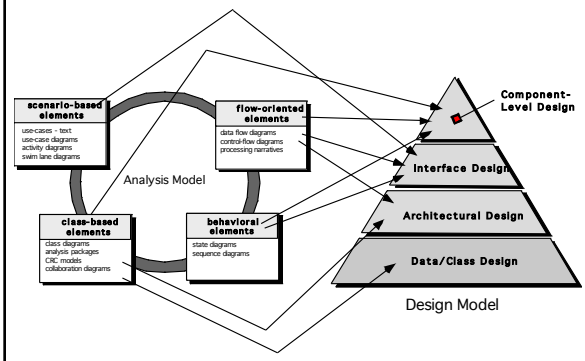
SOFTWARE ENGINEERING

Week 8
Software Design Engineering

Agenda

1. Software Design Concepts
2. Architectural design
3. Structured Design
4. Object Oriented Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

Analysis Model → Design Model



1. Software Design Concepts ⇦
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

Software Design Concepts

8.1

Layers of Software Design

- Software design is the process of applying various techniques and principles to define a system in sufficient detail to permit its physical implementation (coding).
- Design comes after analysis and it is basically based on analysis models.

Design Concepts (1)

- **Component:**
  - Any piece of software or hardware that has a clear role.
  - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
  - Many components are designed to be reusable.
  - Conversely, others perform special-purpose functions.
- **Module:**
  - A component that is defined at the programming language level.
  - For example, functions are modules in C.
  - For example, methods, classes and packages are modules in Java.

Design Concepts (2)

- **Modularity:**
  - A complex system may be divided into simpler pieces called *modules*.
  - A system that is composed of modules is called *modular*.
  - When dealing with a module we can ignore details of other modules.
  - Use divide and conquer method for modularity.
  - For two modules m1 and m2, the effort relation is as follows:

    E(m1) + E(m2) < E(m1+m2)

7

Design Concepts (3)

- **Abstraction:**
  - Abstraction is a means of achieving stepwise refinement by suppressing unnecessary details.
  - It is to conceptualize problem at a higher level.
  - Abstractions allow you to understand and concentrate on the essence of a subsystem without having to know unnecessary details.
  - The designer should keep the level of abstraction as high as possible.
- **Data abstraction (Abstract Data Type):**
  - A data type together with the operations performed on instantiations of that data type.
- **Procedural abstraction:**
  - The designer defines a procedure at a higher level.

8

Example: C definition without Data Abstraction

```
struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  int DTarihi_gun, DTarihi_ay, DTarihi_yil;
  int IGTarihi_gun, IGTarihi_ay, IGTarihi_yil;
} ;
```

9

Example: C definition with Data Abstraction

```
typedef struct
{
  int gun, ay, yil;
} tarih;

struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  tarih DogumTarihi;      //abstraction
  tarih IseGirisTarihi;   //abstraction
} ;
```

10

Example:C program without Procedural Abstraction

```
#include <stdio.h>
#include <stdlib.h>
#define N 5
int main()
{
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  int i;
  for (i=0; i < N; i++)  printf("%d \t", a[i]);
  printf("\n");
  for (i=0; i < N; i++)  printf("%d \t", b[i]);
  return 0;
}
```

