SOFTWARE ENGINEERING

Week 8
Software Design Engineering II

1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

Object Oriented Design Principles

8.4

Object-Oriented Design Steps

- OOD consists of two steps:
- Step 1. Complete the class diagram
  - Determine the formats of the attributes
  - Assign each method, either to a class or to a client that sends a message to an object of that class
- Step 2. Perform the detailed design

The Elevator Problem Case Study

A product is to be installed to control n elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

1. Each elevator has a set of m buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited by the elevator
2. Each floor, except the first and the top floor, has two buttons, one to request an up-elevator, one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor, then moves in the desired direction
3. If an elevator has no requests, it remains at its current floor with its doors closed

The Elevator Problem Case Study

- There are two sets of buttons
- Elevator buttons
  - In each elevator, one for each floor
- Floor buttons
  - Two on each floor, one for up-elevator, one for down-elevator

First Iteration of Class Diagram

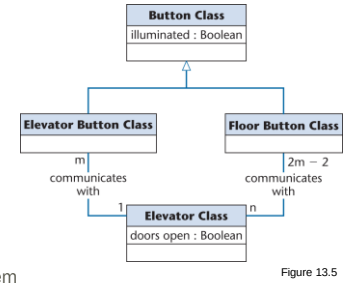


Figure 13.5

- Problem
  - Buttons do not communicate directly with elevators
  - We need an additional class: **Elevator Controller Class**

OOD: Elevator Problem Case Study

- Step 1. Complete the class diagram
- Consider the CRC card

CLASS **Elevator Controller Class**
RESPONSIBILITY 1. Send message to **Elevator Button Class** to turn on button 2. Send message to **Elevator Button Class** to turn off button 3. Send message to **Floor Button Class** to turn on button 4. Send message to **Floor Button Class** to turn off button 5. Send message to **Elevator Class** to move up one floor 6. Send message to **Elevator Class** to move down one floor 7. Send message to **Elevator Doors Class** to open 8. Start timer 9. Send message to **Elevator Doors Class** to close after timeout 10. Check requests 11. Update requests
COLLABORATION 1. **Elevator Button Class** (subclass) 2. **Floor Button Class** (subclass) 3. **Elevator Doors Class** 4. **Elevator Class**

Figure 13.9 (again)

Second Iteration of Class Diagram

- All relationships are now 1-to-n
  - This makes design and implementation easier

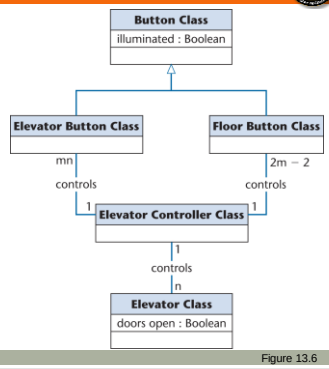


Figure 13.6

OOD: Elevator Problem Case Study

- Responsibilities
  - 8. Start timer
  - 10. Check requests, and
  - 11. Update requests

  are assigned to the elevator controller
- Because they are carried out by the elevator controller

OOD: Elevator Problem Case Study

- The remaining eight responsibilities have the form
  - "Send a message to another class to tell it do something"
- These should be assigned to that other class
  - Responsibility-driven design
  - Safety considerations
- Methods `open doors`, `close doors` are assigned to class **Elevator Doors Class**
- Methods `turn off button`, `turn on button` are assigned to classes **Floor Button Class** and **Elevator Problem Class**

