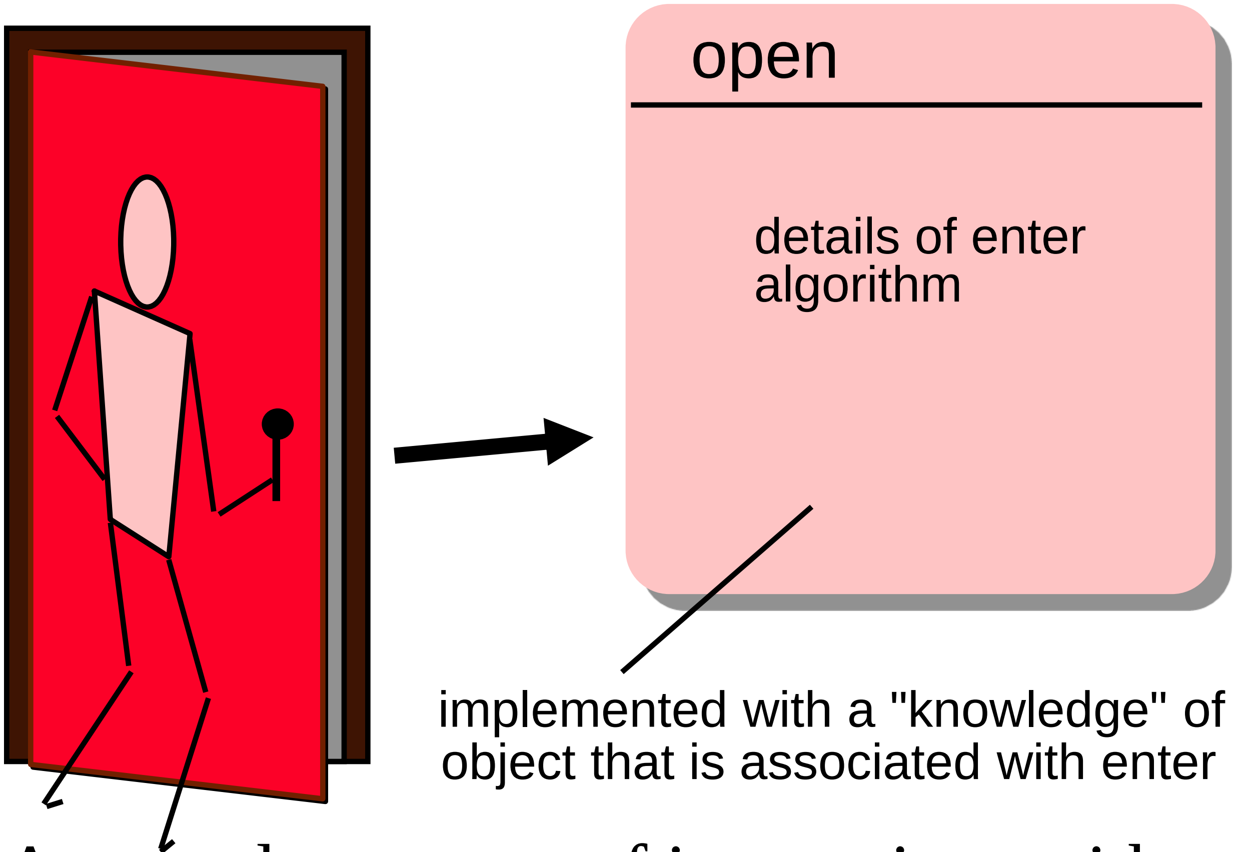
- Abstraction:
  - "*Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details..*"
  - Software Engineering is a process of refining abstractions
  - Modern programming languages allow for abstraction, e.g. abstract data types
  - Types: data and procedural