11

Example:C program with Procedural Abstraction

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

void yaz(int dizi[], int M) {
  int i;
  for (i=0; i < M; i++)
    printf("%d \t", dizi[i]);
  printf("\n");
}
int main() {
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  yaz(a, N); //abstraction
  yaz(b, N); //abstraction
  return 0;
}
```

12

Layers of Software Design

1. Architectural Design
   - Defines the relationship among major structural elements of the program.
   - The modular framework of a program can be derived from the analysis model and interaction of subsystems depicted in DFD diagrams.
   - Module Interface Design describes how software modules communicate
   - Interface implies a flow of information, therefore DFD/CFD diagrams provide the basis.
2. Modular Design
3. User Interface Design

Layers of Software Design

1. Architectural Design
2. Modular Design
   1. Data Design
      - Transforms the information domain model created during the analysis into the data structures, file structures, database structures that will be required to implement software.
      - Data objects and relationships in ERD and the detailed data content depicted in the data dictionary provide the basis.
   2. Behavioral Design
      - Transforms structural elements of the program architecture into a procedural description of software components.
      - Information obtained from PDL (Algorithms / Flowcharts) serve as basis.
3. User Interface Design

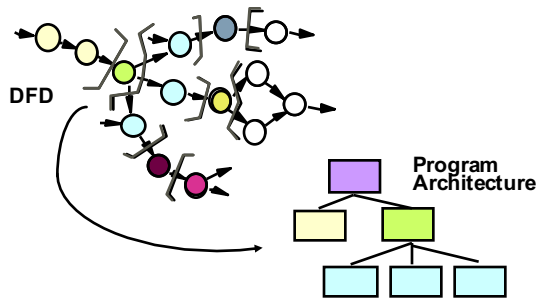
Data and Actions

- Two aspects of a product
  - Actions that operate on data
  - Data on which actions operate
- The two basic ways of designing a product
  - Operation-oriented design
  - Data-oriented design
- Third way
  - Hybrid methods
  - For example, object-oriented design

Architectural Design

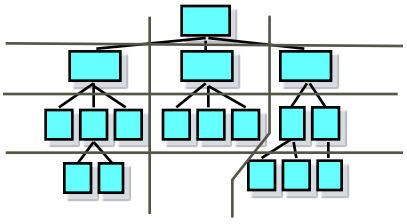
§ 7.1 §

Deriving Program Architecture



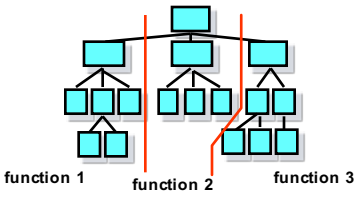
Partitioning (factoring) the Architecture

- Horizontal and Vertical partitioning are required



Vertical Partitioning

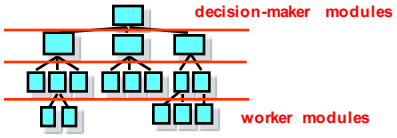
- define separate branches of the hierarchy for each major function group
- use control modules to coordinate communication between functions



19

Horizontal Partitioning

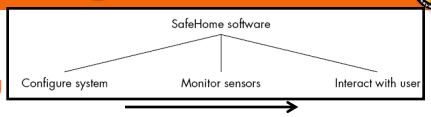
- design so that modules are layered
- decision making modules should reside at the top of the architecture



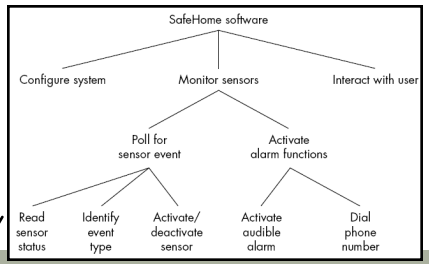
20

Example: SafeHome

Horizontal partitioning



Vertical partitioning



21

1. Software Design Concepts
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

Object Oriented Design Approach

8.3

Object-Oriented Design (OOD)

- Aim
  - Design the product in terms of the classes extracted during OOA
- If we are using a language without inheritance (e.g., C, Ada 83)
  - Use abstract data type design
- If we are using a language without a type statement (e.g., FORTRAN, COBOL)
  - Use data encapsulation

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

Object-Oriented Design Steps

- Complete the class diagram
  - The formats of the attributes can be directly deduced from the analysis artifacts
- Example: Dates
  - U.S. format (mm/mm/yyyy)
  - European format (dd/mm/yyyy)
  - In both instances, 10 characters are needed
- The formats could be added during analysis
  - To minimize rework, *never* add an item to a UML diagram until strictly necessary

Object-Oriented Design Steps

- Step 1. Complete the class diagram
  - Assign each method, either to a class or to a client that sends a message to an object of that class
- Principle A: Information hiding
- Principle B: If an operation is invoked by many clients of an object, assign the method to the object, not the clients
- Principle C: Responsibility-driven design

The MSG Foundation Case Study

- Complete the class diagram
- The final class diagram is shown in the next slide
  - **Date Class** is needed for C++
  - Java has built-it functions for handling dates

Final Class Diagram: MSG Foundation

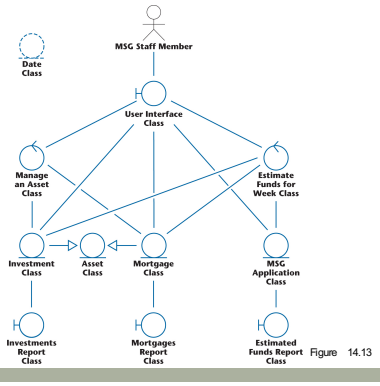


Figure 14.13

Class Diagram with Attributes: MSG Foundation

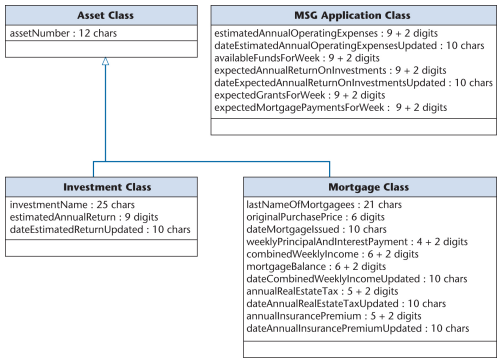


Figure 14.14

Assigning Methods to Classes: MSG Foundation

- Example: `setAssetNumber`, `getAssetNumber`
  - From the inheritance tree, these accessor/mutator methods should be assigned to **Asset Class**
  - So that they can be inherited by both subclasses of **Asset Class** (**Investment Class** and **Mortgage Class**)

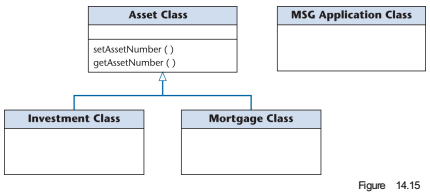


Figure 14.15

Detailed Design: MSG Foundation

- Determine what each method does
- Represent the detailed design in an appropriate format
  - PDL (pseudocode) here

Method EstimatedFundsForWeek::computeEstimatedFunds

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
   float expectedWeeklyInvestmentReturn;          (expected weekly investment return)
   float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                  (expected total mortgage payments
                                                  less total weekly grants)
   float estimatedFunds = (float) 0.0;            (total estimated funds for week)
Create an instance of an investment record.
   Investment inv = new Investment ( );
Create an instance of a mortgage record.
   Mortgage mort = new Mortgage ( );
Invoke method totalWeeklyReturnOnInvestment.
   expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );
Invoke method expectedTotalWeeklyNetPayments   (see Figure 14.17)
   expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );
Now compute the estimated funds for the week.
   estimatedFunds = (expectedWeeklyInvestmentReturn
         – (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
         + expectedTotalWeeklyNetPayments);

Store this value in the appropriate location.
   MSGApplication.setEstimatedFundsForWeek (estimatedFunds);
} // computeEstimatedFunds
```

