İTÜ

İSTANBUL TEKNİK ÜNİVERSİTESİ 1773

SOFTWARE ENGINEERING

Week 9

Software Design:

Object-Oriented Modeling Perspective

1. Software Design Concepts 
2. Structured Design
   2.1. Case Study: SafeHome
3. Object Oriented Design Principles
   3.1. Unified Modeling Language
   3.2. Case Study: Elevator
4. User Interface Design

Software Design Concepts

§ 8.1 §

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.

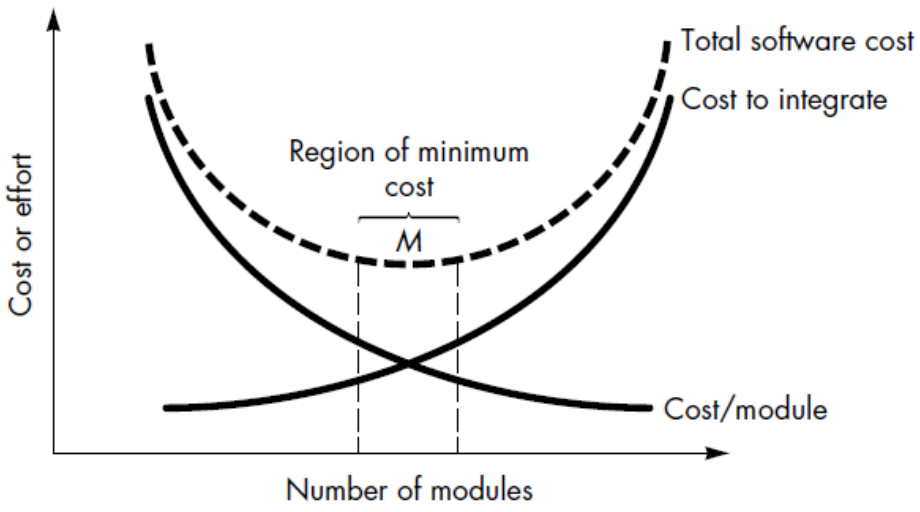
- **Module:**
  - A component that is defined at the programming language level.
  - For example, functions are modules in C.
  - For example, methods, classes and packages are modules in Java.

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:

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

FIGURE 13.2
Modularity and software cost



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.

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;
} ;
```

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
} ;
```

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;
}
```

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;
}
```

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

Design Characteristics

- 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.

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 is a 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.

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)

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

Example of content coupling

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}
```

```
public class Point
{
  private int x, y;
  ...
  public void setX(int x){ this.x = x; }
  public void setY(int y){ this.y = y; }
}
```

```
public class Arch
{
  private Line baseline;
  ...
  void move(int newX, int newY){
    Point theEnd = baseline.getEnd();
    theEnd.setX(newX);
   theEnd.setY(newY);
  }
}
```

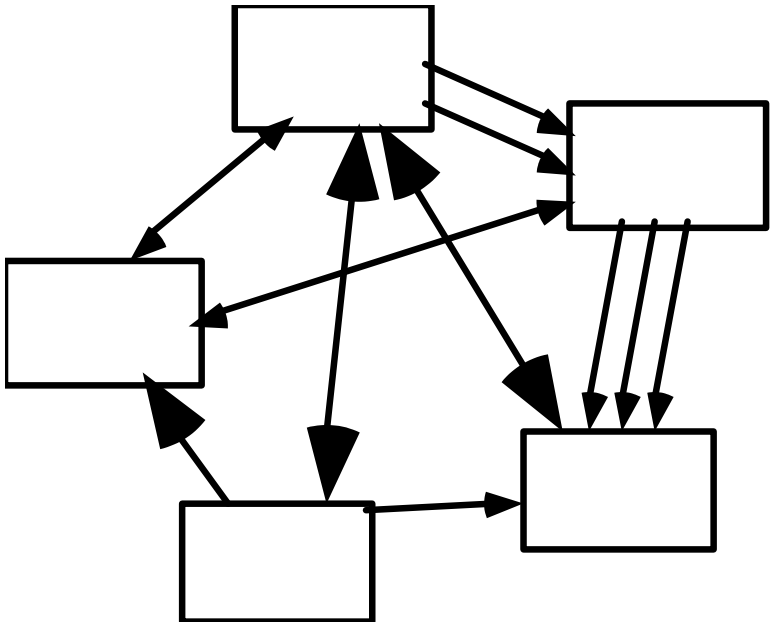
One solution (other solutions exist)

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
  public void moveEnd(int x, int y)  {
     end = new Point(x, y);
   }
}
```

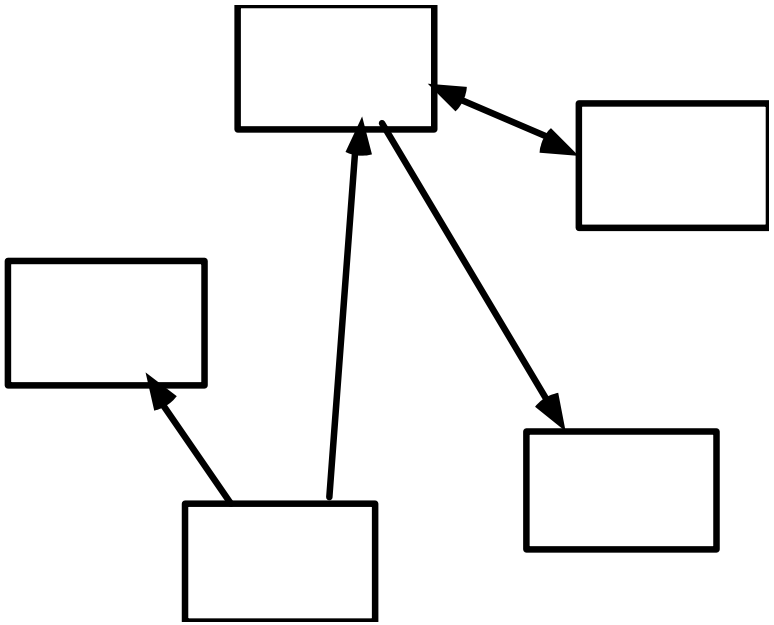
```
public class Point
{
  private int x, y;
    ...

}
```

```
public class Arch
{
  private Line baseline;
  ...
  void move(int newX, int newY){
    baseLine.moveEnd(newX, newY);
  }
}
```

Coupling



High coupling



Low coupling

1. Software Design Concepts
2. Structured Design 
   2.1. Case Study: SafeHome
3. Object Oriented Design Principles
   3.1. Unified Modeling Language
   3.2. Case Study: Elevator
4. User Interface Design

Structured Design

§ 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

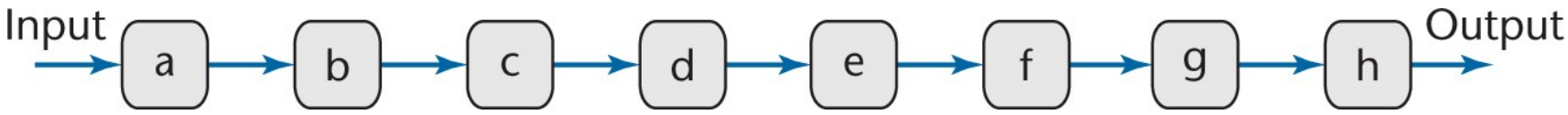


Figure 14.1

Data Flow Analysis

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

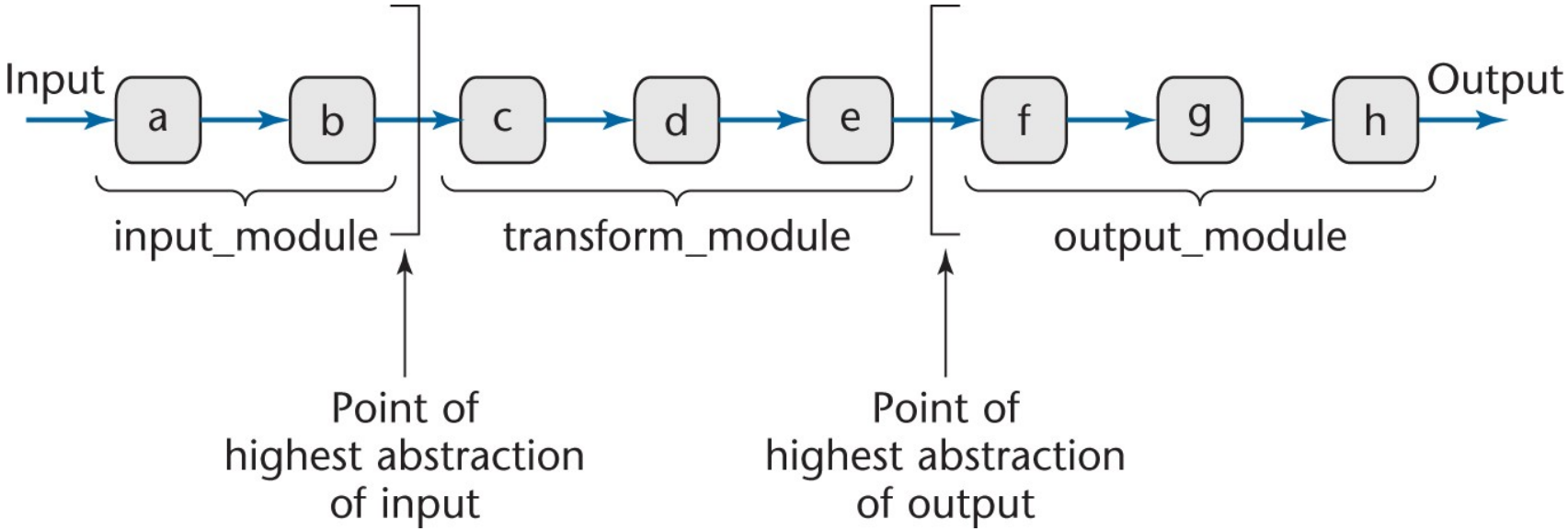


Figure 14.2

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*)