Data Abstraction



- A named collection of data that describes a data object

Procedural Abstraction

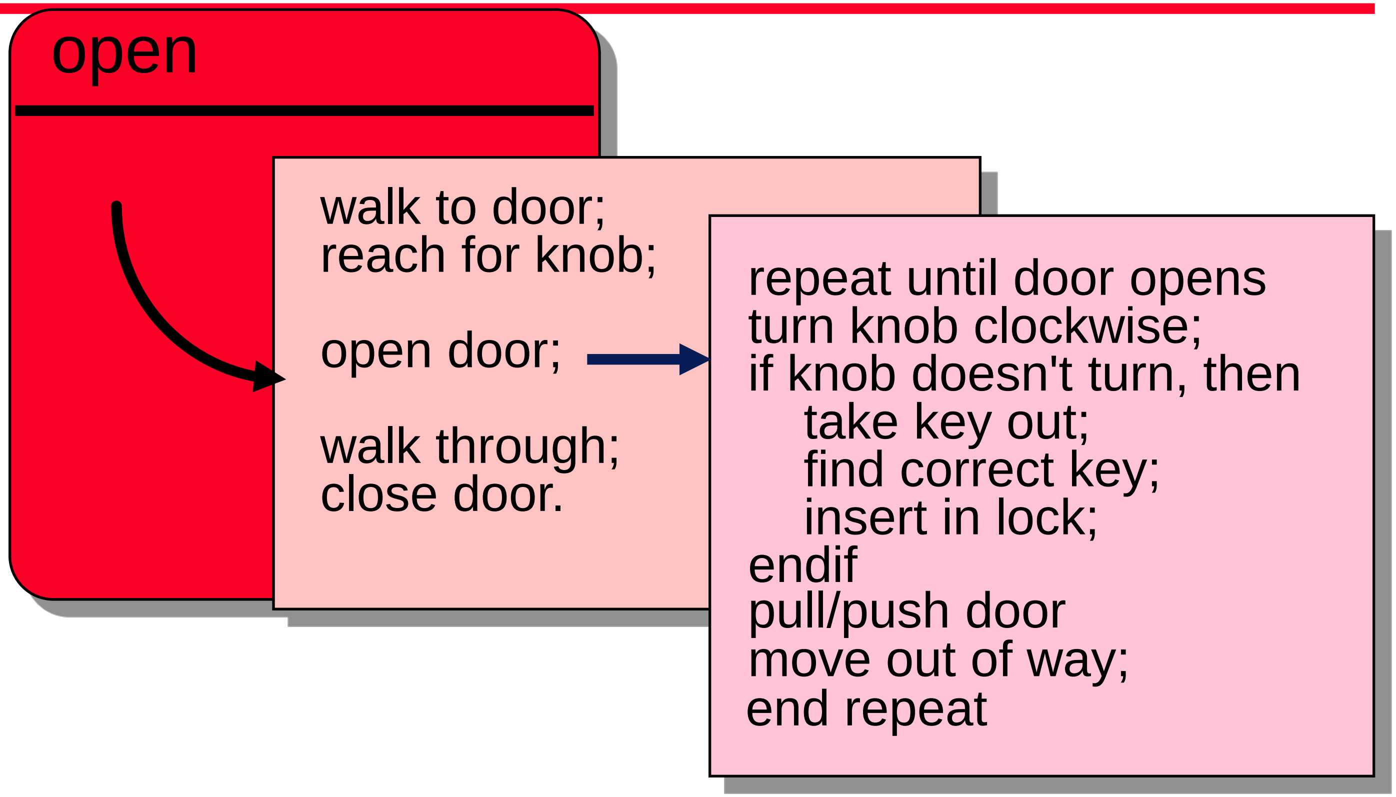


- A named sequence of instructions with a specific and limited function

Refinement

- Stepwise refinement
  - The simplest realistic design method, widely used in practice.
  - gradual top-down elaboration of detail
  - Abstraction and refinement are complementary. Abstraction suppresses low-level detail while refinement gradually reveals it.