Figure 14.16

Method Mortgage::totalWeeklyNetPayments

```
public float totalWeeklyNetPayments ( )
This method computes the net total weekly payments made by the mortgagors, that is, the expected total weekly
mortgage amount less the expected total weekly grants.
{
   File mortgageFile = new File ("mortgage.dat");          (file of mortgage records)
   float expectedTotalWeeklyMortgages = (float) 0.0;       (expected total weekly mortgage payments)
   float expectedTotalWeeklyGrants = (float) 0.0;          (expected total weekly grants)
   float interestPayment;                                  (interest payment)
   float escrowPayment;                                    (escrow payment)
   float capitalRepayment;                                 (capital repayment)
   float weeklyPayment;                                    (mortgage payment for week)
   float maximumPermittedMortgagePayment;                  (maximum amount the couple may pay)
Open the file of mortgages, name it inFile, and read each element in turn.
   read (inFile);
Compute the interest payment, escrow payment, and capital repayment for this mortgage.
   interestPayment = mortgageBalance * INTEREST_RATE / WEEKS_IN_YEAR ;
   escrowPayment = (annualPropertyTax + annualInsurancePremium) / WEEKS_IN_YEAR;
   capitalRepayment = weeklyPrincipalAndInterestPayment – interestPayment;
   mortgageBalance -= capitalRepayment;
First assume that the couple can pay the mortgage in full, without a grant.
   weeklyPayment = weeklyPrincipalAndInterestPayment + escrowPayment;
Add the weekly Principal and Interest payment to the running total of mortgage payments
   expectedTotalWeeklyMortgages += weeklyPrincipalAndInterestPayment;
Now determine how much the couple can actually pay.
   maximumPermittedMortgagePayment = currentWeeklyIncome *
         MAXIMUM_PERC_OF_INCOME;
If a grant is needed, add the grant amount to the running total of grants
   if (weeklyPayment > maximumPermittedMortgagePayment)
   expectedTotalWeeklyGrants += weeklyPayment – maximumPermittedMortgagePayment;
   }
Close the file of mortgages. Return the total expected net payments for the week.
   return (expectedTotalWeeklyMortgages – expectedTotalWeeklyGrants);
} // totalWeeklyNetPayments
```

Figure 14.17

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

Object-Oriented Design: The Elevator Problem Case Study

- Step 1. Complete the class diagram
- Consider the second iteration of the CRC card for the elevator controller

OOD: Elevator Problem Case Study

- 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)

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**

Detailed Class Diagram: Elevator Problem

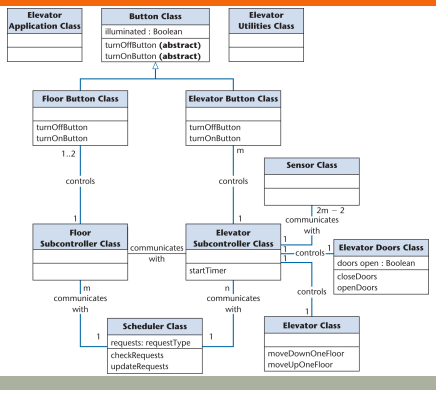


Figure 14.11

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

1. Software Design Concepts
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

Structured Design Approach

§ 8.2 ☙

Data-Oriented Design

- Basic principle
  - The structure of a product must conform to the structure of its data
- Three very similar methods
  - Michael Jackson [1975], Warnier [1976], Orr [1981]
- Data-oriented design
  - Has never been as popular as action-oriented design
  - With the rise of OOD, data-oriented design has largely fallen out of fashion

Operation-Oriented Design

- Data flow analysis
  - Use it with most specification methods (Structured Systems Analysis here)
- Key point: We have detailed action information from the DFD

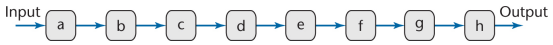


Figure 14.1

Data Flow Analysis

- Every product transforms input into output
- Determine
  - "Point of highest abstraction of input"
  - "Point of highest abstract of output"

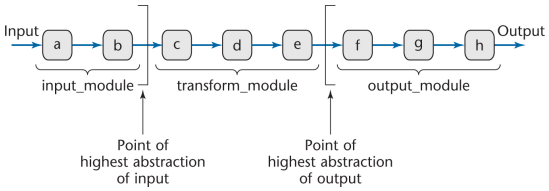


Figure 14.2

Data Flow Analysis (contd)

- Decompose the product into three modules
- Repeat stepwise until each module has high cohesion
  - Minor modifications may be needed to lower the coupling

Mini Case Study: Word Counting

- Example:

  Design a product which takes as input a file name, and returns the number of words in that file (like UNIX *wc*)

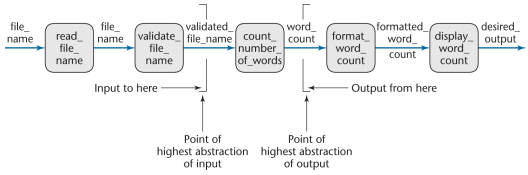
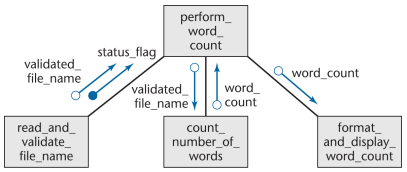


Figure 14.3

Mini Case Study: Word Counting

- First refinement



- Now

cohesion

Mini Case Study: Word Counting

- Second refinement

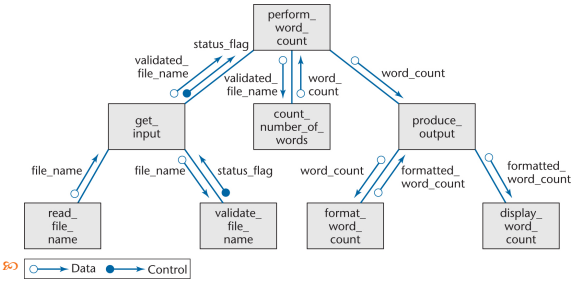


Figure 14.5

Word Counting: Detailed Design

- The architectural design is complete
  - So proceed to the detailed design
- Two formats for representing the detailed design:
  - Tabular
  - Pseudocode (PDL — program design language)

Detailed Design: Tabular Format

Module name	**count_number_of_words**
Module type	Function
Return type	**integer**
Input arguments	**validated_file_name : string**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	This module determines whether **validated_file_name** is a text file, that is, divided into lines of characters. If so, the module returns the number of words in the text file; otherwise, the module returns **−1**.