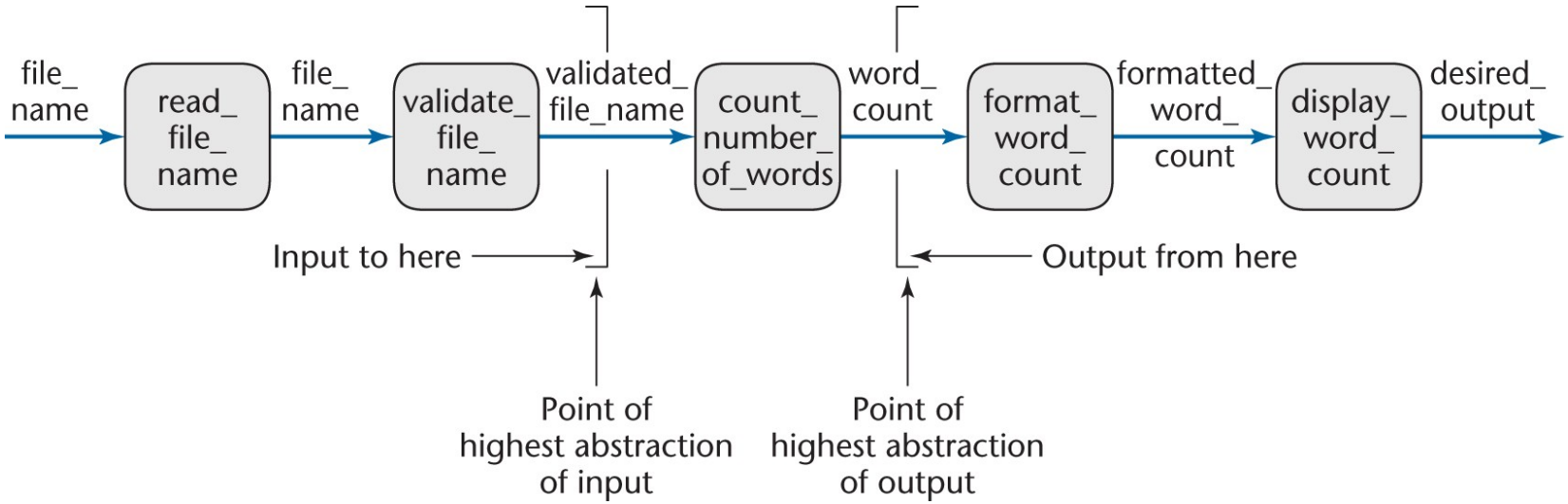


Figure 14.3

Mini Case Study: Word Counting

- First refinement

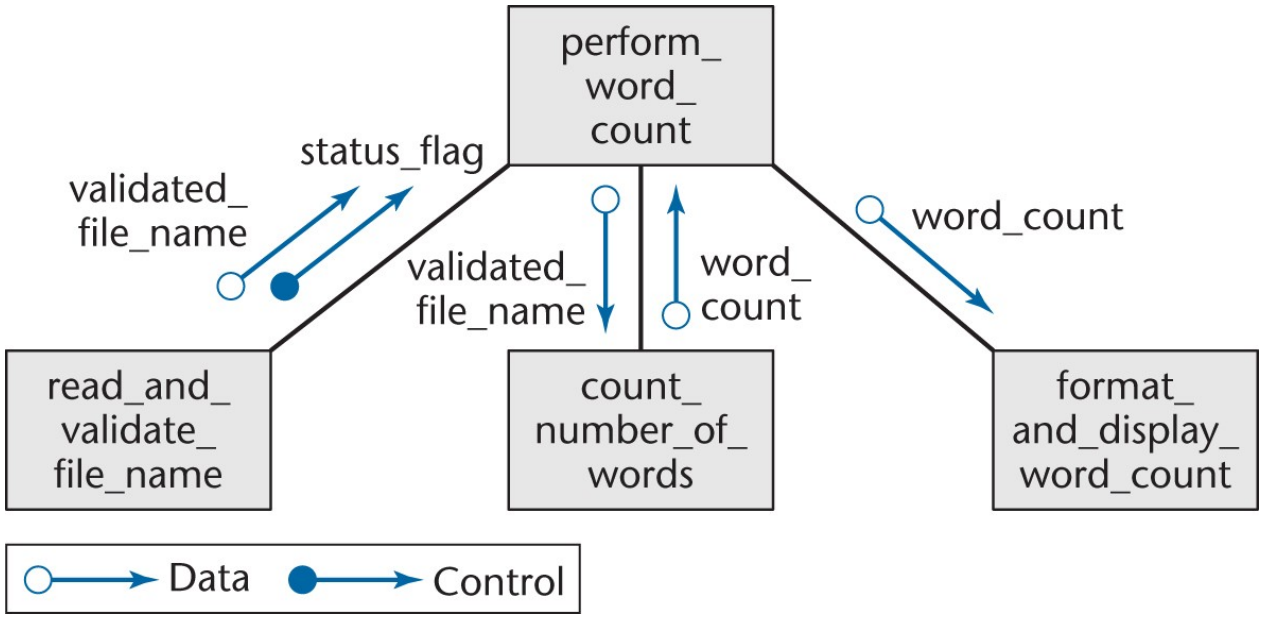


Figure 14.4

- Now refine the two modules of communicational cohesion

Mini Case Study: Word Counting

- Second refinement

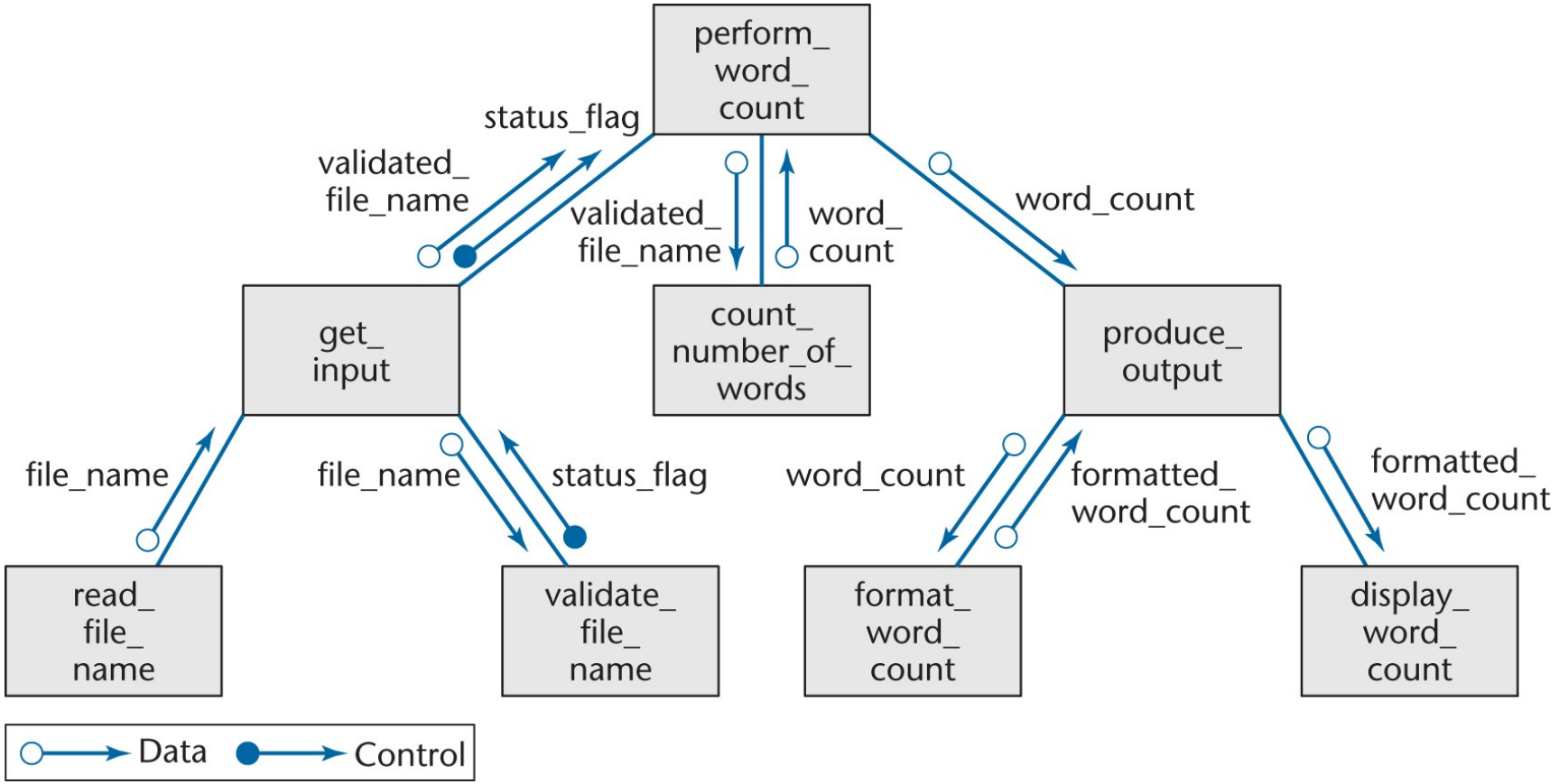


Figure 14.5

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

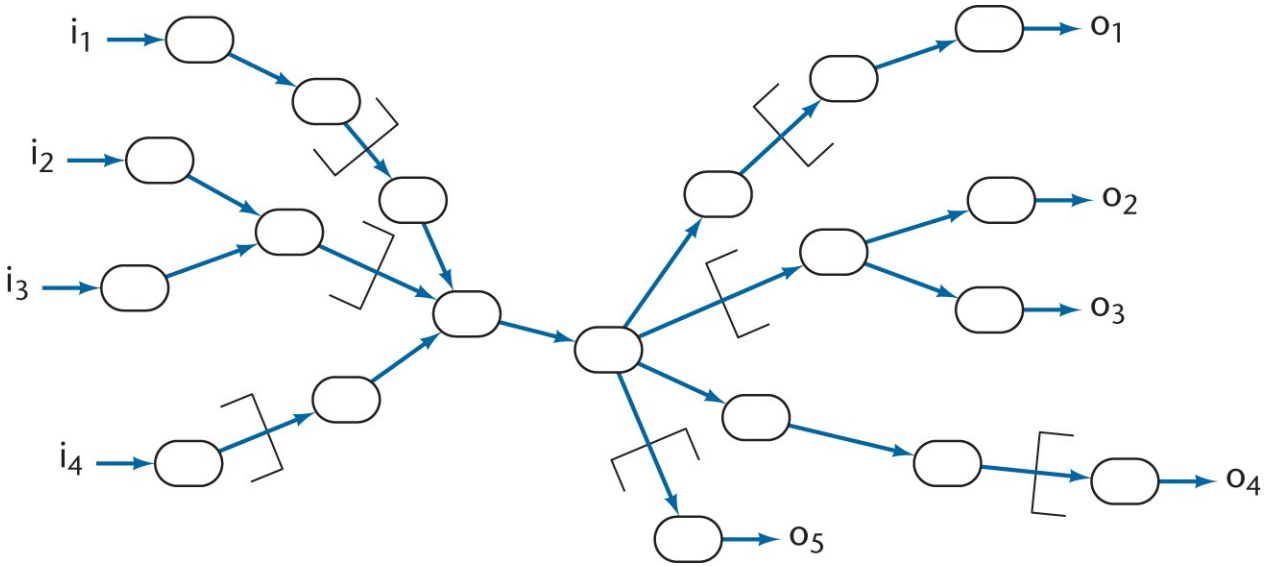


Figure 14.8

- Continue until each module Works on a single responsibility

Case Study: SafeHome

§ 8.3 §

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.

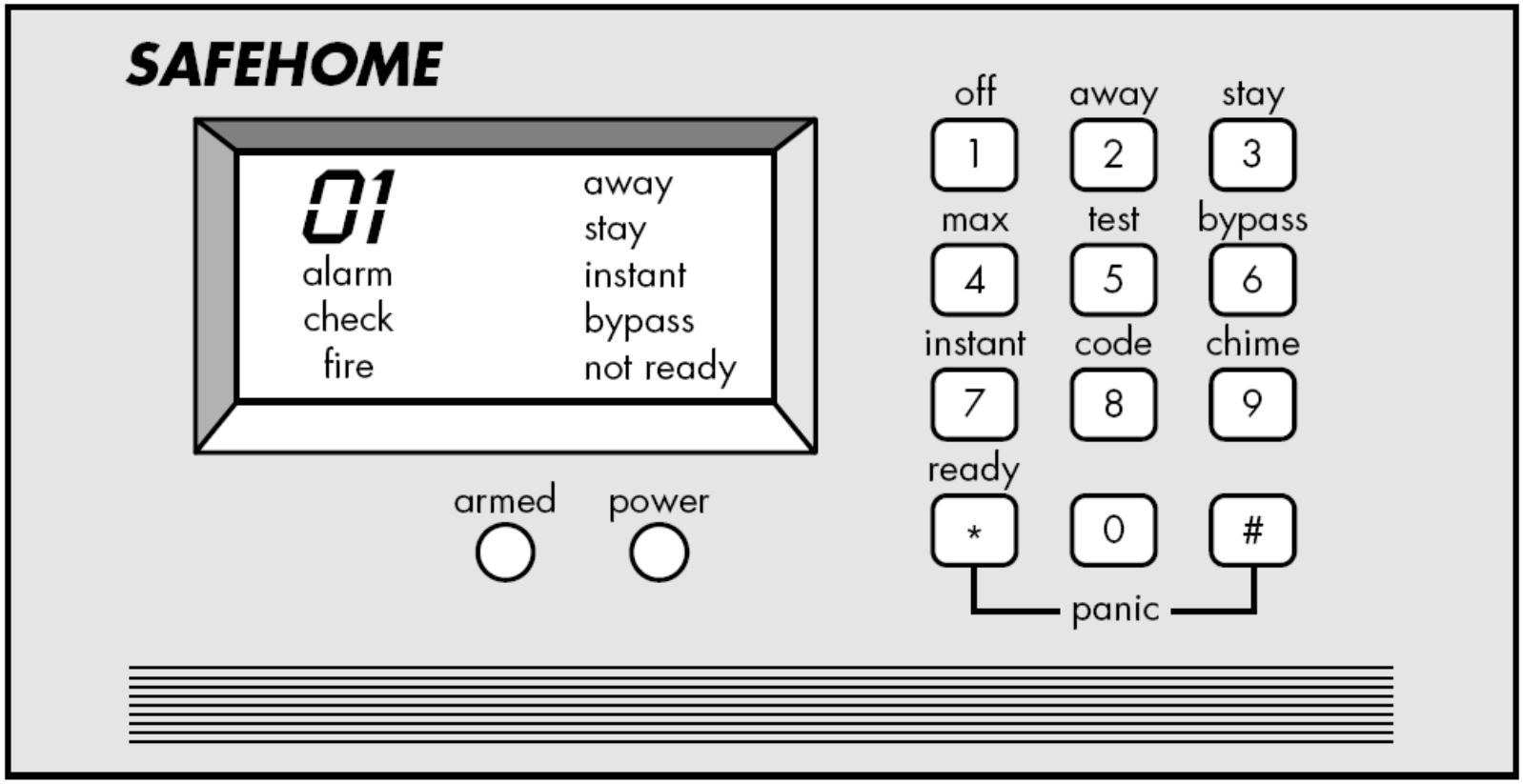
- Data objects: Underlined nouns
- *Processes: Italic verbs*

Statement of Software Scope (2)