Stepwise Refinement



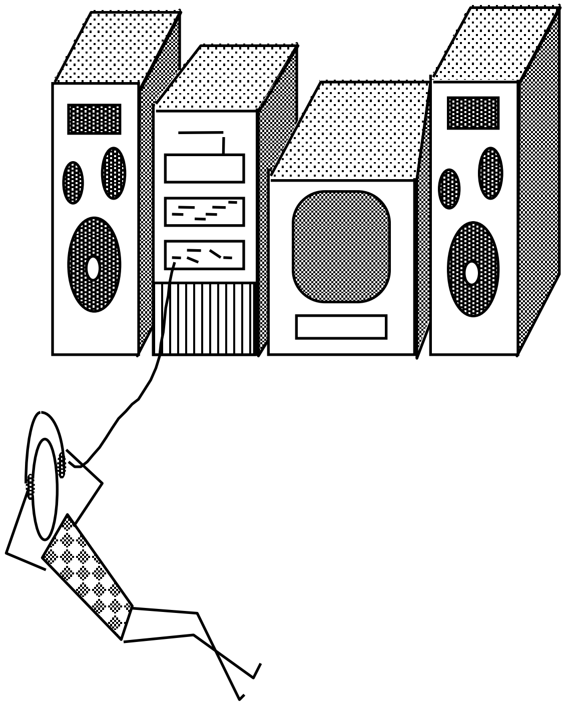
- Process of elaboration, proposed by Niklaus Wirth, that decomposes a high-level statement of function until low-level programming language statements are reached

Stepwise Refinement

- Not appropriate for large-scale, distributed systems: mainly applicable to the design of methods.
- Basic idea is:
  - Start with a high-level specification of what a method is to achieve;
  - Break this down into a small number of problems (usually no more than 10)
  - For each of these problems do the same;
  - Repeat until the sub-problems may be solved immediately.

Modularity

- Software should be split into separately named and addressable components
- A Module is "*a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier*"
- Procedures, functions and objects are all modules