Figure 14.6(c)

Detailed Design: PDL Format

```
void perform_word_count ( )
{
   String              validated_file_name;
   Int                 word_count;

   if (get_input (validated_file_name) is null)
      print "error 1: file does not exist";
   else
   {
      set word_count equal to count_number_of_words (validated_file_name);
      if (word_count is equal to -1)
         print "error 2: file is not a text file";
      else
         produce_output (word_count);
   }
}

String get_input ( )
{
      String                  file_name;

      file_name = read_file_name ( );
      if (validate_file_name (file_name) is true)
      {
         return file_name;
      }
      else
         return null;
}

void display_word_count (String formatted_word_count)
{
      print formatted_word_count, left justified;
}

String format_word_count (int word_count);
{
      return "File contains" word_count "words";
}
```

Figure 14.7

Data Flow Analysis Extensions

- In real-world products, there is
  - More than one input stream, and
  - More than one output stream

Data Flow Analysis Extensions

- Find the point of highest abstraction for each stream

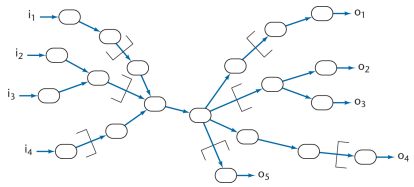


Figure 14.8

- Continue until each module has high cohesion
  - Adjust the coupling if needed

Transaction Analysis

- DFA is poor for transaction processing products
  - Example: ATM (automated teller machine)

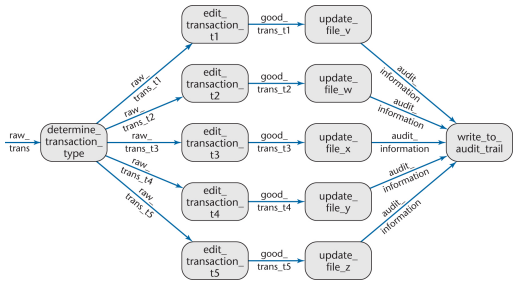


Figure 14.9

Corrected Design Using Transaction Analysis

- Software reuse
- Have one generic `edit` module, one generic `update` module
- Instantiate them 5 times

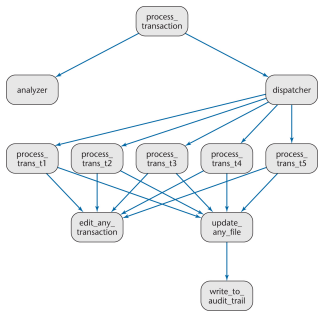


Figure 14.10

1. Software Design Concepts
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

Design Principles

8.3

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.

58

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.

59

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)

60

1.Functional Cohesion

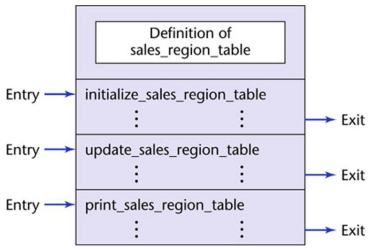
- A module with functional cohesion performs exactly one action.
- This is achieved when *all the code that computes a particular result* is kept together - and everything else is kept out.
- Example:

  calculate_sales_commission;
- Advantages:
  - Easier to understand
  - More reusable
  - Corrective maintenance is easier due to fault isolation
  - Easier to extend

61

2.Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure.
- Example:



62

3.Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data.
- All the *actions that access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out.
- Example:

  update_record_in_database_AND_write_it_to_audit_trail;
  calculate_new_coordinates_and_send_them_to_terminal;

63

4.Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed in the product.
- Procedures that are used one after another are kept together.
  - Even if one does not necessarily provide input to the next.
- Example:

  read_part_number_AND_update_repair_record_on_master_file;
- Disadvantages
  - Actions still weakly connected, so not reusable.

64

5.Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time.
- Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out.
- Example:

  initialize_sales_district_table,
  read_first_transaction_record,
  read_first_old_master_record;

  (a.k.a. perform_initialization)

65

6.Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module.
- Example:

```
int  op_code = 7;
do_operation  (op_code,
               dummy_1,
               dummy_2,
               dummy_3);
```

- Disadvantage: The interface is difficult to understand.
- (It is not clear which of the dummy variables will be used when op_code is equal to 7.)