Third Iteration of Class Diagram

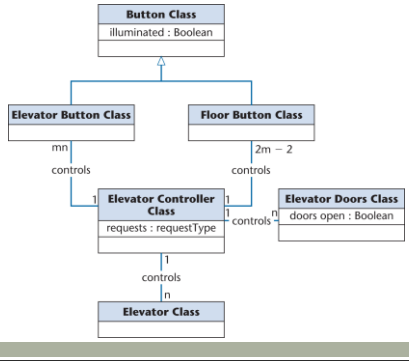


Figure 13.10

Fourth Iteration of Class Diagram

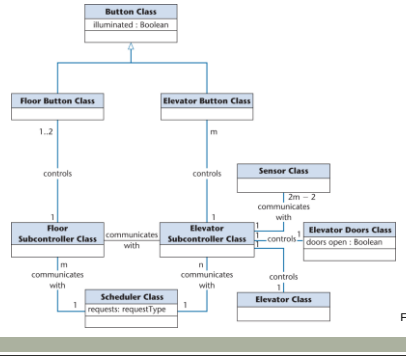


Figure 13.12

Detailed Class Diagram: Elevator Problem

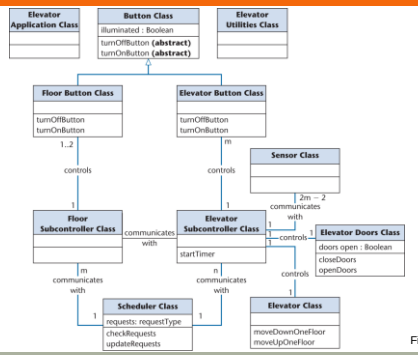


Figure 14.11

Dynamic Modeling: The Elevator Problem Case Study

- Produce a UML statechart
- State, event, and predicate are distributed over the statechart

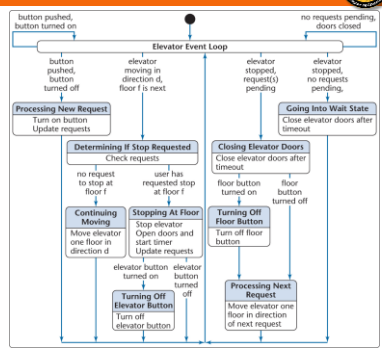


Figure 13.7

Detailed Design: Elevator Problem

- Detailed design of `elevatorEventLoop` is constructed from the statechart

```
void elevatorSubcontrollerEventLoop (void)
{
  while (TRUE)
  {
    if (an elevatorButton has been pressed)
      if (elevatorButton is off)
      {
        elevatorButton::turnOnButton;
        scheduler::newRequestMade;
      }
    else if (elevator is moving up)
    {
      wait for sensor message that elevator is arriving at floor;
      scheduler::checkRequests;
      if (there is no request to stop at floor f)
        elevator::moveUpOneFloor;
      else
      {
        stop elevator by not sending a message to move;
        if (elevatorButton is on)
          elevatorButton::turnOffButton;
        elevatorDoors::openDoors;
        startTimer;
      }
    }
    else if (elevator is moving down)
      [similar to up case]
    else if (elevator is stopped and request is pending)
    {
      wait for timeout;
      elevatorDoors::closeDoors;
      determine direction of next request;
      elevator::moveUp/DownOneFloor;
      wait for sensor message that elevator has left floor;
      floorSubcontroller::elevatorHasLeftFloor;
    }
    else if (elevator is at rest and not (request is pending))
    {
      wait for timeout;
      elevatorDoors::closeDoors;
    }
    else
      /* there are no requests, elevator is stopped with elevatorDoors closed, so do nothing */
  }
}
```

Figure 14.12

Design Principles

- Software modules should be in a hierarchical organization.
- Software should be modular, that is, the software should be logically partitioned into elements that perform specific functions.
- Should contain both data abstraction and procedural abstraction.
- Should lead to interfaces that reduce the complexity of connections between modules (low coupling).
- Must be an understandable guide for coders, testers and maintainers.
- Should exhibit uniformity and integration.

16

Design Strategies

- **Stepwise refinement:**
  - It is a top-down design strategy.
  - A higher level abstraction is refined to a lower level abstraction with more details in each step.
  - Several steps are taken.
- **Top-down design:**
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed individual algorithms that will be used.
- **Divide and conquer:**
  - Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things .
  - Separate people can work on each part.
  - Each individual component is smaller, and therefore easier to understand.