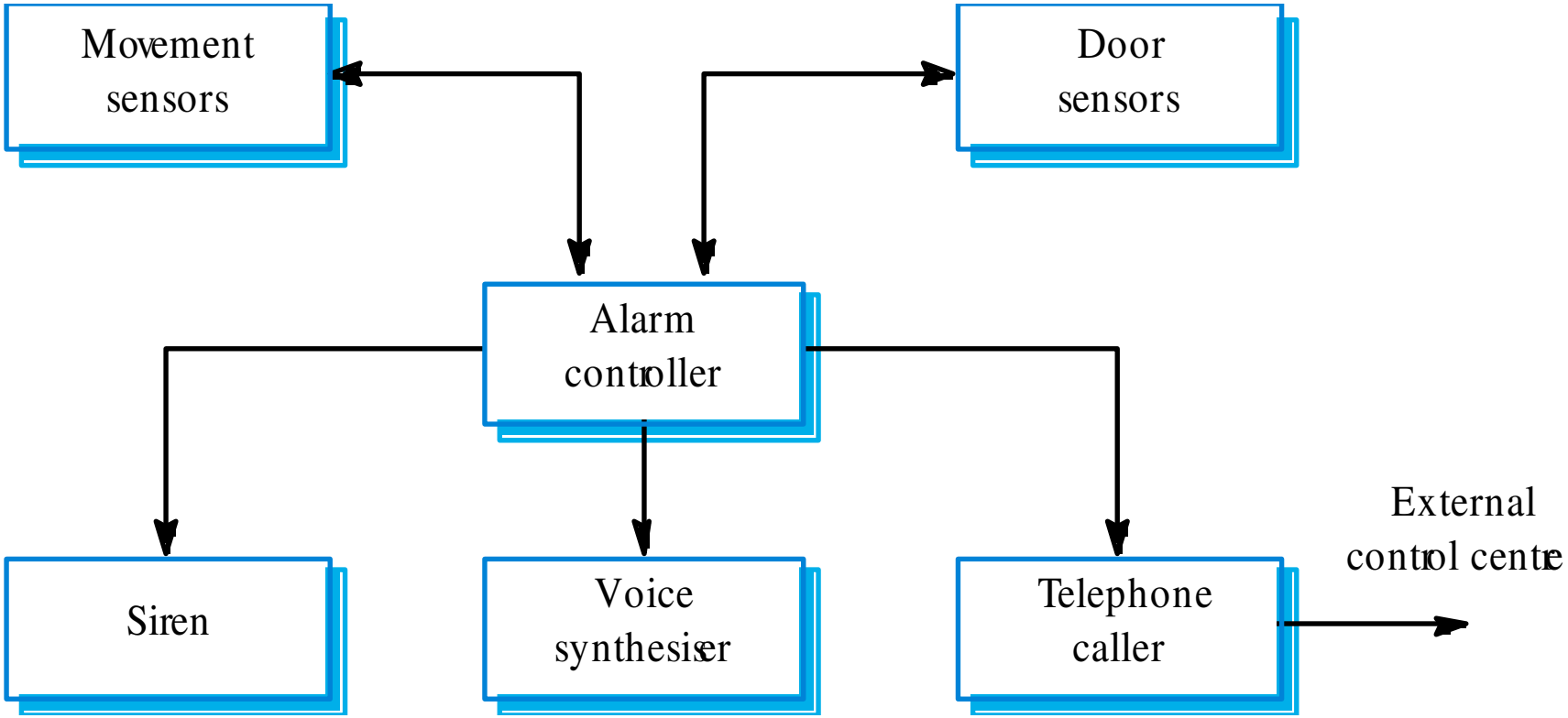
When a sensor event is ***recognized***, the software ***invokes*** an audible alarm attached to the system. After a delay time, that is ***specified*** by the homeowner during the system configuration activities, the software dials a telephone number of a monitoring service agency, ***provides*** information 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



SafeHome "Alarm sub-system"



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.

Customer Requirements

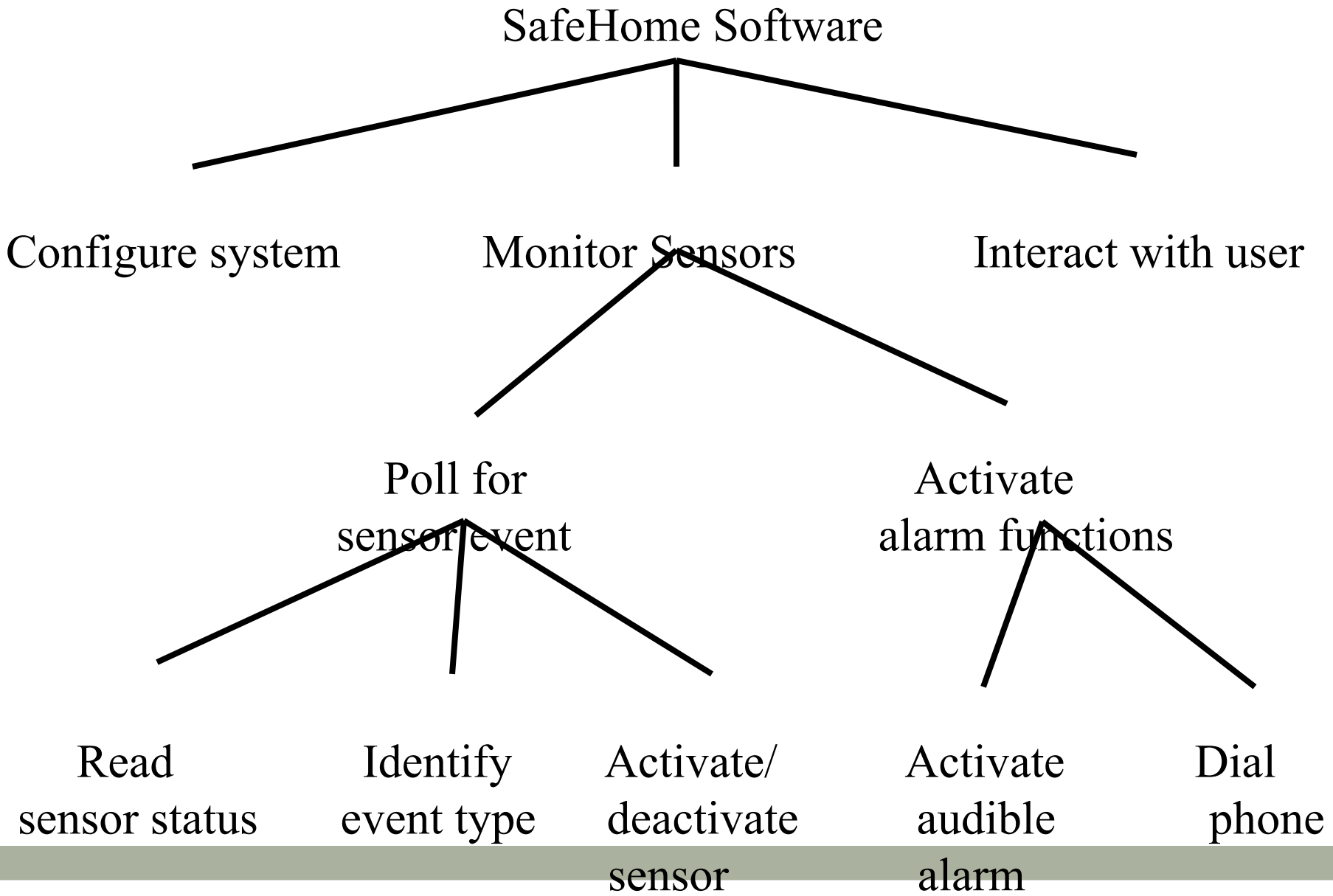
Objects:
- Smoke detectors
- Door and window sensors
- Motion detectors
- An audio-alarm
- A control panel with a display screen
- Telephone numbers to call

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

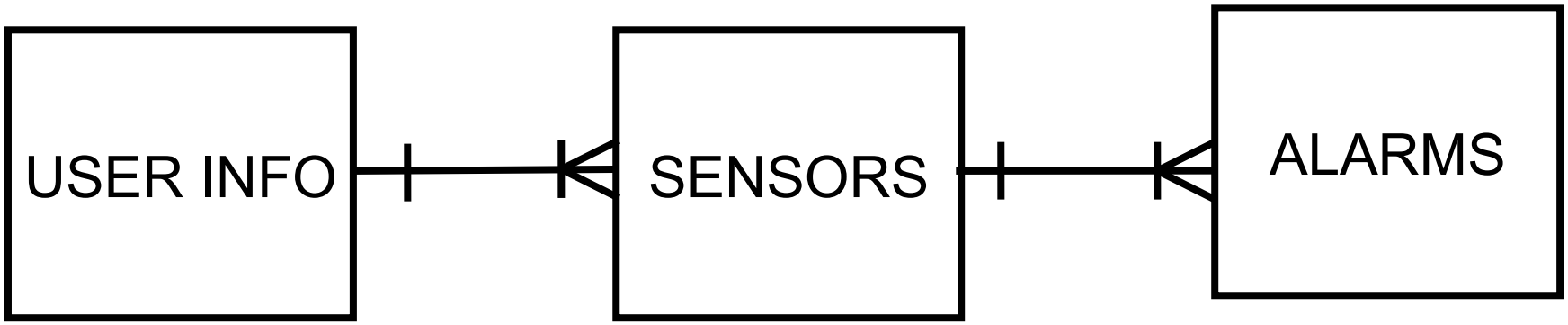
Performance Criteria:
- A sensor event should be recognized within one second
- An event priority scheme should be implemeted