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

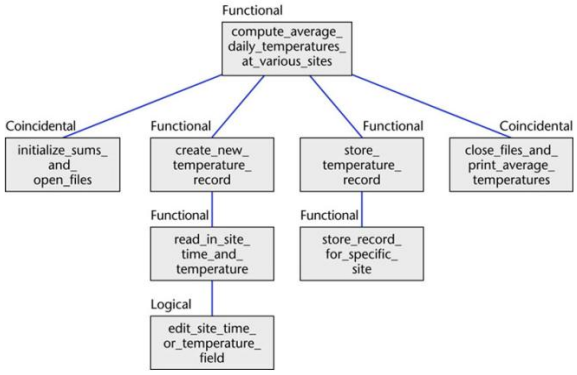
66

7.Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions.
- Example:
  print_next_line,
  reverse_string_of_second_parameter,
  add_7_to_fifth_parameter,
  convert_fourth_parameter_to_floating_point;
- Disadvantage:
  - Module not reusable.
    - Solution: Break the module into separate modules, each performing one task.
  - Degrades maintainability
  - No reuse

67

Cohesion Examples



68

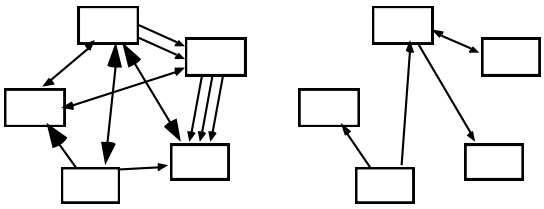
Coupling

- **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

69

Coupling



High coupling Low coupling

70

1.Data Coupling

- Two modules are data coupled if all parameters are the same data types (simple parameters or data structures)
- The more arguments a module has, the higher the coupling
  - All modules that use the called module must pass all the arguments
  - Coupling should be reduced by not defining modules with unnecessary arguments
- Example:
  compute_product (first_num, second_num);

71

2.Stamp Coupling

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure.
- Examples:
  calculate_withholding (employee_record);
  print_inventory_record (warehouse_record);
- Disadvantages:
  - It is not clear, without reading the entire module, which fields of a record are accessed or changed.
  - More data than necessary is passed

72

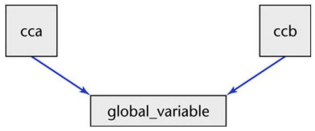
3.Control Coupling

- Two modules are control coupled if one passes an element of control to the other.
- Occurs when one procedure calls another using *a* ***'flag'*** *or* ***'command'*** that explicitly controls what the second procedure does.
- Example:
  - An operation code is passed to a module with logical cohesion
- Disadvantages:
  - The modules are not independent
  - To make a change you have to change both the calling and called modules

73

4.Common Coupling

- Two modules are common coupled if they have write access to **global data.**
- All the modules using the global variable become coupled to each other.
- Example:
  - Modules cca and ccb can access *and change* the value of global_variable
- Disadvantages:
  - A change during maintenance to the declaration of a global variable necessitates corresponding changes in all modules
  - Common-coupled modules are difficult to reuse



74

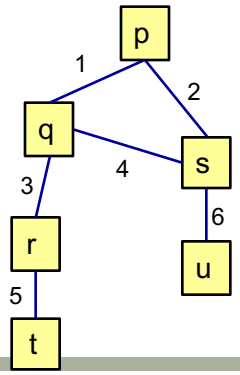
5.Content Coupling

- Two modules are content coupled if one directly references contents of the other.
- Occurs when one module modifies data that is ***internal*** to another module.
- Example:
  - Module p refers to a local data of module q and numerically changes it.
- Disadvantage: Almost any change to module q, requires a change to module p.
- To reduce content coupling you should therefore *encapsulate* all instance variables (declare variables as `private` in Java)
- Avoid call-by-reference function calling in C.

75

Coupling Examples

• Modules p, t, u access the same database in update mode.



76

Interface descriptions

Interface Number	Inputs	Outputs
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

77

Coupling between modules

	q	r	s	t	u
p	Data	—	Data or Stamp	Common	Common
q		Control	Data or Stamp	—	—
r			—	Data	—
s				—	Data
t					Common

78

1. Software Design Concepts
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome

User Interface Design

8.3

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.

80

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.

81

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).

82

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*.

83

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.

84

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

85

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

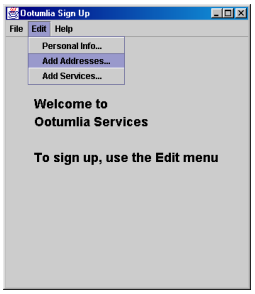
10. Provide all necessary *help*.
    - Integrate help with the application.
    - Ensure that the help is accurate.

11. Be *consistent and uniform*.
    - Use similar layouts and graphic designs throughout your application.
    - Follow look-and-feel standards.

86

Example: Welcome screen

Wrong

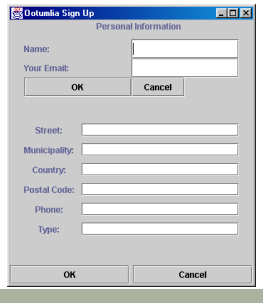


Correct

87

Example: Personal information screen

Wrong

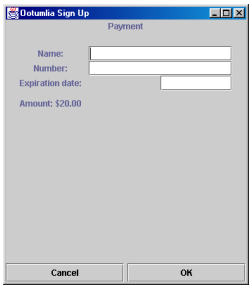


Correct

88

Example: Payment screen

Wrong



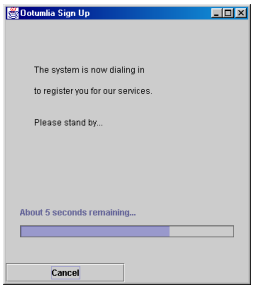
Correct

89

Example: Sign up screen

Wrong

Correct

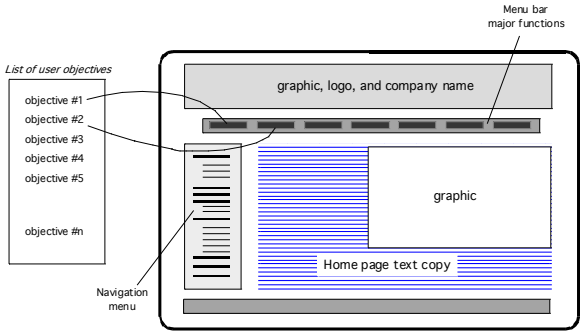


90

User Interface Patterns

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

Example: Web page layout



1. Software Design Concepts
2. Architectural design
3. Object Oriented Design
4. Structured Design
5. Design Principles
6. User Interface Design
7. Case Study: SafeHome ⇦

Case Study: SafeHome

8.7

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.

