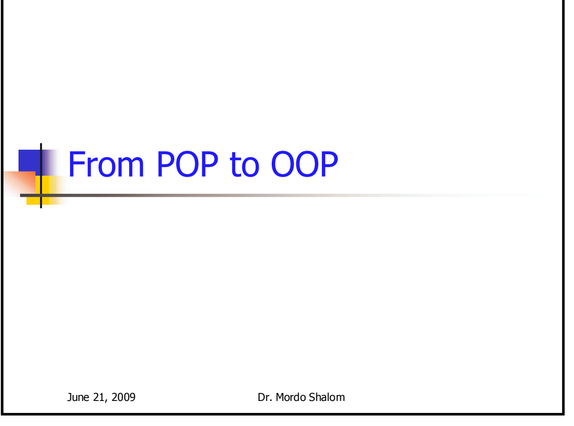


(Desired) Properties of software

- Is written once, modified a lot (provided that it is used).
- Is more read then written.
- Software should be written primarily for the sake of its readers (i.e. maintainers)
  - Different people
  - Different places
  - Different time

(Desired) Properties of software

- Assume you have two software systems A and B.
  - A – works, but not as readable as B
  - B – more readable then A, but has some bugs.

  (This is a hypothetic situation, usually better written software has less bugs.)
- Which one is better?

(Desired) Properties of software

- Answer: B is better because most probably:
  - When modified (and this will happen soon) A will contain bugs.
  - B can be easily debugged.

A comment on our examples

- Having this in mind, our programs are
  - examples of (hopefully) well-written programs,
  - not necessarily bug free.

A word about documentation

- A software should be:
  - Self-documenting:
    - meaningful and consistent naming. Is “flag” a good name?
    - clear flow of control:
      - write as you think
      - think naturally

      (things are usually straightforward, bad programmers complicate them)
  - Document everything which is not crystal clear:
    - Distinguish between:
      - external documentation, and
      - Internal documentation
  - Not over-documented.

  int counter = 0; // A counter, initialized to zero

About style

- Readability is achieved by good style in the low level:
  - Clearly define and use coding conventions.
    - Small functions (<= 30 lines)
    - Spacing and indentation
    - …

A simple Elevator Simulator

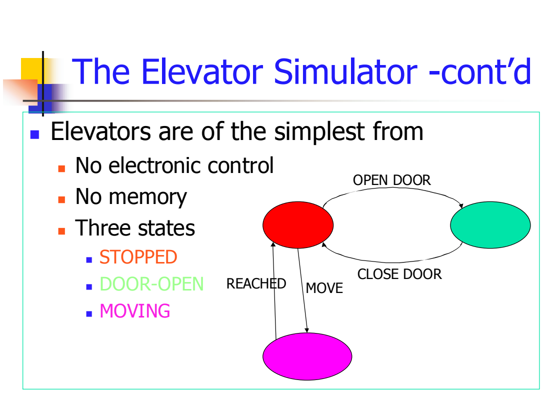
- A building:
  - Lowest floor (≤ 0)
  - Highest floor (≥ 0)
  - Number of elevators
  - Number of persons
- All elevators have the same characteristics:
  - Acceleration time (seconds)
  - Slow-down time (seconds)
  - Time between floors (seconds)

The Elevator Simulator -cont'd

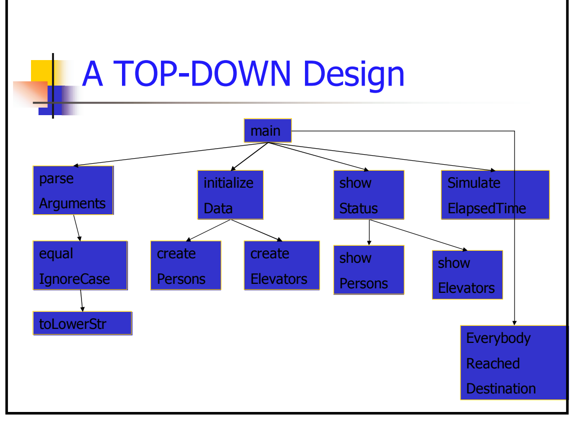
- Each Person wants to get
  - from his/her "From Floor"
  - to his/her "To Floor"
  - distributed uniformly between lowest floor and highest floor.
- All the persons are present at the beginning of the simulation
- Simulation ends when all the persons reach their destinations.