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

SafeHome Functions



SafeHome Entity Relationship Diagram

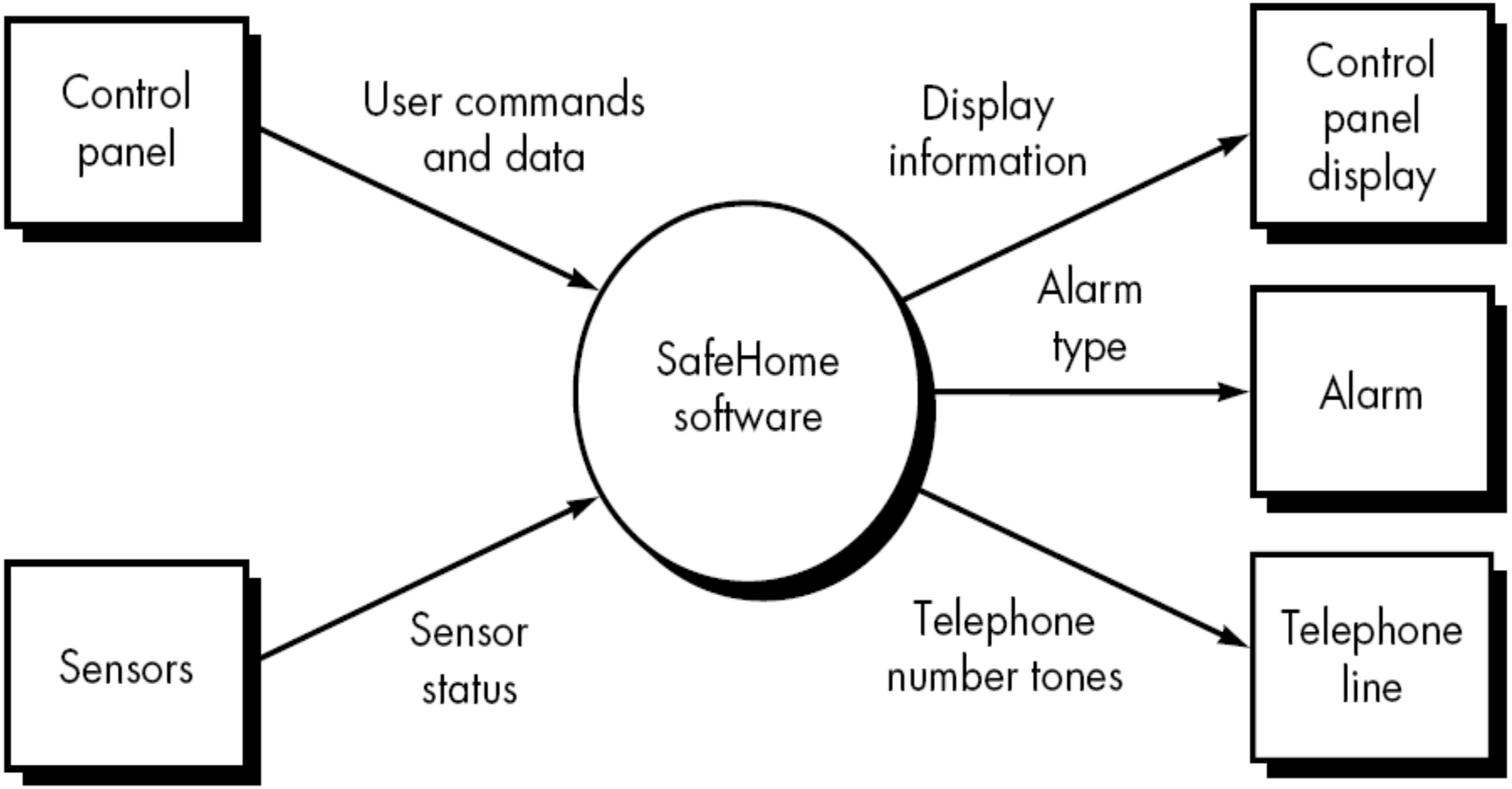


- address
- password

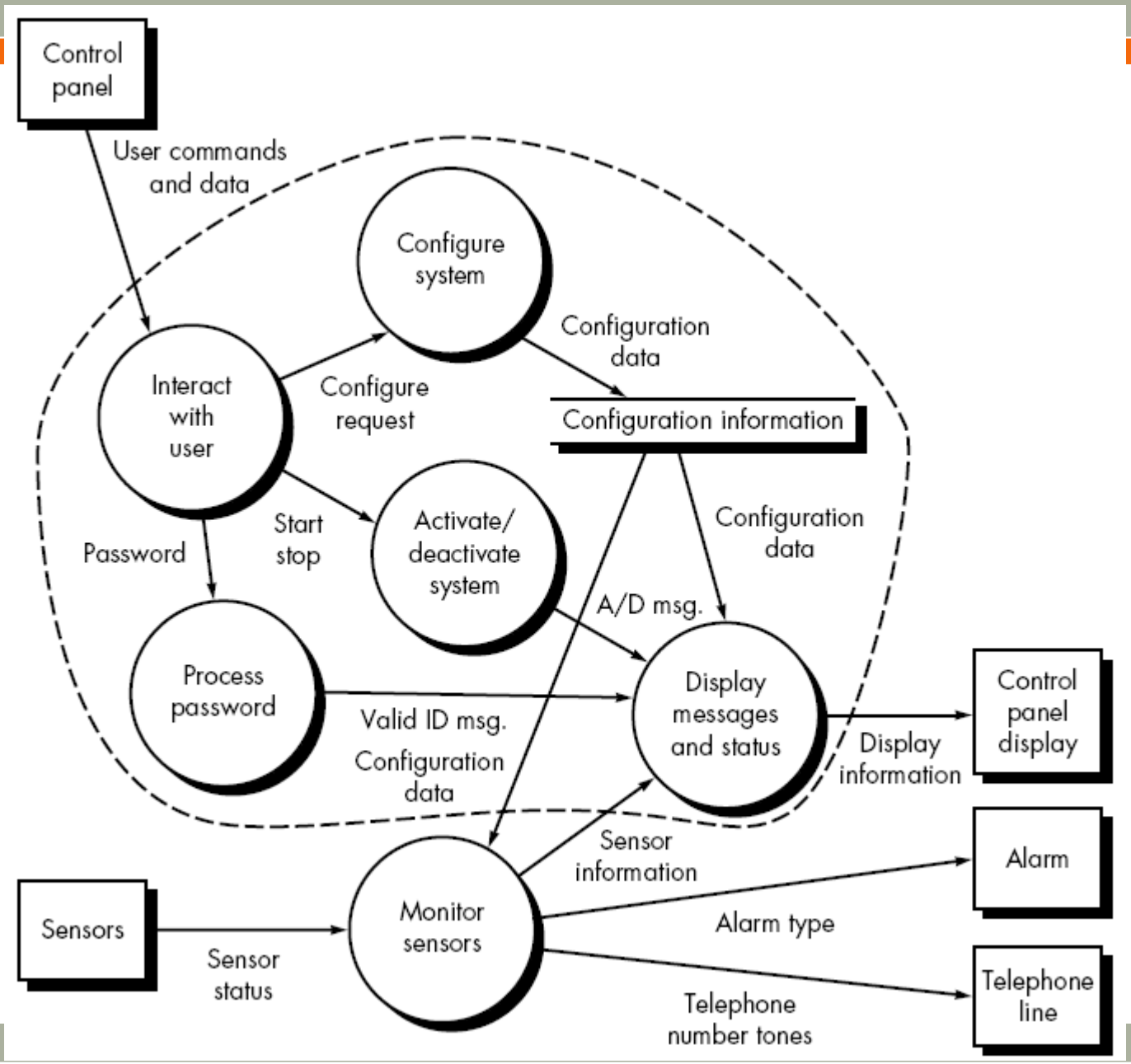
- sensor ID
- sensor name
- sensor type (smoke, thief, etc)
- sensor location

- sensor ID
- alarm type
- time-out duration
- phone number
- agency name
- message (sound or text)

Level-0 DFD

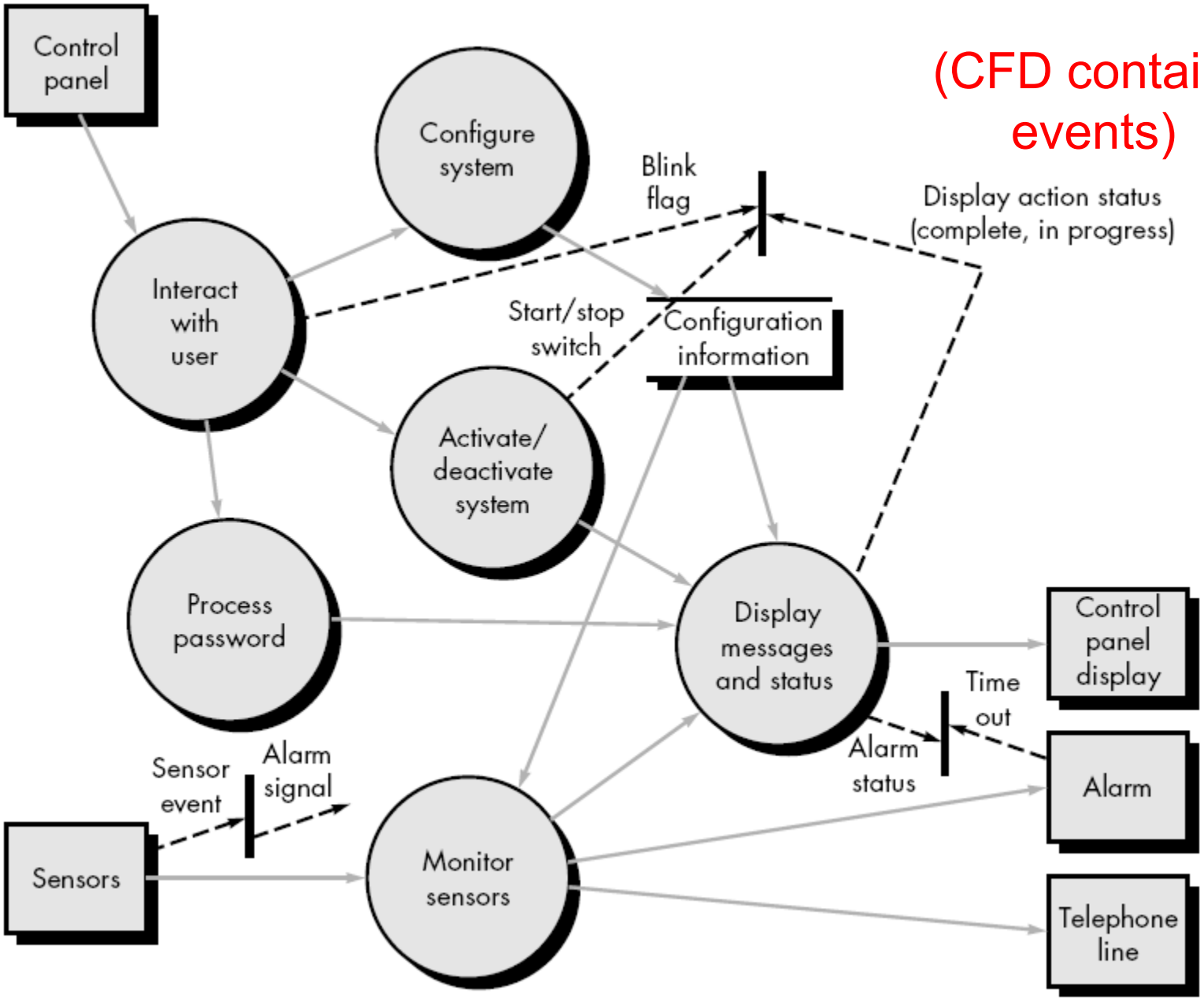

Control panel
User commands and data
Display information
Control panel display
SafeHome software
Alarm type
Alarm
Sensors
Sensor status
Telephone number tones
Telephone line

Level-1 DFD

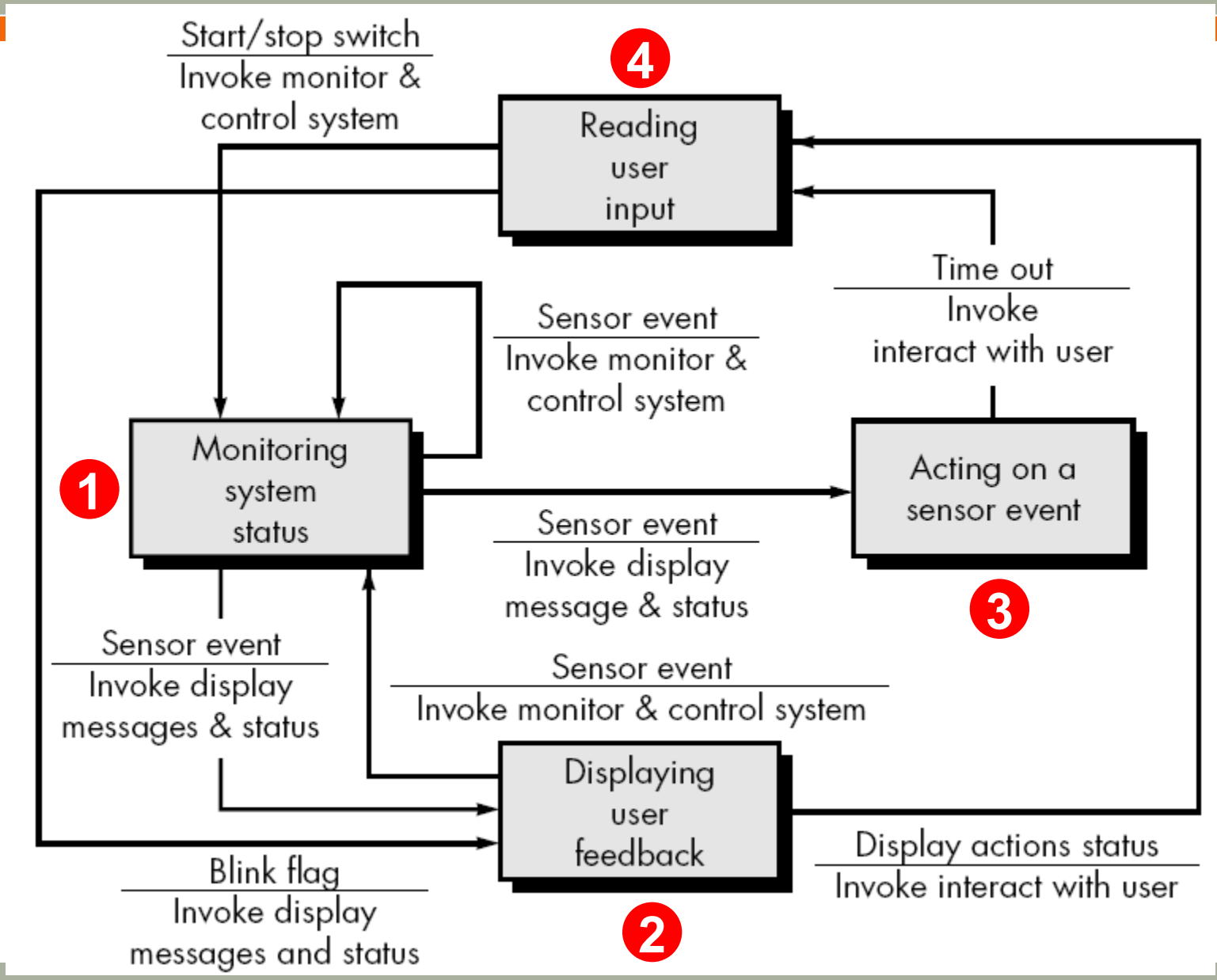

Control panel
User commands and data
Configure system
Configuration data
Interact with user
Configure request
Configuration information
Start stop
Activate/ deactivate system
Configuration data
Password
A/D msg.
Process password
Display messages and status
Control panel display
Valid ID msg.
Configuration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