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.

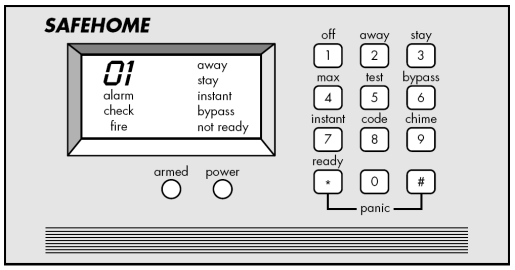
- Data objects: Underlined nouns
- *Processes: Italic verbs*

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***.

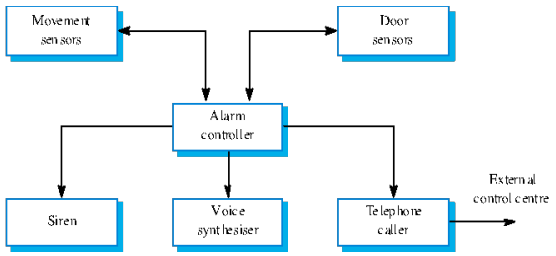
All interaction with SafeHome is ***managed*** by a user-interaction subsystem that ***reads*** input provided through the keypad and function keys, ***displays*** prompting messages and system status on the LCD display. Keyboard interactions takes the following form: (continues...)

SafeHome Control Panel



97

SafeHome "Alarm sub-system"



98

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.

99

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

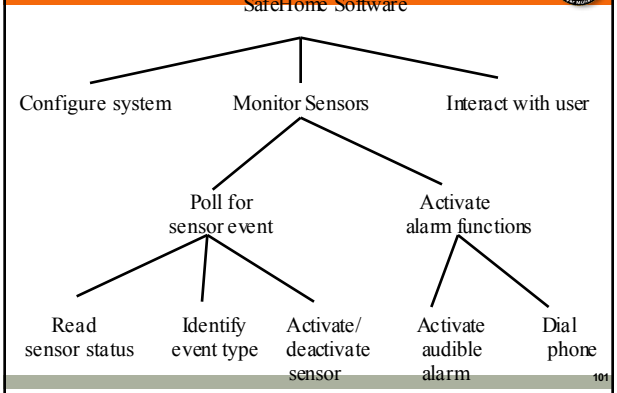
Services:
- Setting the alarm
- Monitoring the sensors
- Dialing the phone
- Programming the control panel
- Reading the display