17

Cohesion

- **Cohesion** is a measure of dependencies ***within*** a module.
- If a module contains many closely related functions its cohesion is high.
- The designer should aim **high cohesion.**
  - Module is understandable as a meaningful unit
  - Functions of a module are closely related to one another
  - This makes the system as a whole easier to understand and change

1. Functional cohesion	(best)
2. Informational cohesion	(desirable)
3. Communicational cohesion	
4. Procedural cohesion	
5. Temporal cohesion	
6. Logical cohesion	(should be avoided)
7. Coincidental cohesion	(worst)

18

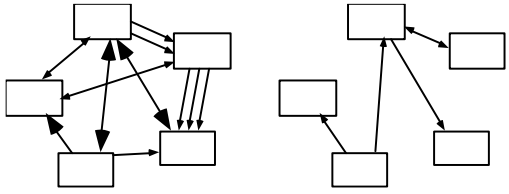
Coupling

- **Coupling** is a measure of the dependencies ***between*** two modules.
- If two modules are strongly coupled, it is hard to modify one without modifying the other.
- The designer should aim **low coupling.**
  - Modules have low interactions with others
  - Understandable separately

1. Data coupling
2. Stamp coupling
3. Control coupling
4. Common coupling
5. Content coupling

19

Coupling



High coupling Low coupling

20

1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

User Interface Design

8.5

Aspects of usability

- **Usability:** The system should allow the user to learn and to use the basic capabilities easily.
- Usability can be divided into separate aspects:
  - ***Learnability***
    - The speed with which a new user can become proficient with the system.
  - ***Efficiency of use***
    - How fast an expert user can do their work.
  - ***Error handling***
    - The extent to which it prevents the user from making errors, detects errors, and helps to correct errors.
  - ***Acceptability***
    - The extent to which users like the system.

22

Terminology of Graphical User Interface (GUI)

- **Dialog**: A specific window with which a user can interact, but which is not the main UI window.
- **Control** or **Widget:** Specific components of a user interface.
- **Affordance**: The set of operations that the user can do at any given point in time.
- **State**: At any stage in the dialog, the system is displaying certain information in certain widgets, and has a certain affordance.
- **Mode**: A situation in which the UI restricts what the user can do.
- **Modal dialog**: A dialog in which the system is in a very restrictive mode.
- **Feedback**: The *response from the system* whenever the user does something, is called feedback.
- **Encoding techniques**. Ways of encoding information so as to communicate it to the user.

23

User Interface Design Principles

Principle	Description
User familiarity	Use terms and concepts which are drawn from the experienced users.
Consistency	Be consistent in that, similar operations should be activated in the same way.
Recoverability	Include mechanisms to allow users to recover from errors.
User guidance	Provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	Provide appropriate interaction facilities for different types of users (such as clerk or manager).

24

Usability Principles (1)

1. Base the User Interface designs on users' *tasks*.
   - Perform use case analysis to structure the UI.
2. Ensure that the sequences of actions to achieve a task are as *simple* as possible.
   - Reduce the amount of manipulation the user has to do.
   - Ensure the user does not have to navigate anywhere to do subsequent steps of a task.
3. Ensure that the user always knows what he should do next.
   - Ensure that the user can see *what commands are available*.
   - Make the *most important commands stand out*.

25

Usability Principles (2)

4. Provide good feedback including effective error messages.
   - Inform users of the *progress* of operations and of their *location* as they navigate.
   - When something goes wrong, explain the situation in adequate detail and *help the user to resolve the problem*.
5. Ensure that the user can always get out, go back or undo an action.
   - Ensure that all operations can be *undone*.
   - Ensure it is easy to *navigate back* to where the user came from.
6. Ensure that response time is adequate.
   - Keep response time less than a second for most operations.
   - Warn users of longer delays and inform them of progress.

26

Usability Principles (3)

7. Use *understandable encoding techniques*.
   - Choose encoding techniques with care.
   - Use labels to ensure all encoding techniques are fully understood by users.
8. Ensure that the UI's appearance is *uncluttered*.
   - Avoid displaying too much information.
   - Organize the information effectively.
   - Use consistent language and meaningful keywords
   - Avoid abbreviations
   - Make text readable, use both upper and lower case
   - Use colors and graphics effectively

27

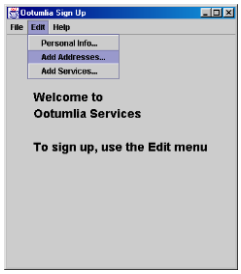
Usability Principles (4)

9. Robustness.
   - Minimize keystroke and mouse travel distance
   - Provide defaults for missing data (e.g. current date)
   - Automatically correct the obvious errors
10. Provide all necessary *help*.
    - Integrate help with the application.
    - Ensure that the help is accurate.
11. Be *consistent and uniform*.
    - Use similar layouts and graphic designs throughout your application.
    - Follow look-and-feel standards.