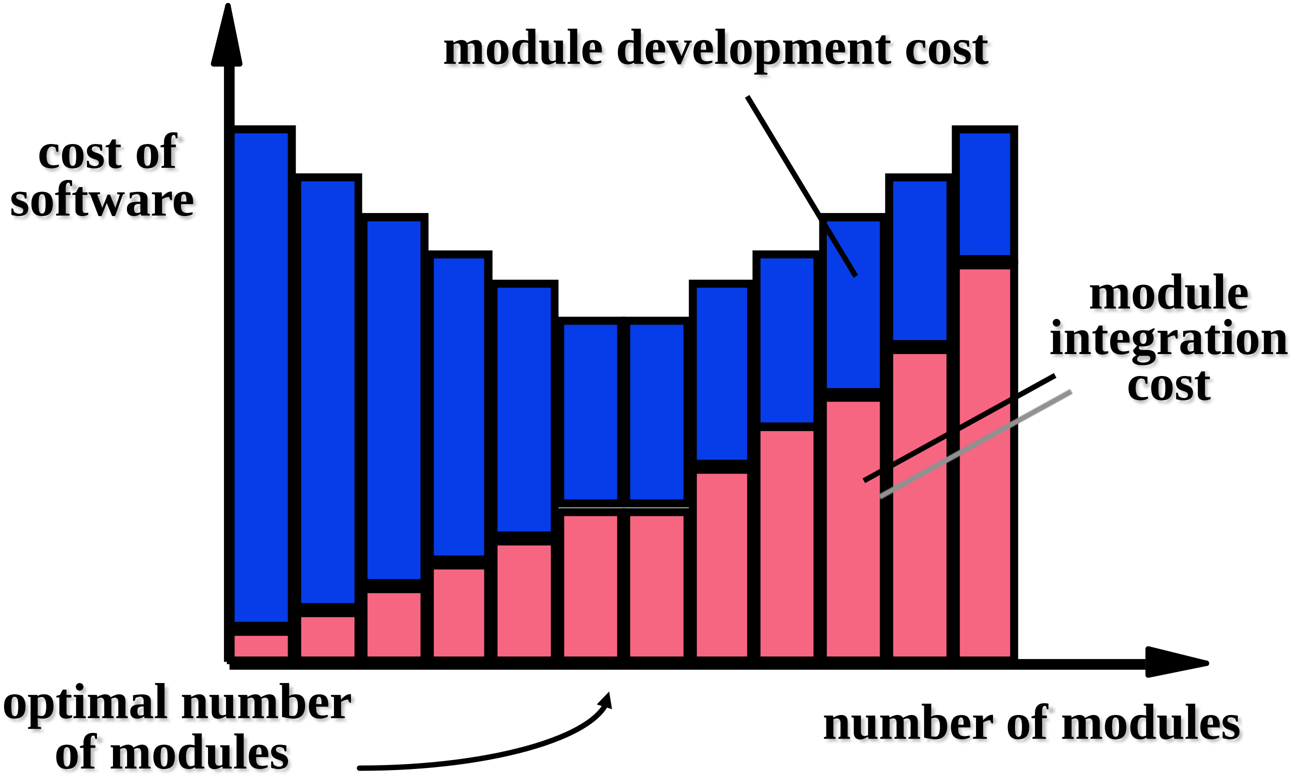


Benefits of Modularity

- Easier to build; easier to change; easier to fix
- “Modularity is the single attribute of software that allows a program to be intellectually manageable”
- Don’t overdo it. Too many modules makes integration complicated
- Sometimes the code must be monolithic (e.g. real-time and embedded software) but the design still shouldn’t be
- Effective modular design is achieved by developing “single minded” (highly cohesive) modules with an “aversion” to excessive interaction (low coupling)