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

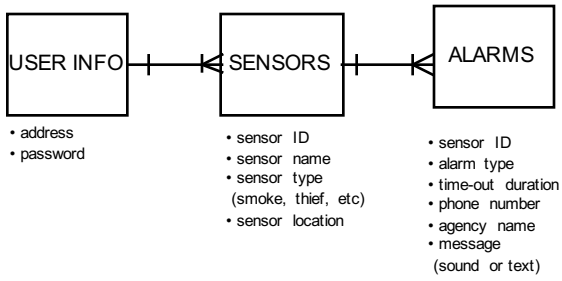
100

SafeHome Functions

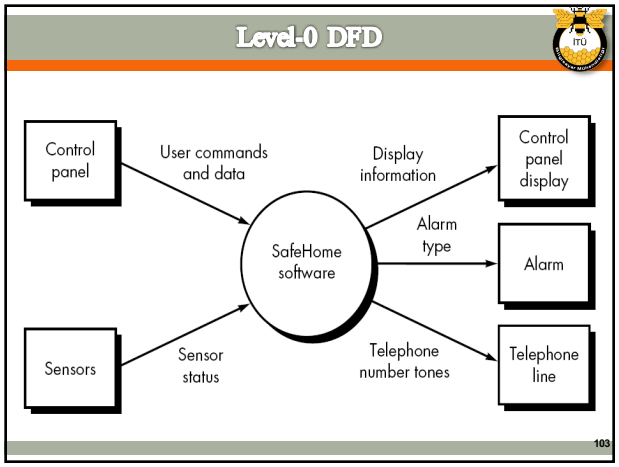


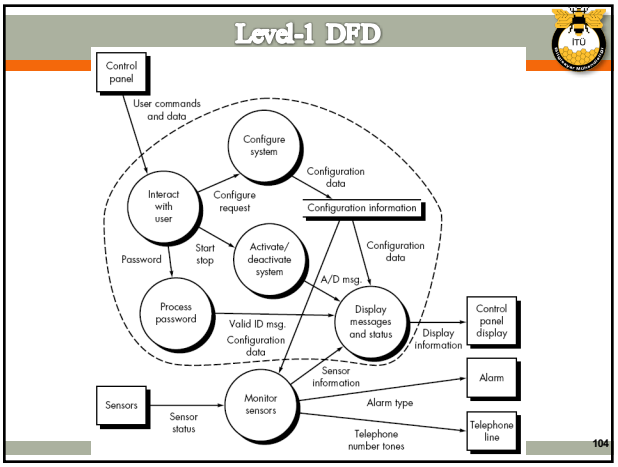
101

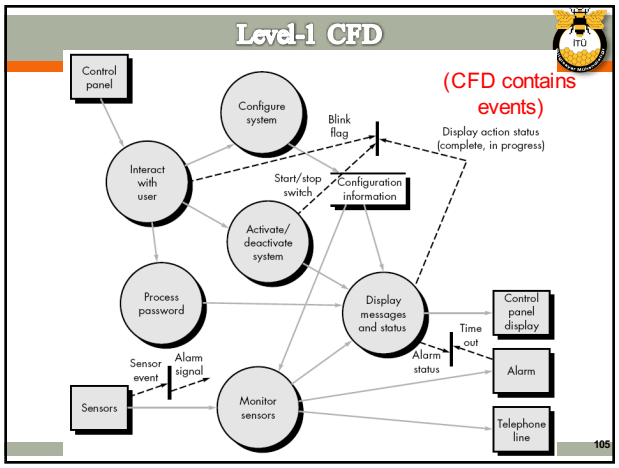
SafeHome Entity Relationship Diagram

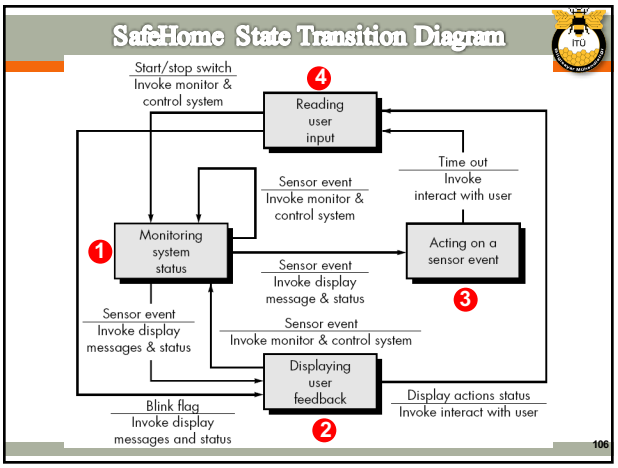


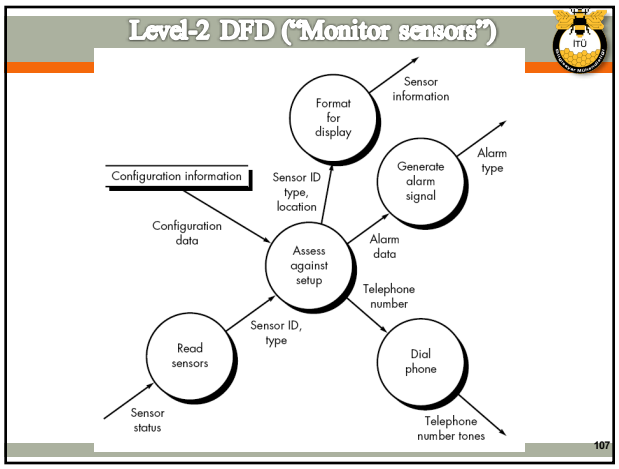
102

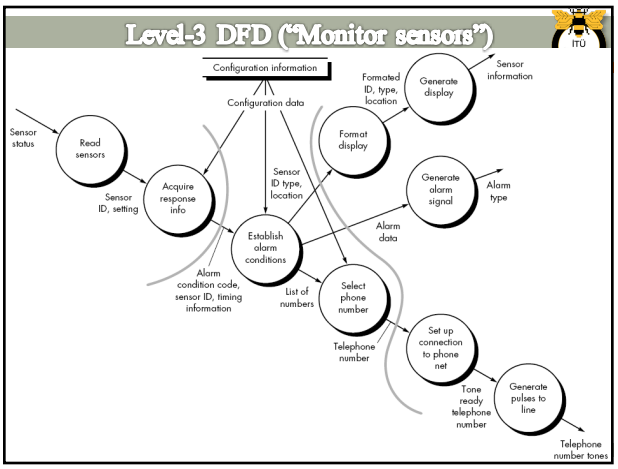

Level-0 DFD
Control panel
User commands and data
Display information
Control panel display
SafeHome software
Alarm type
Alarm
Sensors
Sensor status
Telephone number tones
Telephone line
103


Level-1 DFD
Control panel
User commands and data
Configure system
Configuration data
Interact with user
Configure request
Configuration information
Password
Start stop
Activate/ deactivate system
Configuration data
A/D msg.
Process password
Valid ID msg.
Configuration data
Display messages and status
Display information
Control panel display
Sensor information
Alarm
Sensors
Sensor status
Monitor sensors
Alarm type
Telephone number tones
Telephone line
104

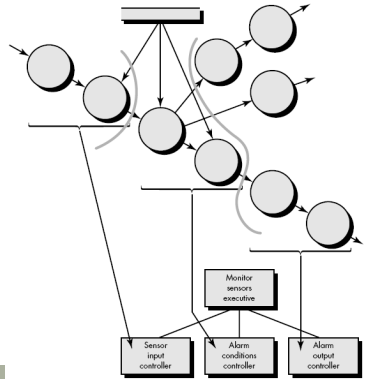

Level-1 CFD
(CFD contains events)
Control panel
Configure system
Blink flag
Display action status (complete, in progress)
Interact with user
Start/stop switch
Configuration information
Activate/ deactivate system
Process password
Display messages and status
Control panel display
Time out
Alarm status
Alarm
Sensor event
Alarm signal
Sensors
Monitor sensors
Telephone line
105


SafeHome State Transition Diagram
Start/stop switch
Invoke monitor & control system
4
Reading user input
Time out
Invoke interact with user
Sensor event
Invoke monitor & control system
1
Monitoring system status
Acting on a sensor event
3
Sensor event
Invoke display message & status
Sensor event
Invoke display messages & status
Sensor event
Invoke monitor & control system
Displaying user feedback
Display actions status
Invoke interact with user
Blink flag
Invoke display messages and status
2
106