Level-1 CFD

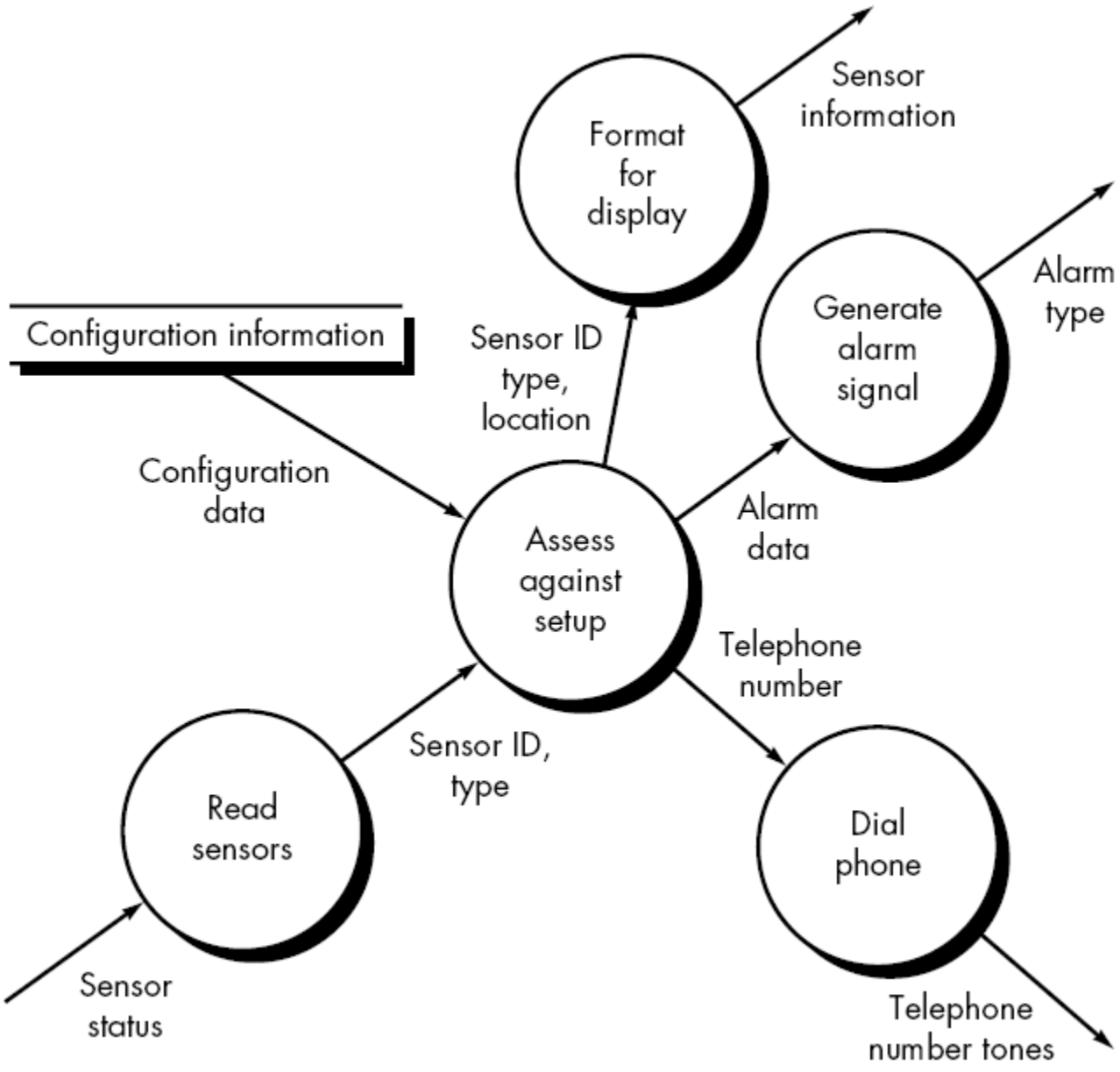
(CFD contains events)



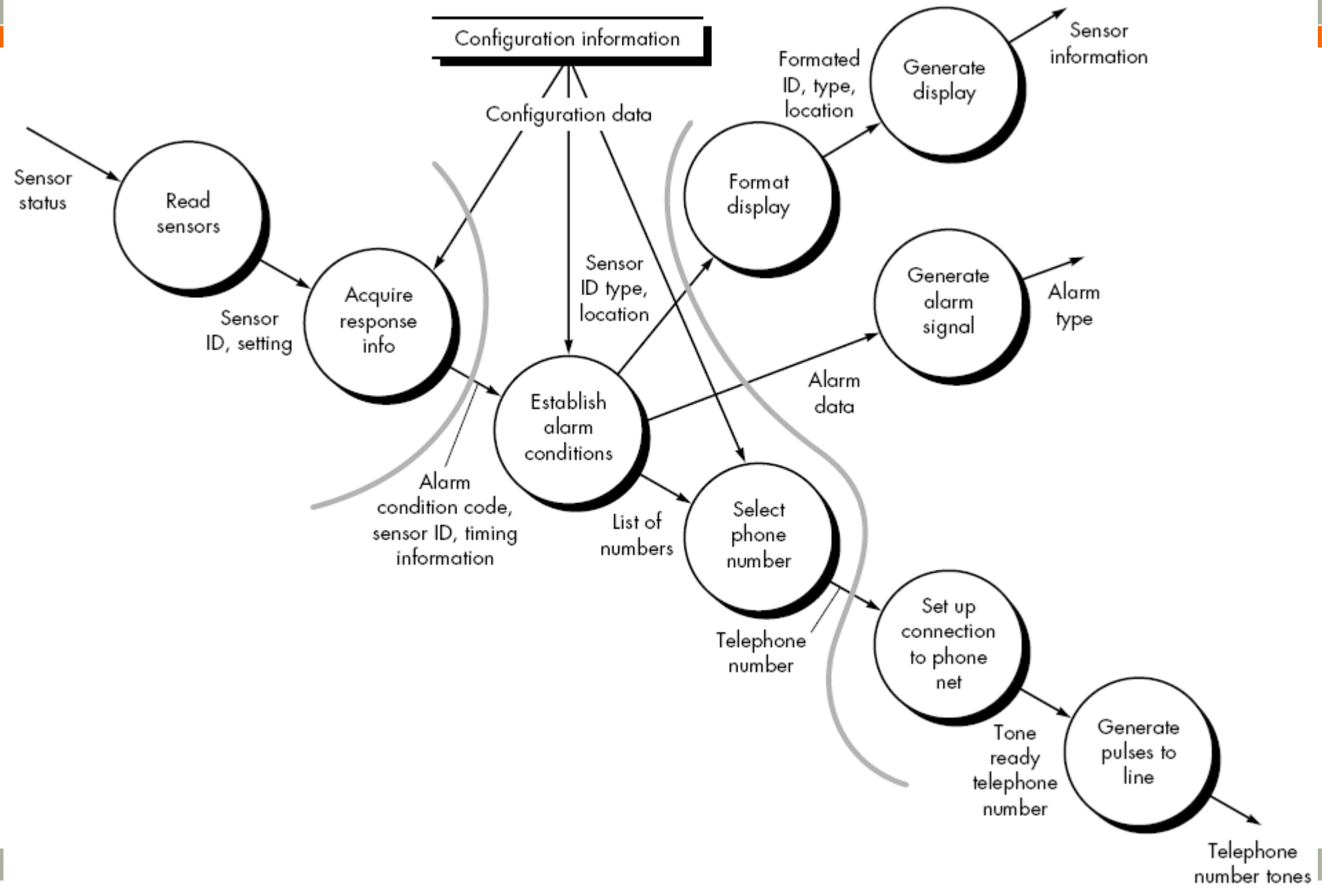
SafeHome State Transition Diagram



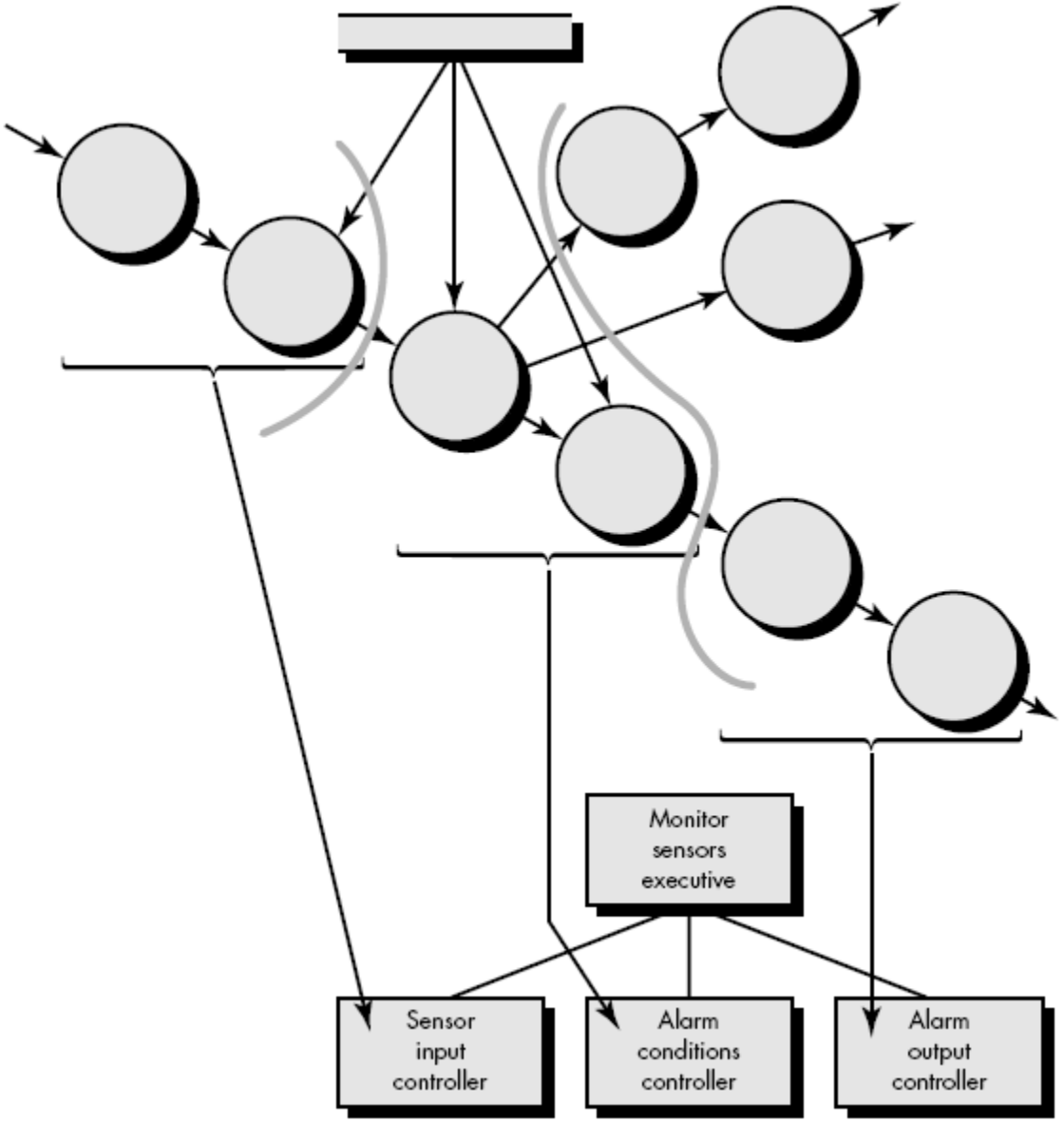
Level-2 DFD ("Monitor sensors")