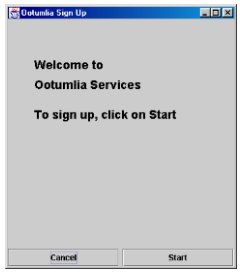
28

Example: Welcome screen

Wrong



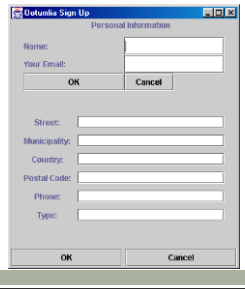
Correct



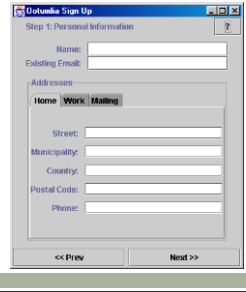
29

Example: Personal information screen

Wrong



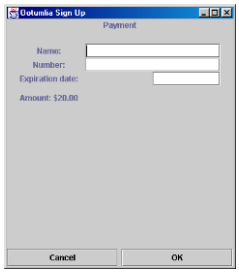
Correct



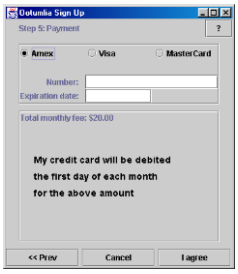
30

Example: Payment screen

Wrong



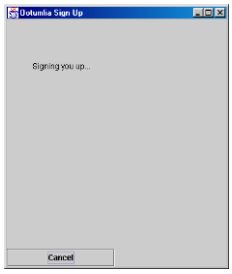
Correct



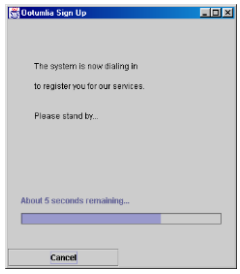
31

Example: Sign up screen

Wrong



Correct



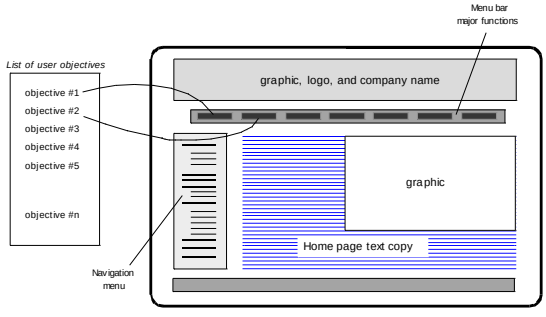
32

User Interface Patterns

- Many standard patterns are available for
  - Page layout
  - Page elements
  - Forms and input
  - Tables
  - Direct data manipulation
  - Navigation
  - Searching

33

Example: Web page layout



34

1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

Case Study: SafeHome

§ 8.6

SafeHome Product Definition

The product, called SafeHome, is a microprocessor based home security system (**embedded**) that would protect against burglary, fire, flooding and others.

- It will be configured by the homeowner.
- It will use appropriate sensors to detect each emergency situation.
- It will automatically make a telephone call to a monitoring agency (police, fire brigade) when a situation is detected.

36

Statement of Software Scope (1)

SafeHome software ***enables*** the homeowner to ***configure*** the security system when ***installed***, ***monitors*** all sensors ***connected*** to the security system, and ***interacts*** with the homeowner through a keypad and function keys ***contained*** in the SafeHome control panel.

During installation, the SafeHome control panel is ***used to "program" and configure*** the system. Each sensor is ***assigned*** a number and type, a master password for ***arming*** and ***disarming*** the system, and telephone numbers are ***input for dialing*** when a sensor event occurs.