The Elevator Simulator -cont'd

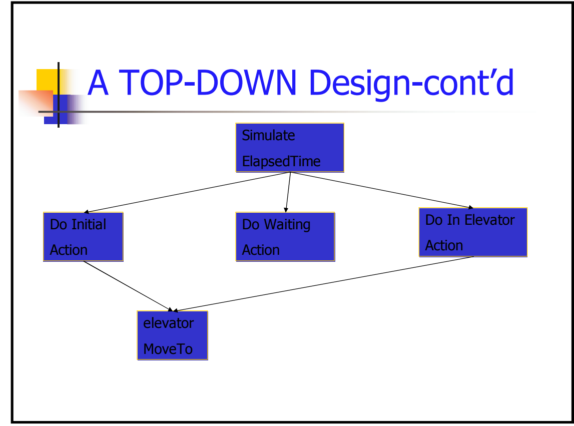
- Elevators are of the simplest from
  - No electronic control
  - No memory
  - Three states
    - STOPPED
    - DOOR-OPEN
    - MOVING

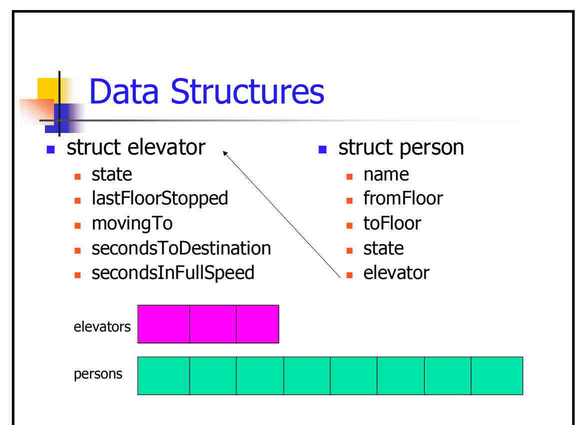


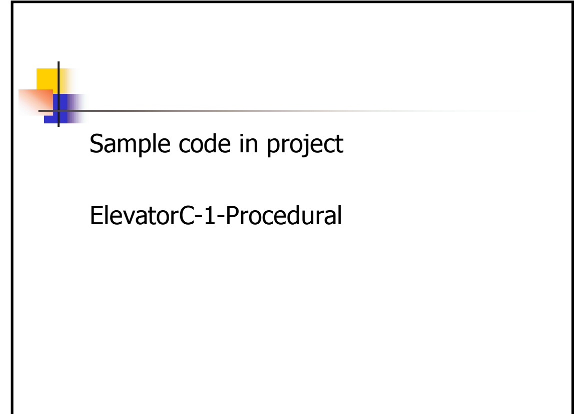
A TOP-DOWN Design

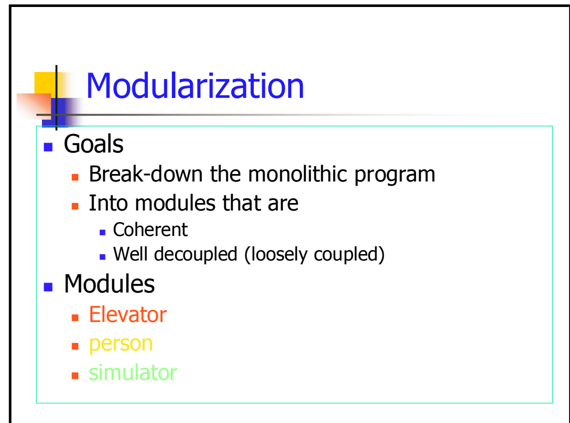


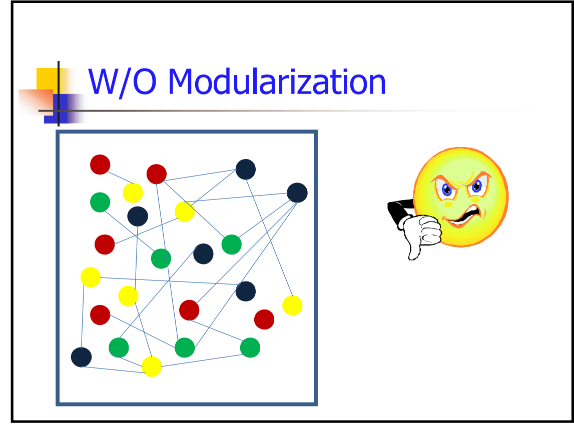
A TOP-DOWN Design-cont'd

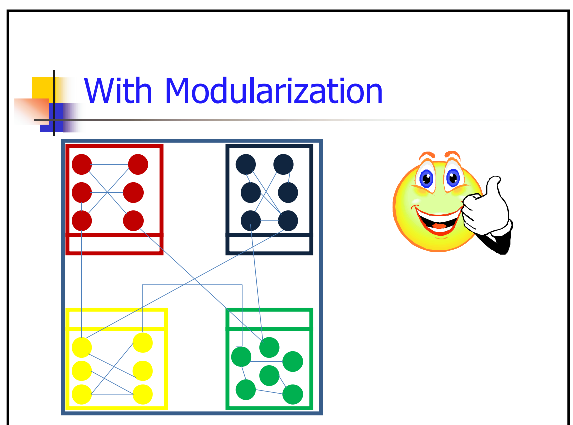