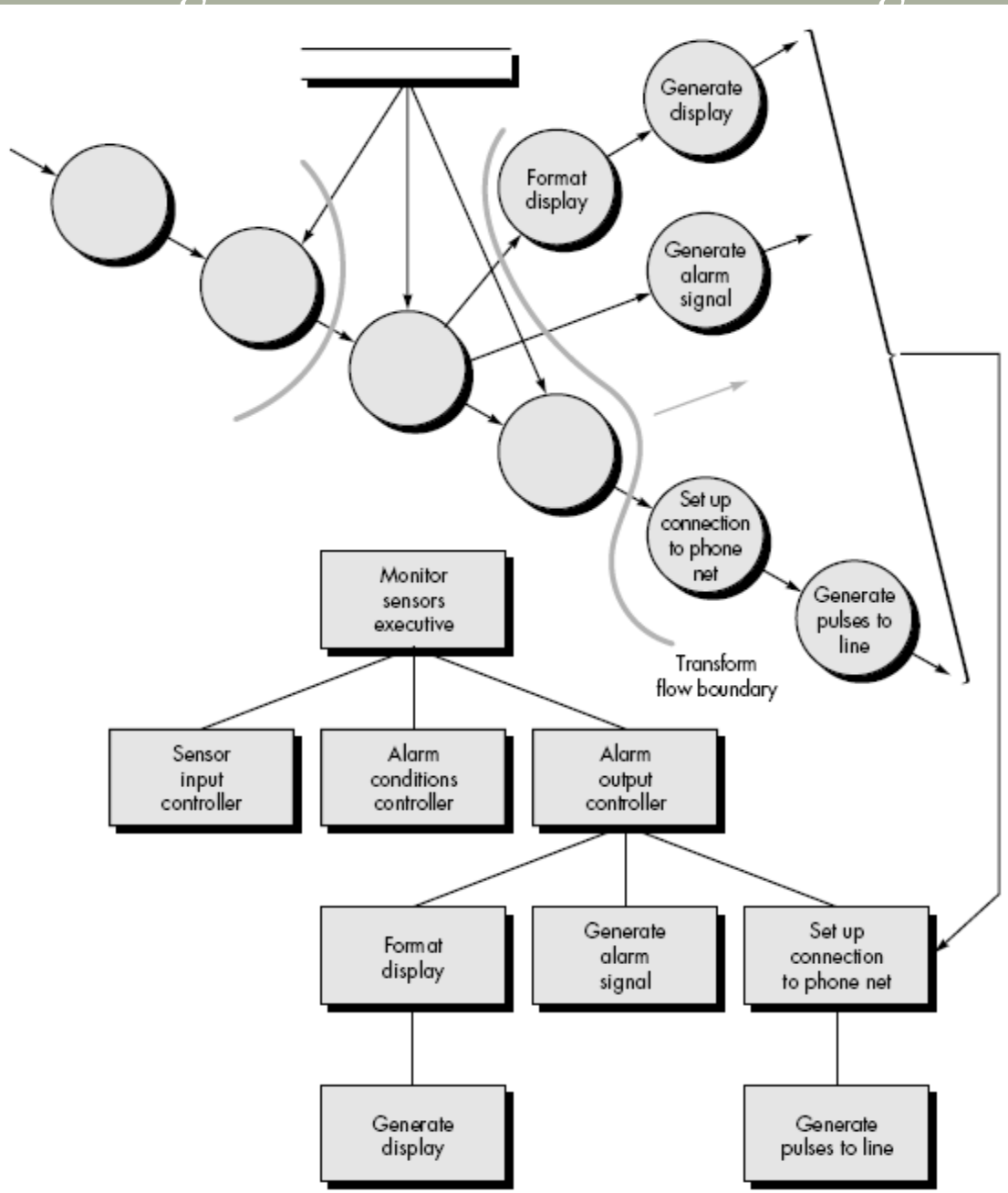
Level-3 DFD ("Monitor sensors")



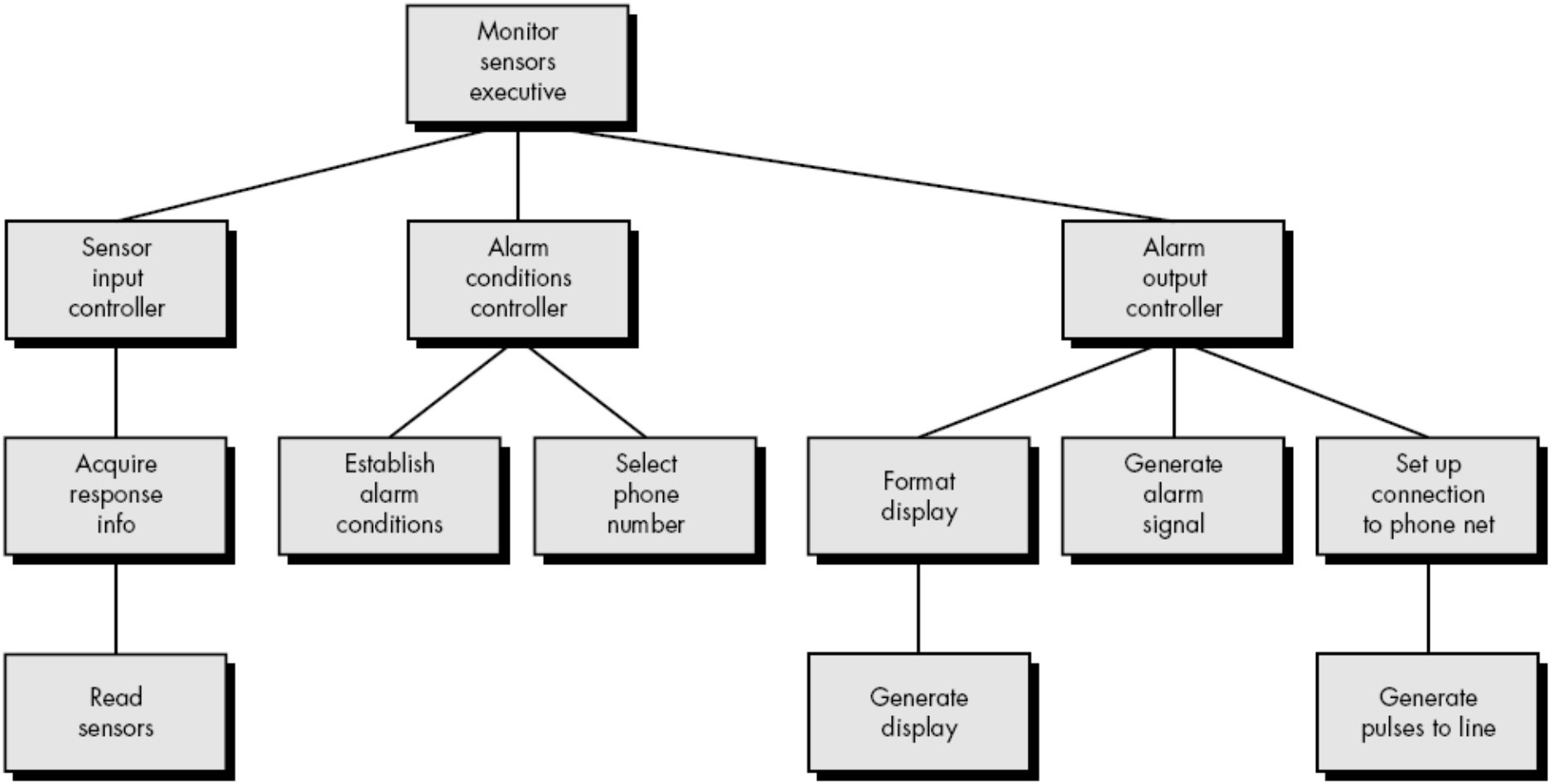
Program Structure: First factoring ("Monitor sensors")



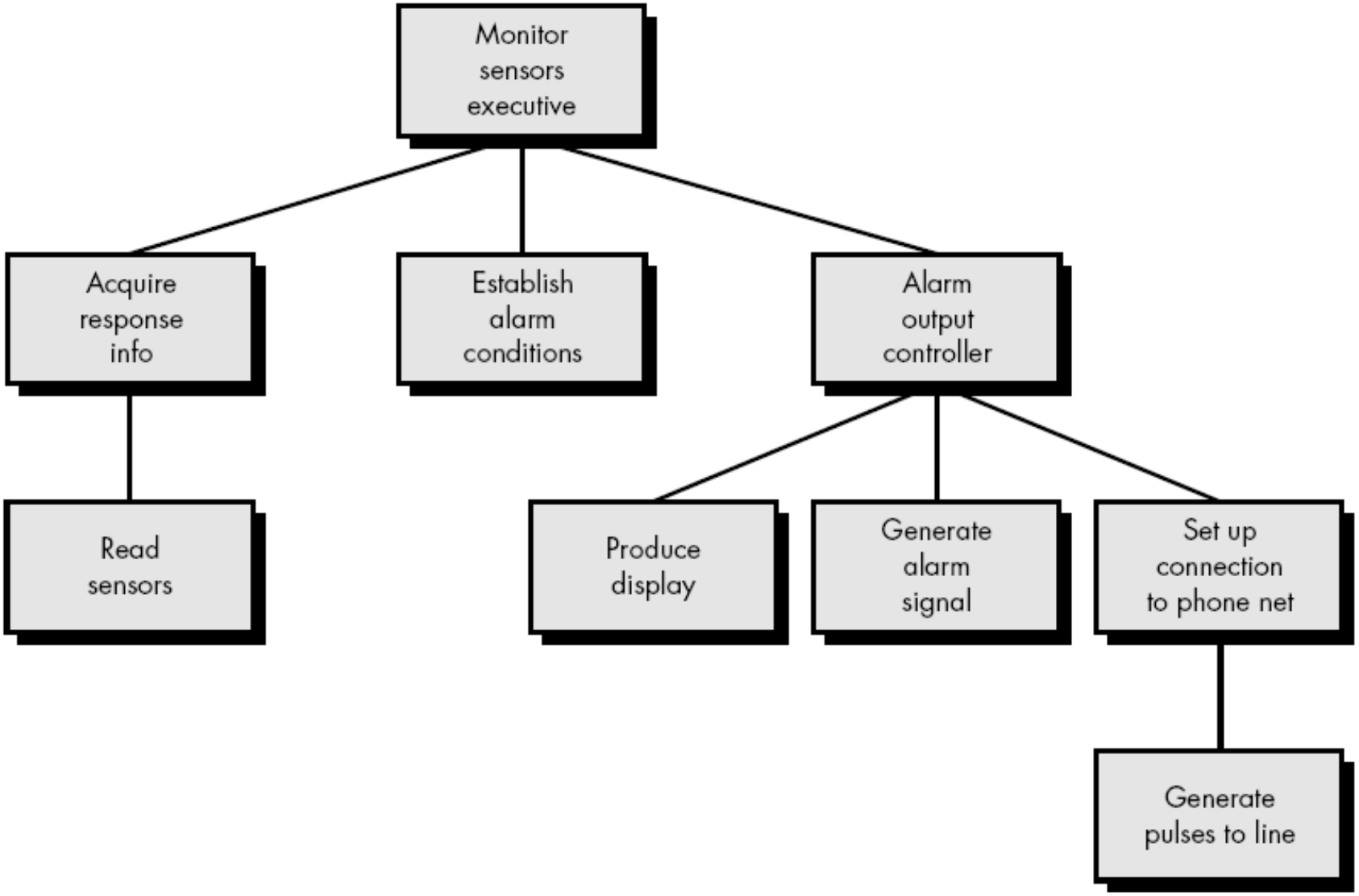
Program Structure: Second factoring



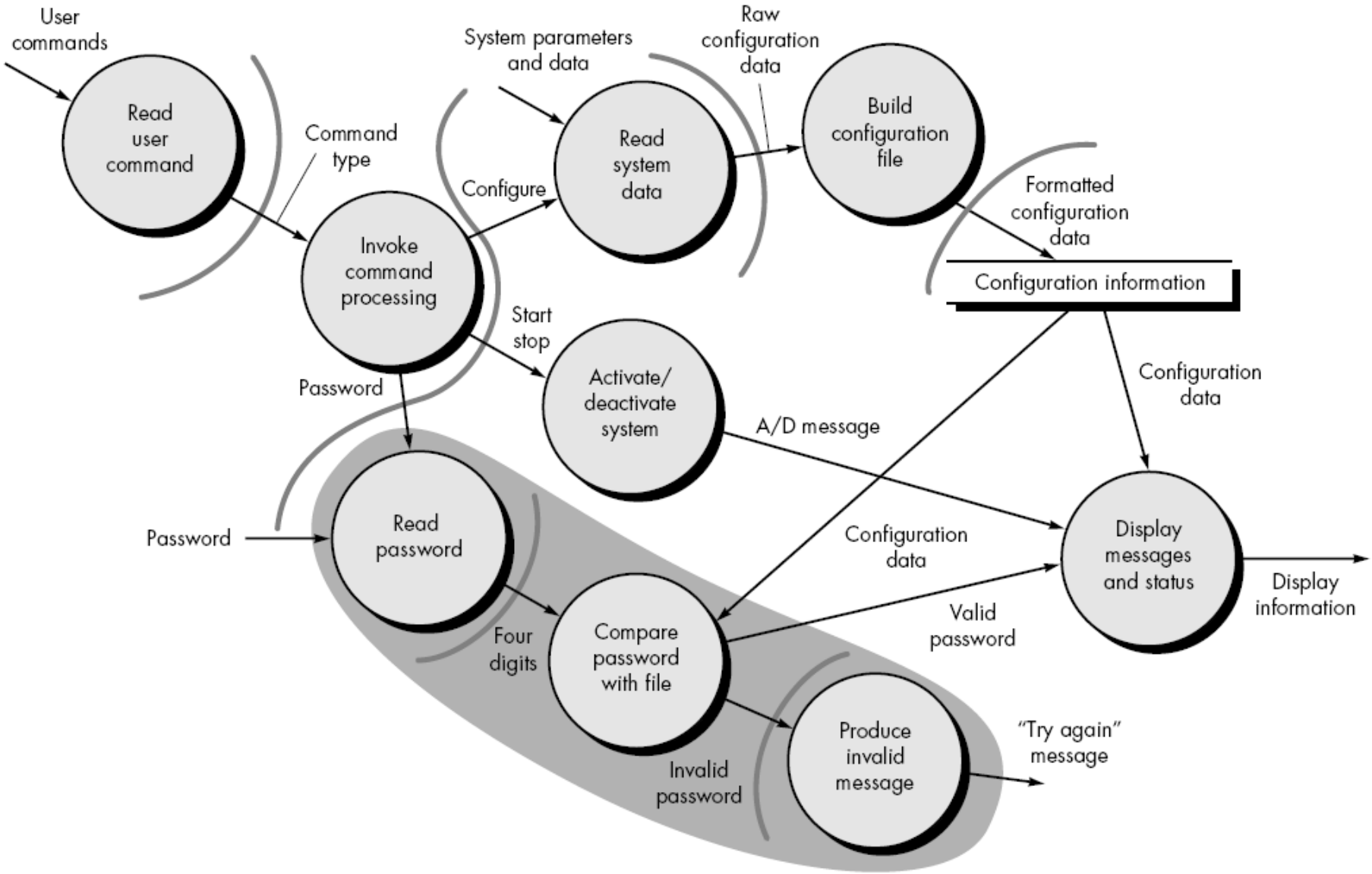
Program Structure: ("Monitor sensors")