- Data objects: Underlined nouns
- *Processes: Italic verbs*

37

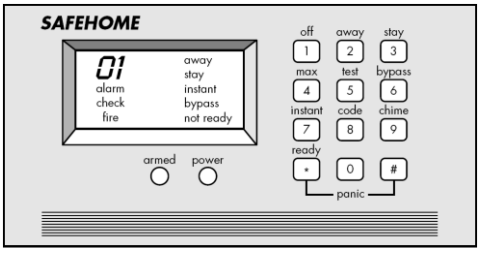
Statement of Software Scope (2)

When a sensor event is ***recognized***, the software ***invokes*** an audible alarm attached to the system. After a delay time, that is ***specified*** by the homeowner during the system configuration activities, the software dials a telephone number of a monitoring service agency, ***provides*** information about the location, ***reporting*** the nature of the event that has been detected. The telephone number will be ***redialed*** every 20 seconds until telephone connection is ***obtained***.

All interaction with SafeHome is ***managed*** by a user-interaction subsystem that ***reads*** input provided through the keypad and function keys, ***displays*** prompting messages and system status on the LCD display. Keyboard interactions takes the following form: (continues...)

38

SafeHome Control Panel



39

Customer Requirements

Objects:
- Smoke detectors
- Door and window sensors
- Motion detectors
- An audio-alarm
- A control panel with a display screen
- Telephone numbers to call

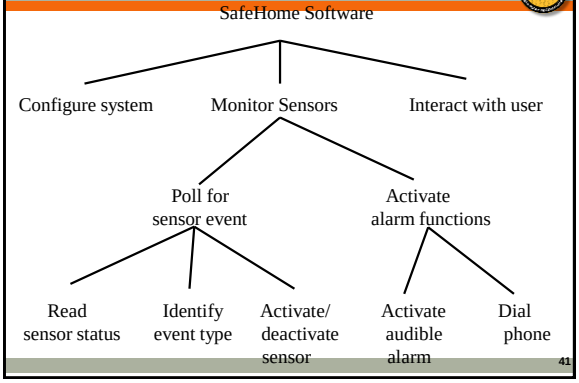
Services:
- Setting the alarm
- Monitoring the sensors
- Dialing the phone
- Programming the control panel
- Reading the display

Performance Criterias:
- A sensor event should be recognized within one second
- An event priority scheme should be implemeted

Constraints:
- Must be user friendly
- Must interface directly to a standard phone line

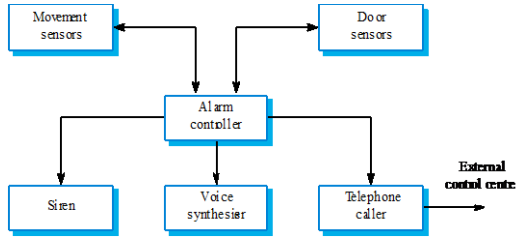
40

SafeHome Functions



41

SafeHome "Alarm sub-system"



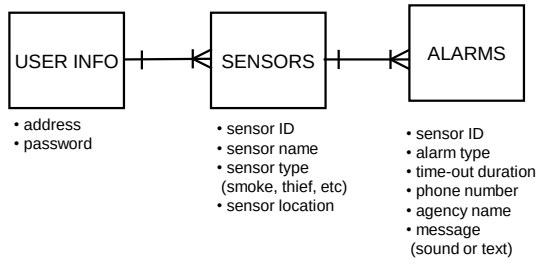
42

Alarm sub-system descriptions

Sub-system	Description
Movement sensors	Detects movement in the rooms monitored by the system
Door sensors	Detects door opening in the external doors of the building
Alarm controller	Controls the operation of the system
Siren	Emits an audible warning when an intruder is suspected
Voice synthesizer	Synthesizes a voice message giving the location of the suspected intruder
Telephone caller	Makes external calls to notify security, the police, etc.