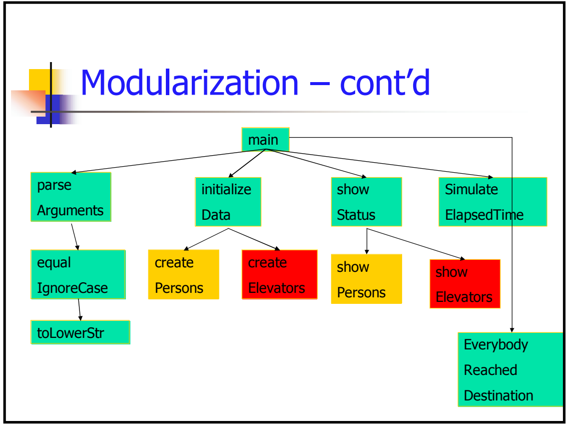

Data Structures
struct elevator
state
lastFloorStopped
movingTo
secondsToDestination
secondsInFullSpeed
struct person
name
fromFloor
toFloor
state
elevator
elevators
persons

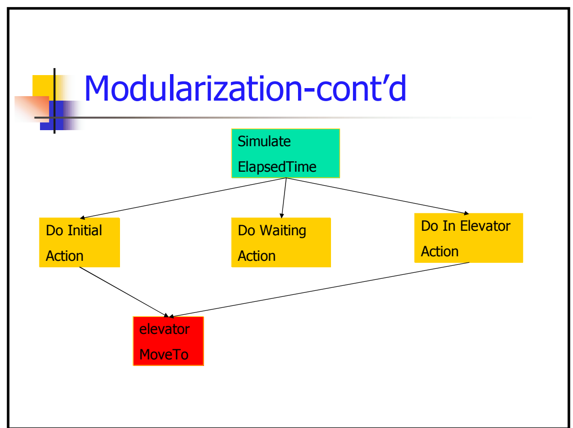

Sample code in project
ElevatorC-1-Procedural

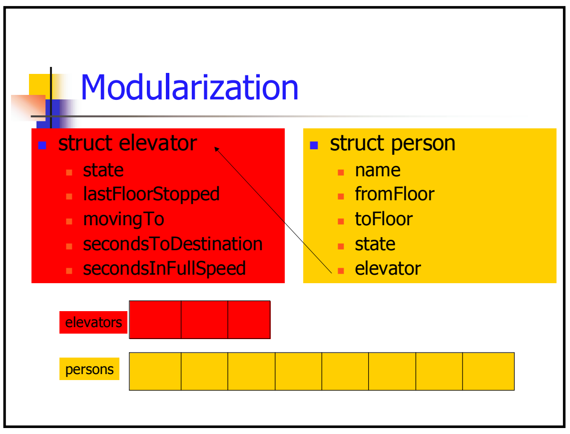

Modularization
Goals
Break-down the monolithic program
Into modules that are
Coherent
Well decoupled (loosely coupled)
Modules
Elevator
person
simulator