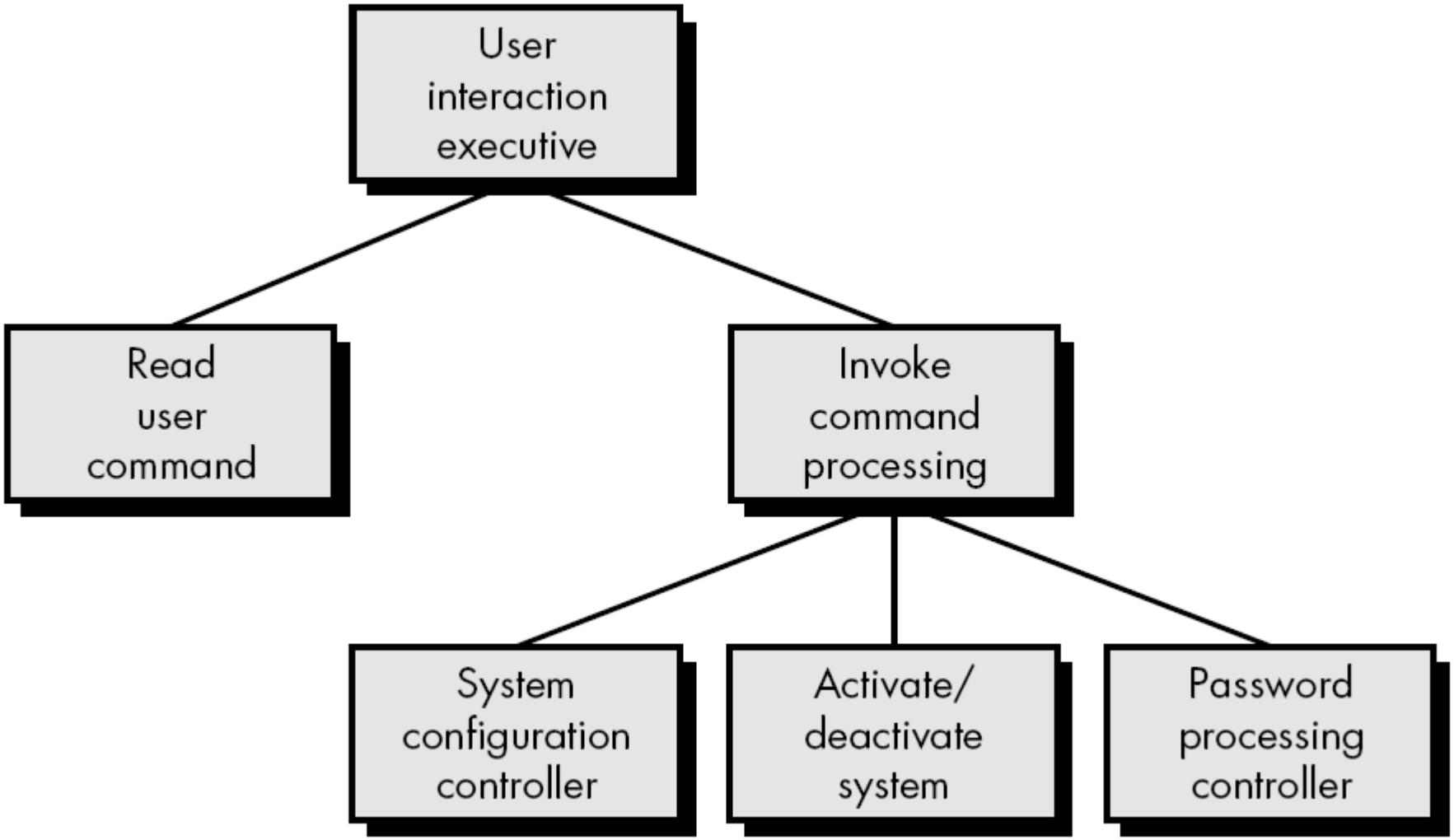
Refined Program Structure: ("Monitor sensors")



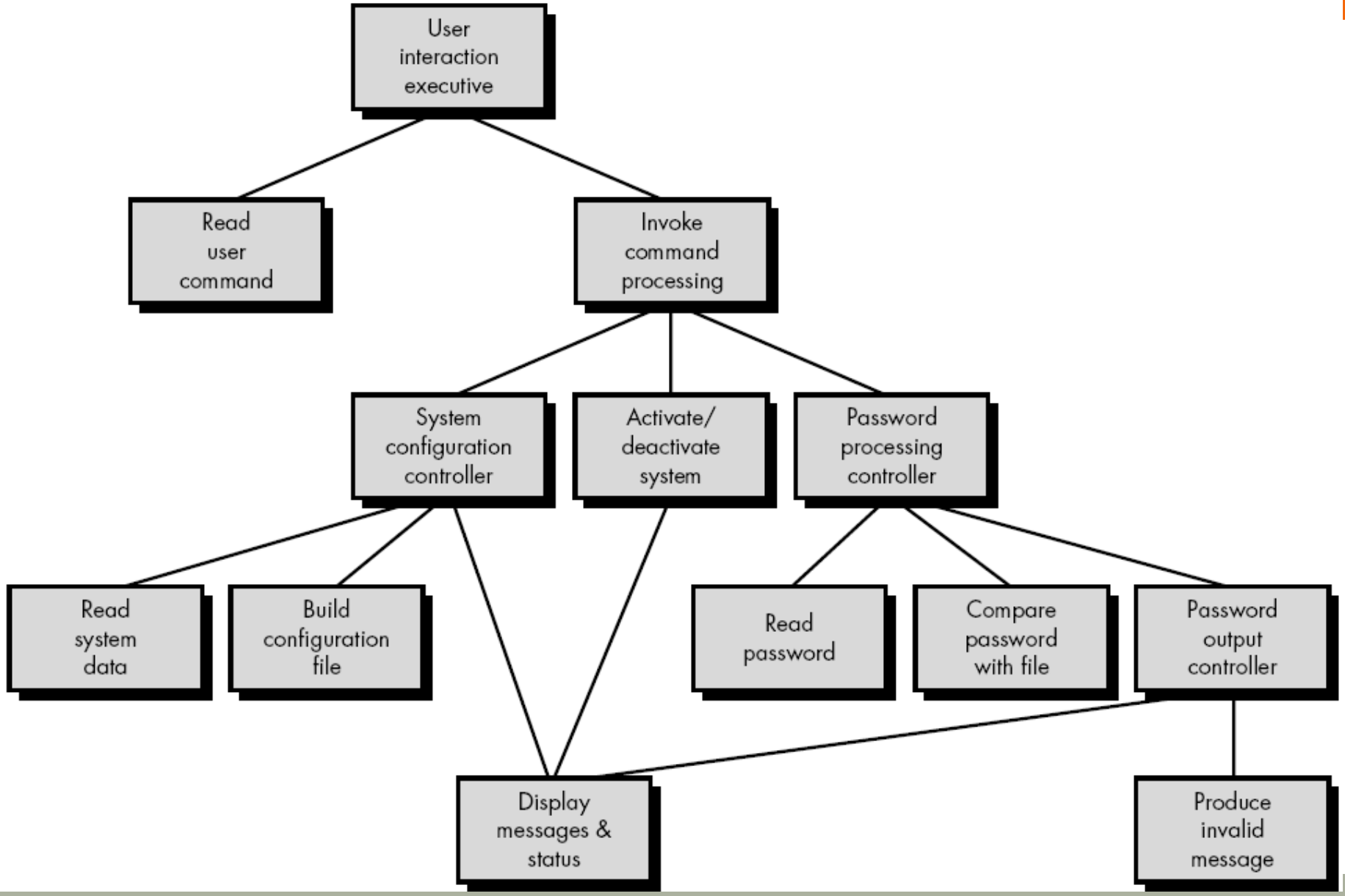
Level-2 DFD (user interaction subsystem)



Program Structure: First factoring (user interaction subsystem)



Program Structure:(user interaction subsystem)



1. Software Design Concepts
2. Structured Design
   2.1. Case Study: SafeHome
3. Object Oriented Design Principles
   3.1. Unified Modeling Language
   3.2. Case Study: Elevator
4. User Interface Design

Unified Modeling Language

8.3.1

What is UML?

- Unified Modeling Language (UML) is the standard tool for visualizing, specifying, constructing, and documenting the artifacts of an object-oriented software.
- UML is not a programming language, but only a visual design notation.
- Can be used with all software development process models.
- Independent of implementation language.
- Many CASE tools uses UML for automatic code generation. Examples: IBM Rational, ArgoUML, etc.
- You may be familiar with some UML concepts introduced in Object Oriented Programming course.

Background

- UML is the result of an effort to simplify and consolidate the large number of **Object Oriented** development methods and notations.

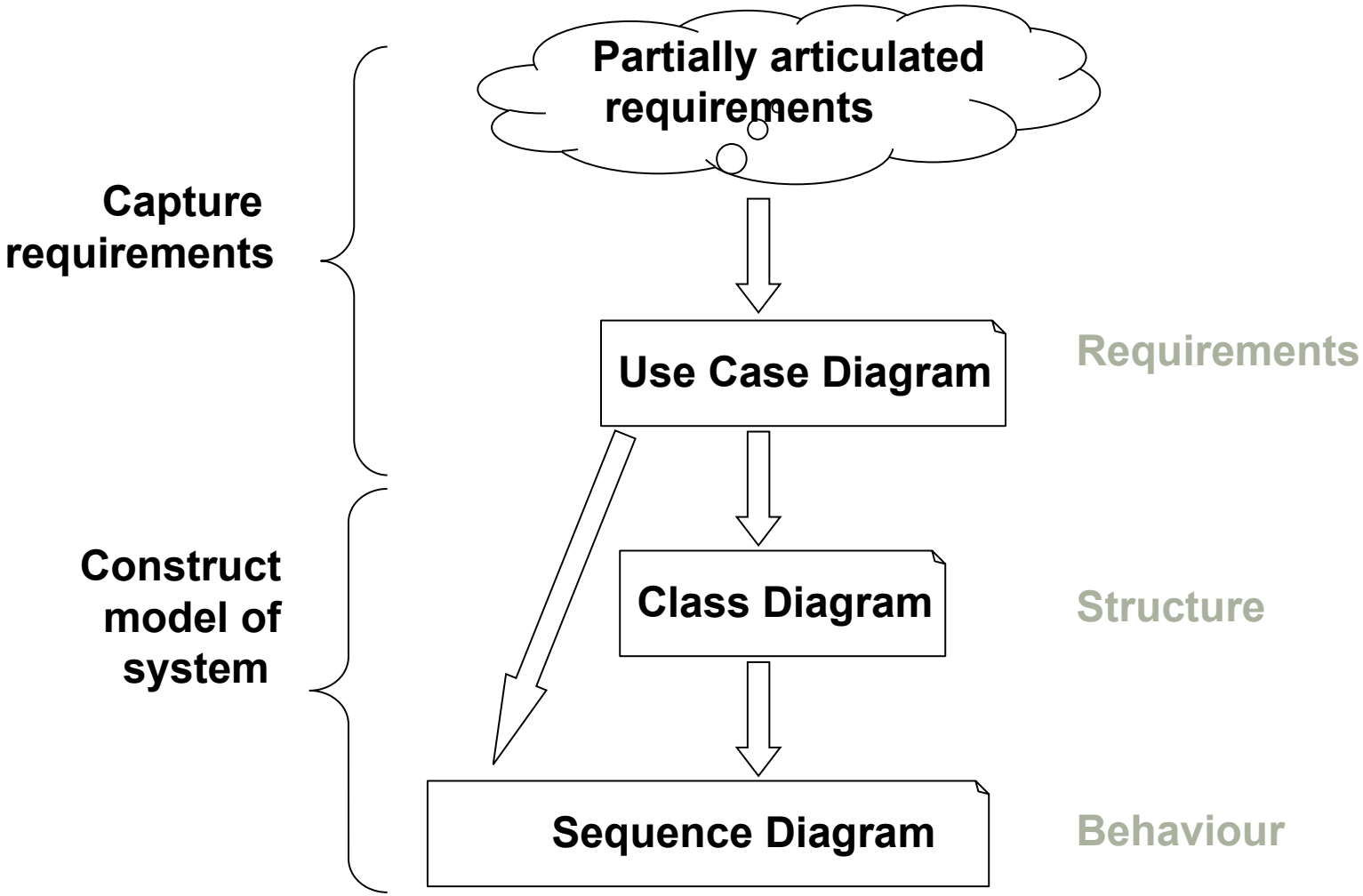
- Developed by the Object Management Group based on work from:
  - Grady Booch [91]
  - James Rumbaugh [91]
  - Ivar Jacobson [92]