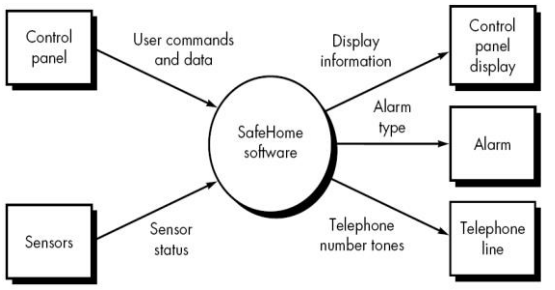
43

SafeHome Entity Relationship Diagram



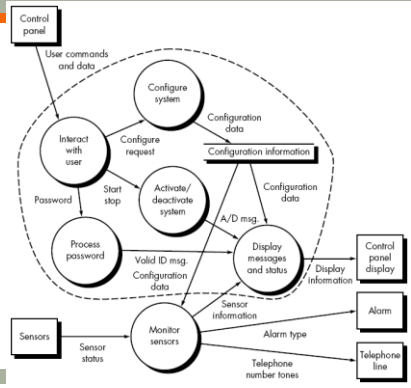
44

Level-0 DFD



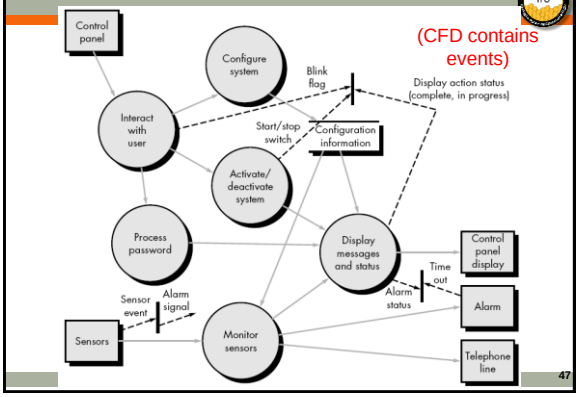
45

Level-1 DFD



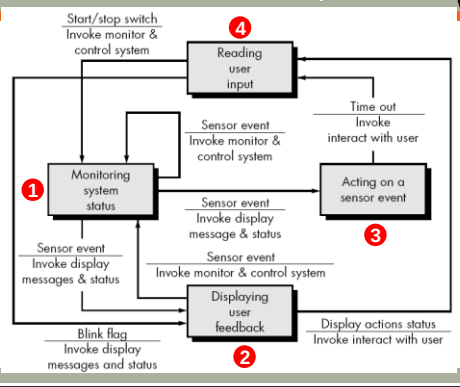
46

Level-1 CFD



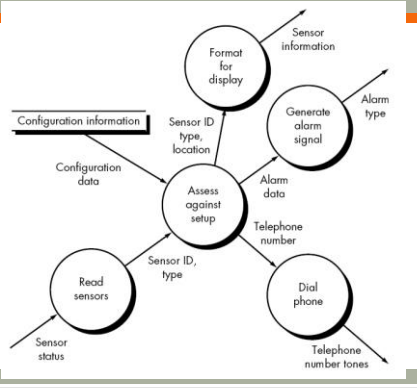
47

SafeHome State Transition Diagram



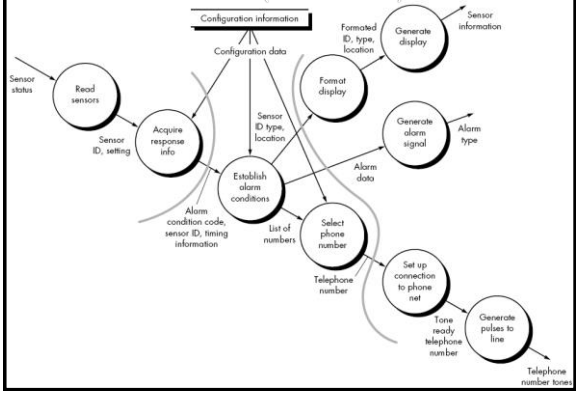
48

Level-2 DFD ("Monitor sensors")

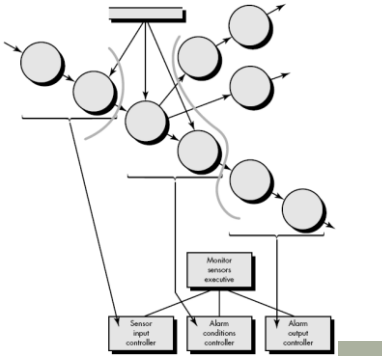


49

Level-3 DFD ("Monitor sensors")

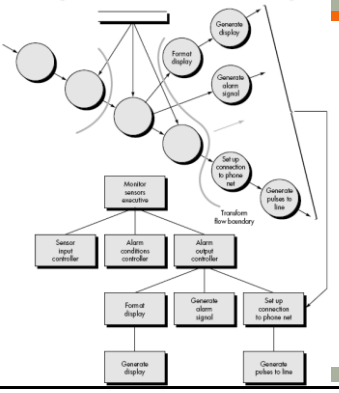


Program Structure: First factoring ("Monitor sensors")