Modularity: Trade-offs

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



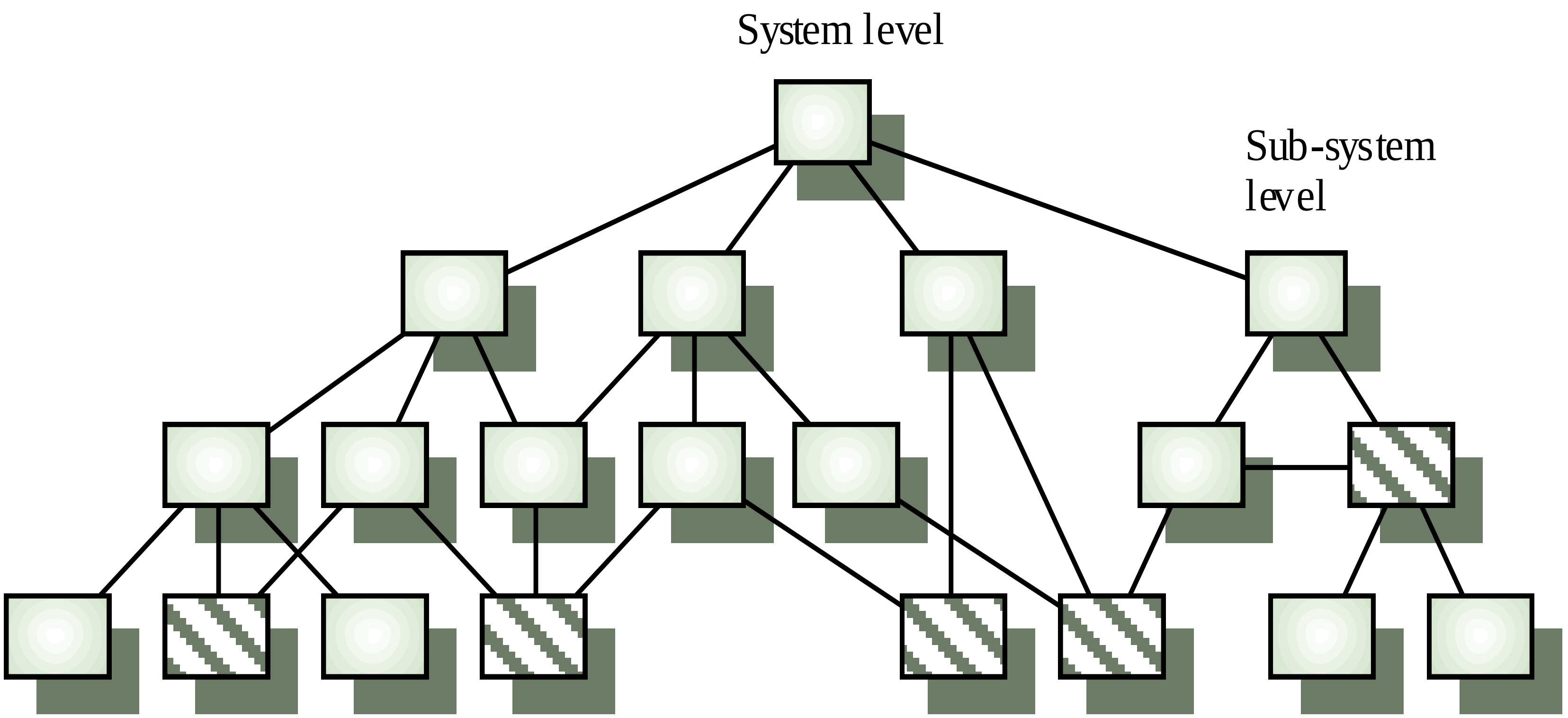
Modularity Support

- A design method supports effective modularity if it evidences:
  - Decomposability - a systematic mechanism for decomposing the problem
  - Composability - able to reuse modules in a new system
  - Understandability - the module can be understood as a standalone unit
  - Continuity - minimizes change-induced side effects
  - Protection - minimizes error-induced side effects

Hierarchical Design Structure

- In principle, top-down design involves starting at the uppermost components in the hierarchy and working down the hierarchy level by level
- In practice, large systems design is never truly top-down. Some branches are designed before others. Designers reuse experience (and sometimes components) during the design process