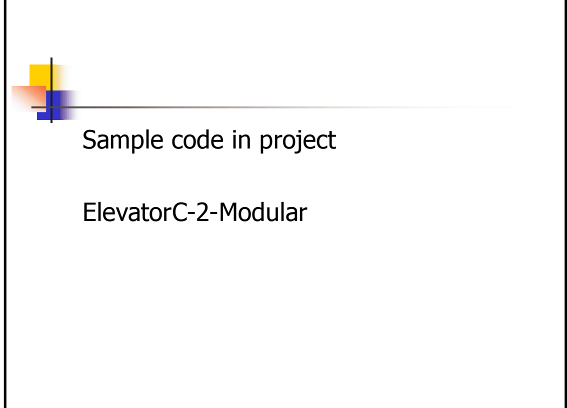

W/O Modularization

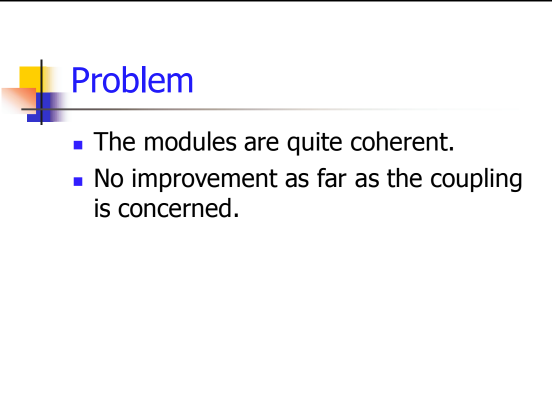

With Modularization

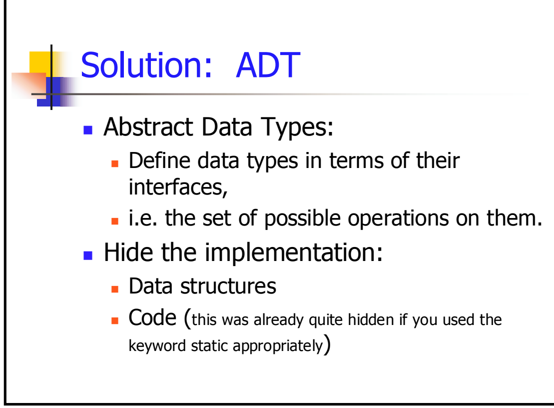

Modularization – cont'd
main
parse Arguments
initialize Data
show Status
Simulate ElapsedTime
equal IgnoreCase
create Persons
create Elevators
show Persons
show Elevators
toLowerStr
Everybody Reached Destination

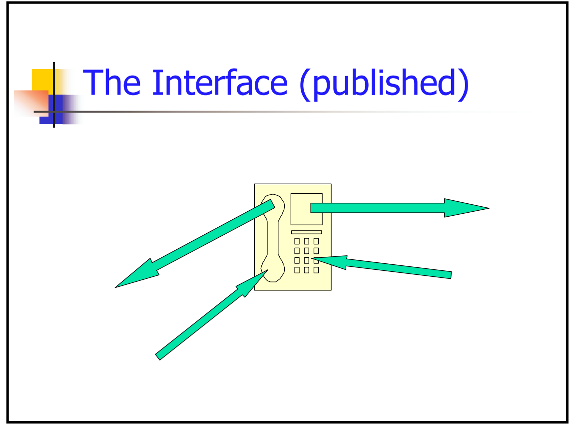

Modularization-cont'd
Simulate ElapsedTime
Do Initial Action
Do Waiting Action
Do In Elevator Action
elevator MoveTo


Modularization
struct elevator
state
lastFloorStopped
movingTo
secondsToDestination
secondsInFullSpeed
struct person
name
fromFloor
toFloor
state
elevator
elevators
persons

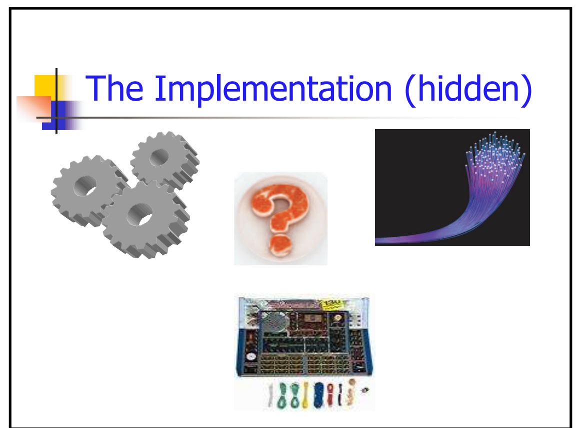

Sample code in project
ElevatorC-2-Modular


Problem
The modules are quite coherent.
No improvement as far as the coupling is concerned.