Level-2 DFD ("Monitor sensors")
Format for display
Sensor information
Generate alarm signal
Alarm type
Configuration information
Sensor ID type, location
Configuration data
Assess against setup
Alarm data
Telephone number
Sensor ID, type
Read sensors
Dial phone
Sensor status
Telephone number tones
107

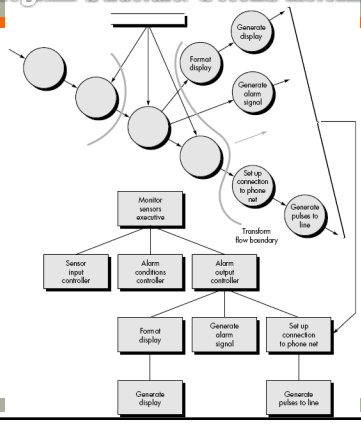

Level-3 DFD ("Monitor sensors")
Configuration information
Configuration data
Formated ID, type, location
Generate display
Sensor information
Sensor status
Read sensors
Format display
Acquire response info
Sensor ID, setting
Sensor ID type, location
Generate alarm signal
Alarm type
Establish alarm conditions
Alarm data
Alarm condition code, sensor ID, timing information
List of numbers
Select phone number
Telephone number
Set up connection to phone net
Tone ready telephone number
Generate pulses to line
Telephone number tones

Program Structure: First factoring ("Monitor sensors")



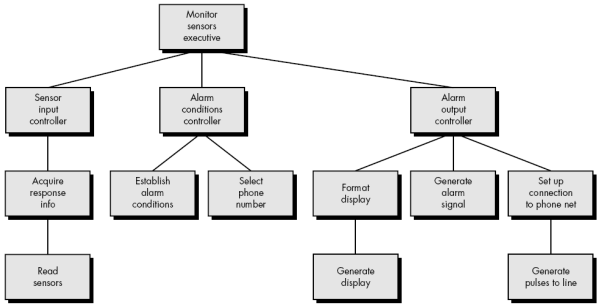
109

Program Structure: Second factoring



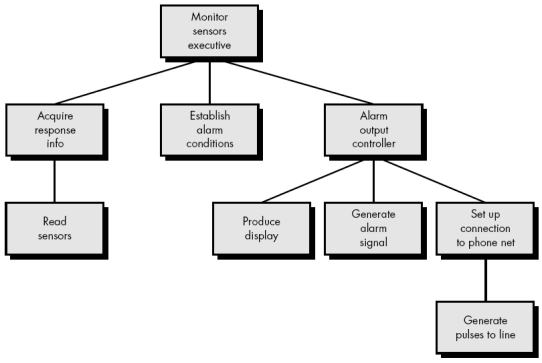
110

Program Structure: ("Monitor sensors")



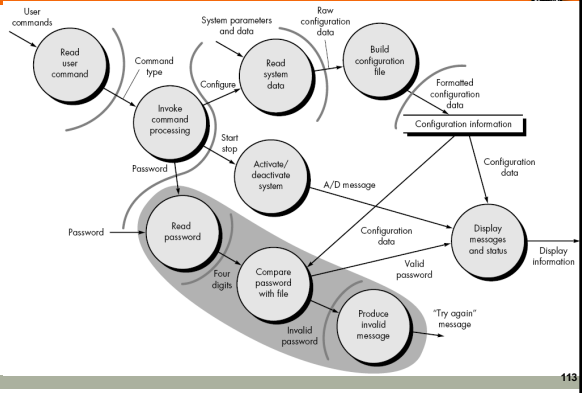
111

Refined Program Structure: ("Monitor sensors")



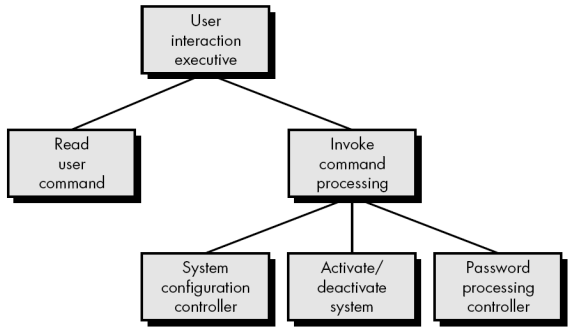
112

Level-2 DFD (user interaction subsystem)

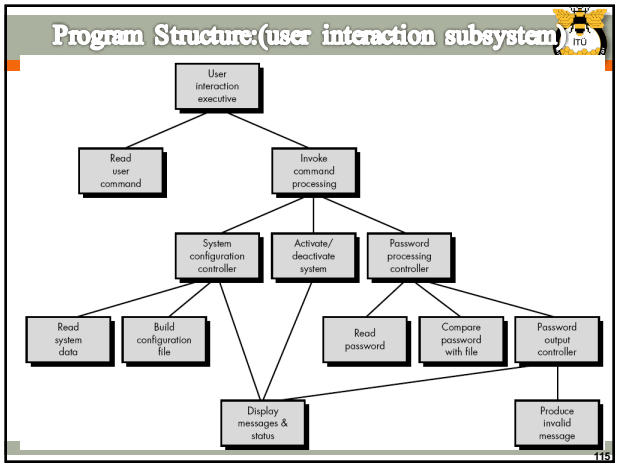


113

Program Structure: First factoring (user interaction subsystem)



114


Program Structure:(user interaction subsystem)
User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller
Read system data
Build configuration file
Read password
Compare password with file
Password output controller
Display messages & status
Produce invalid message
115