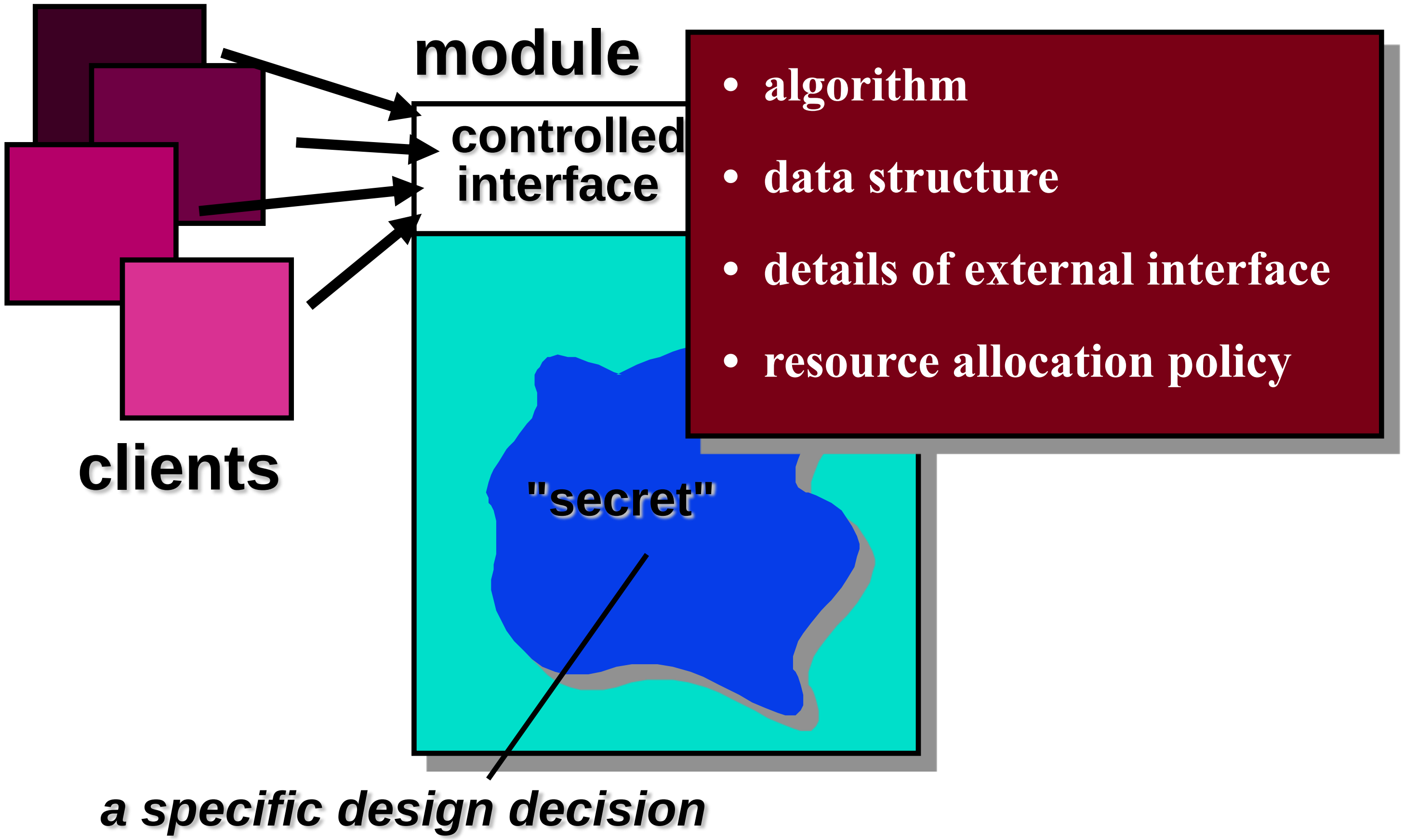
Hierarchical Design Structure



Information Hiding

- each module has a secret
- design involves a series of decision: for each such decision, wonder who needs to know and who can be kept in the dark
- This states:
  - All information about a module, (and particularly *how* the module does what it does) should be private to the module unless it is specifically declared otherwise.
- Thus each module should have some *interface*, which is how the world sees it anything beyond that interface should be hidden.
- The default Java rule:
  - Make everything private

Information Hiding



Functional Independence

- Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure
  - Benefits
    - » Easier to develop
    - » Easier to maintain & test
  - Measures of Independence
    - » Cohesion
      - measure of relative functional strength of a module
      - a cohesive module should (ideally) do just one thing/single task
    - » Coupling
      - measure of the relative interdependence among modules

Functional Independence

- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- By the term functional independence, we mean that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- Functional independence is a key to any good design primarily due to the following reasons:
- **Error isolation**: Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.

Functional Independence

- **Scope of reuse:** Reuse of a module becomes possible- because each module does some welldefined and precise function and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Understandability

- Related to several component characteristics
  - *Cohesion*. Can the component be understood on its own?
  - *Naming*. Are meaningful names used?
  - *Documentation*. Is the design well-documented?
  - *Complexity*. Are complex algorithms used?
- Informally, high complexity means many relationships between different parts of the design. hence it is hard to understand
- Most design quality metrics are oriented towards complexity measurement. They are of limited use

Adaptability

- A design is adaptable if:
  - Its components are loosely coupled
  - It is well-documented and the documentation is up to date
  - There is an obvious correspondence between design levels (design visibility)
  - Each component is a self-contained entity (tightly cohesive)
- To adapt a design, it must be possible to trace the links between design components so that change consequences can be analysed

Design traceability