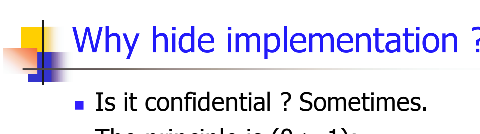

Solution: ADT
Abstract Data Types:
Define data types in terms of their interfaces,
i.e. the set of possible operations on them.
Hide the implementation:
Data structures
Code (this was already quite hidden if you used the keyword static appropriately)


The Interface (published)

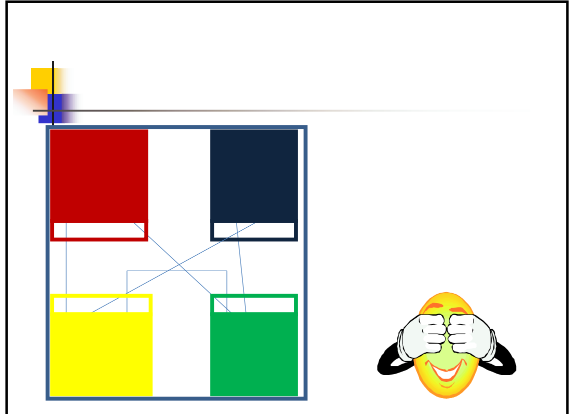
The Implementation (hidden)

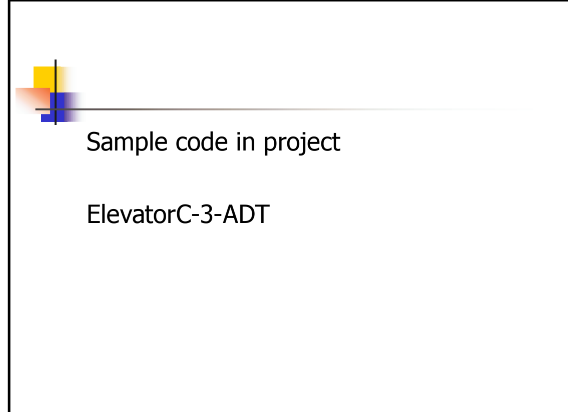


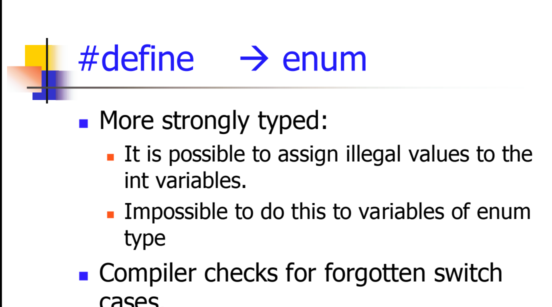
Why hide implementation ?

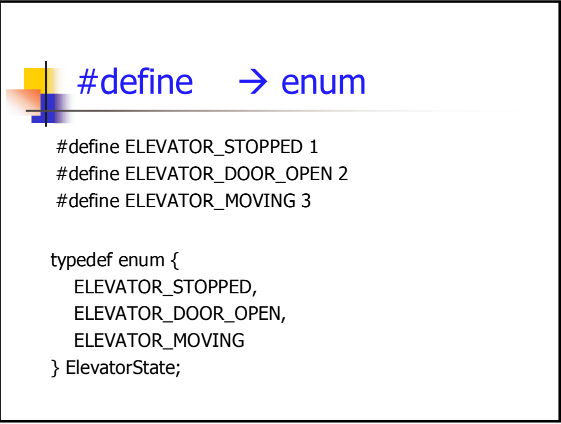


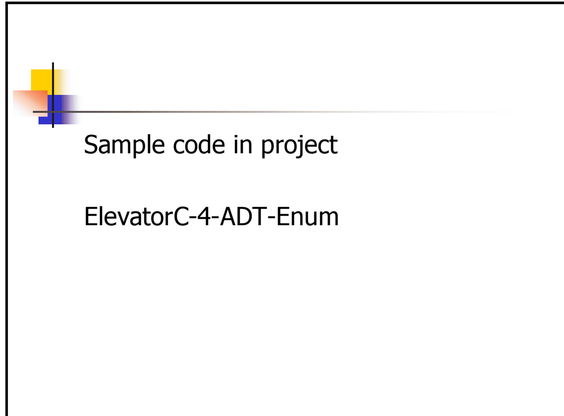
- Is it confidential ? Sometimes.
- The principle is (0 > 1):
  - What you don't know can not hurt you.
  - The less you know (i.e. you assume) the better.










Sample code in project
ElevatorC-4-ADT-Enum