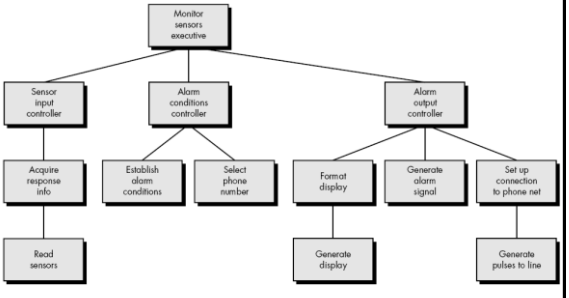
51

Program Structure: Second factoring



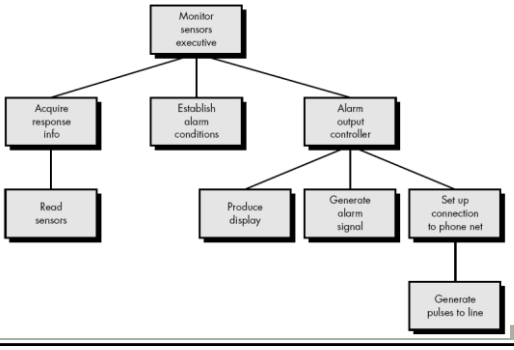
52

Program Structure: ("Monitor sensors")



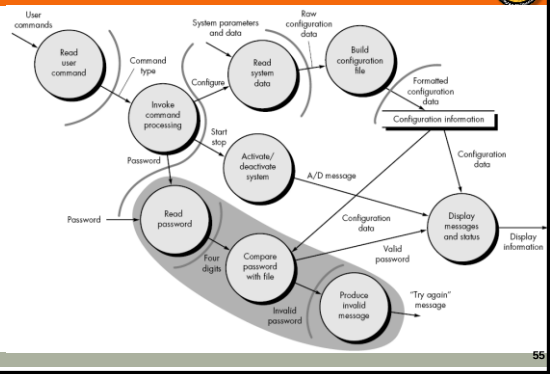
53

Refined Program Structure: ("Monitor sensors")



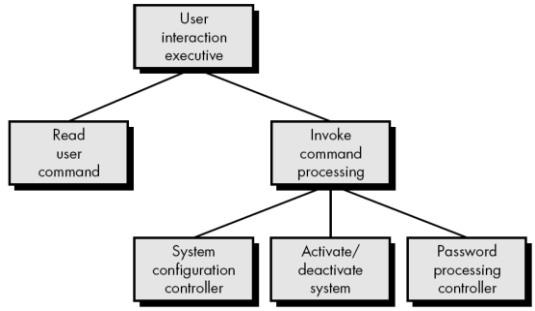
54

Level-2 DFD (user interaction subsystem)



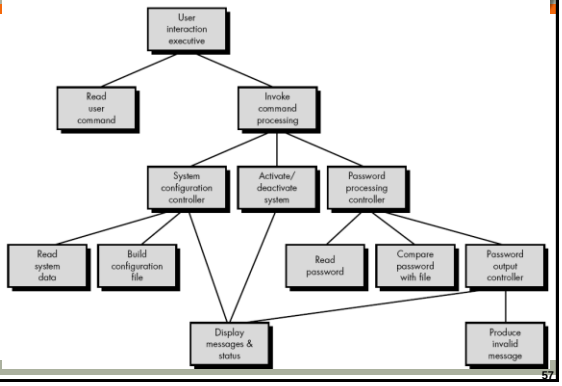
55

Program Structure: First factoring (user interaction subsystem)



56

Program Structure:(user interaction subsystem)



57

Wrap-up

This week we present

- Software Design Concepts
  - Objectives of good software design: Modularity and Reusability
- Structured and Object Oriented Design
  - When and how to apply each approach
  - How to add functions to classes and distingusih between domain and software classes
  - How to identify modules in data flow diagrams
- Design Principles
  - Coupling and cohesion at various different levels.
- User Interface Design

Next Week

- We will be covering *Good Implementation Principles and Software Testing!!!*