- The latest version is UML 2.5 (Dec 2017)
  (See http://www.omg.org or http://www.uml.org)

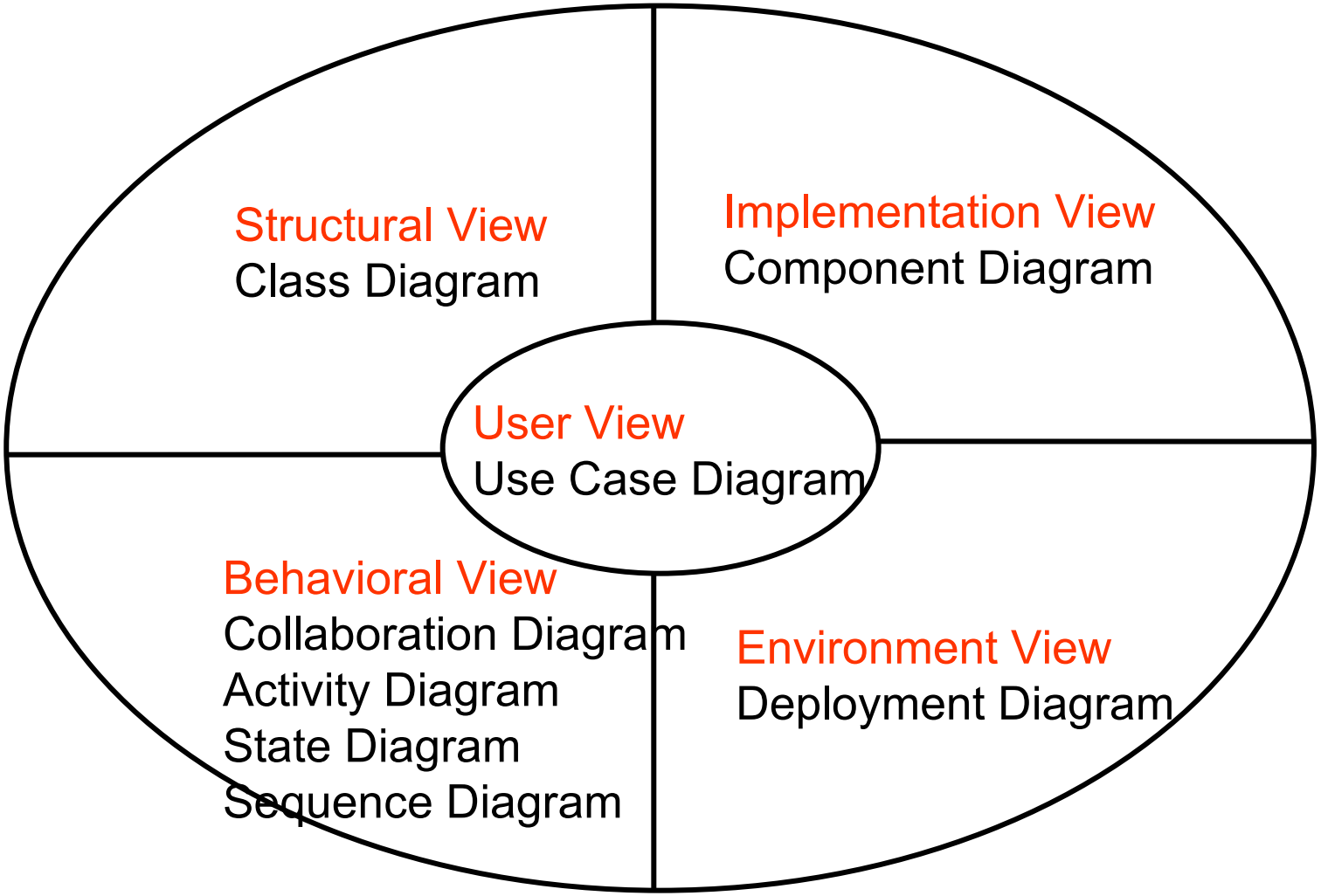
Types of UML Diagrams

- **Use Case Diagrams (*)**
- **Class Diagrams (*)**
- Interaction Diagrams
  - **Sequence Diagrams (*)**
  - Collaboration Diagrams
- State Transition Diagrams
- Activity Diagrams
- Implementation Diagrams
  - Component Diagrams
  - Deployment Diagrams

Minimal UML Process



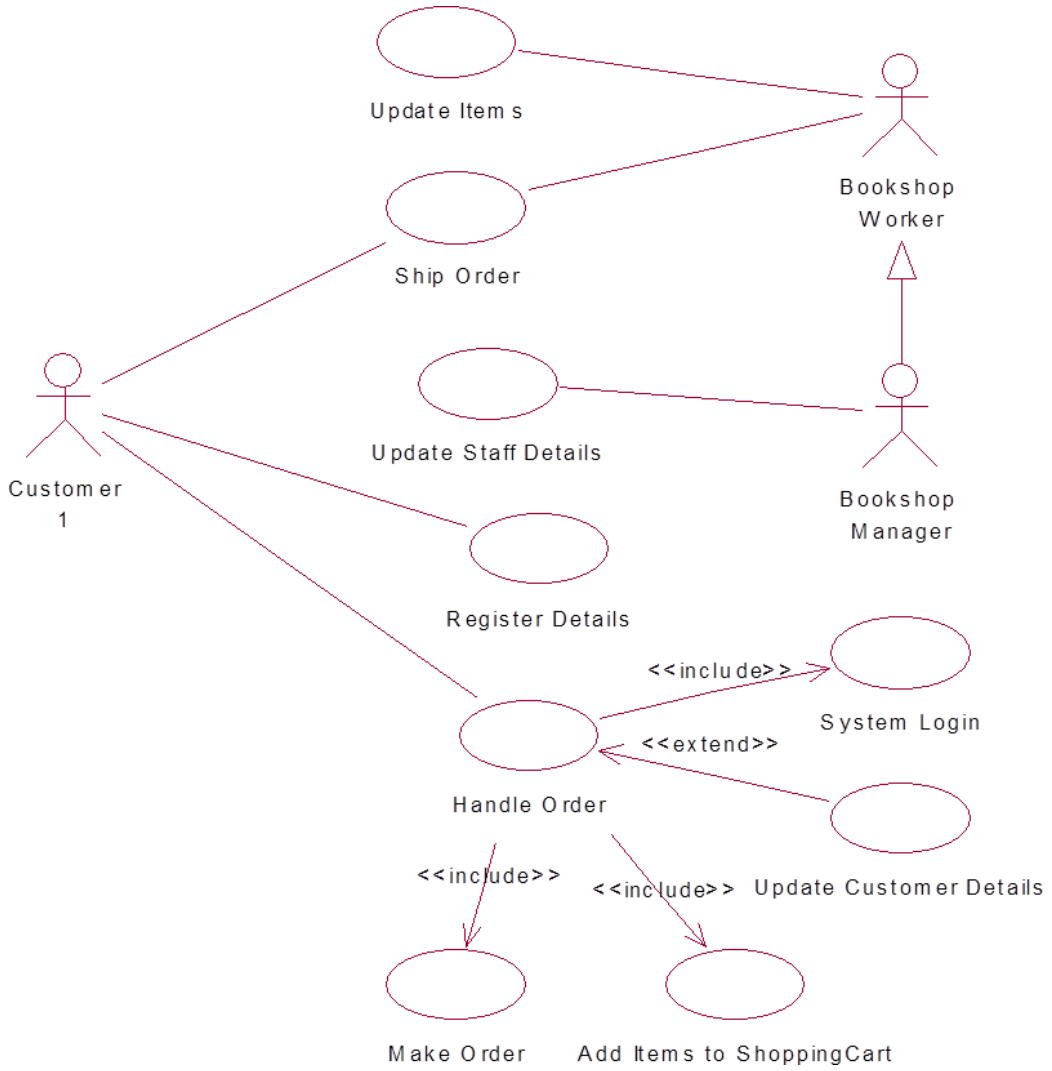
Views of UML Diagrams



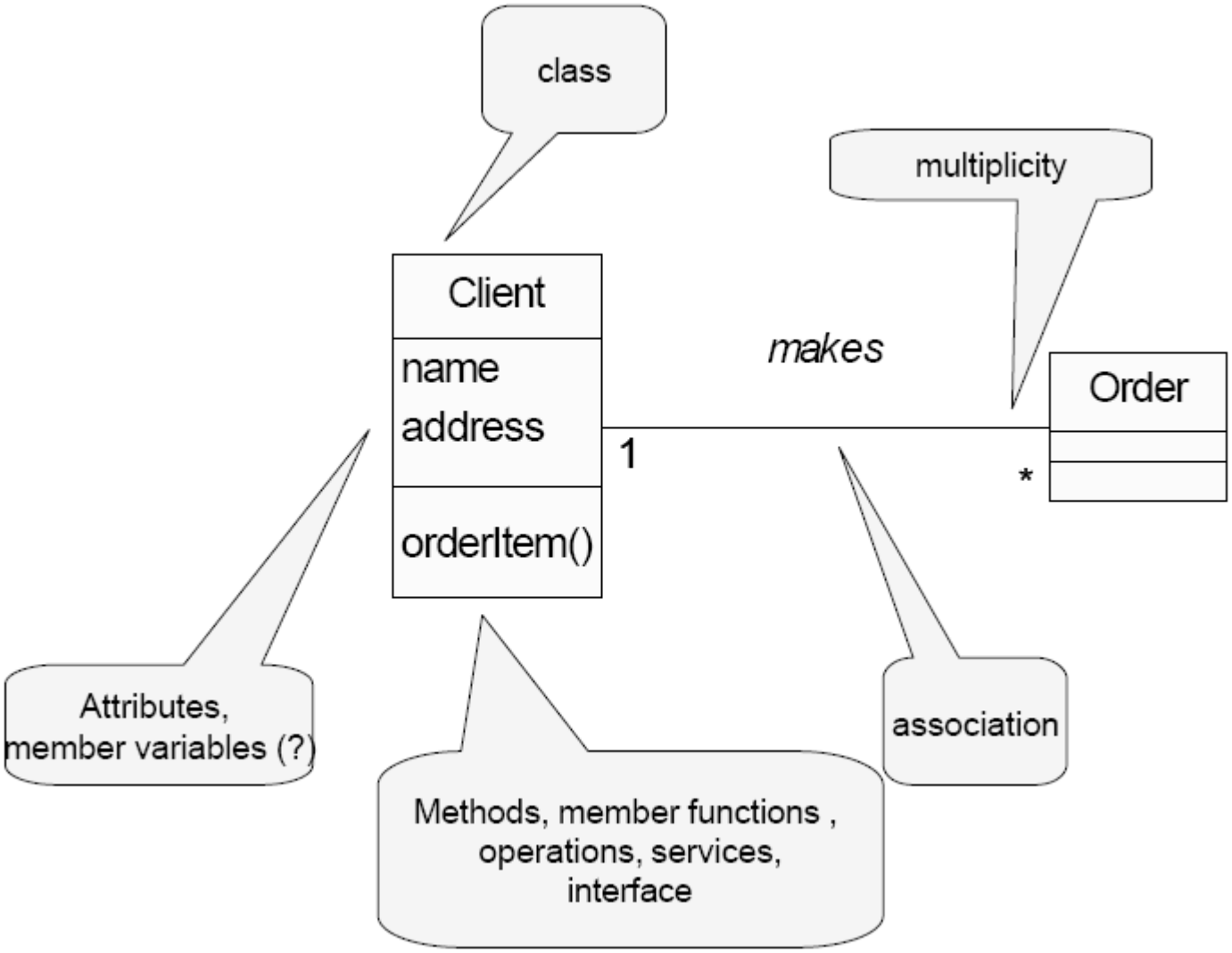
Use Cases

- Use case diagrams are used to visualize, specify, construct, and document the (intended) behavior of the system, during requirements capture and analysis.
- Provide a way for developers, domain experts and end-users to Communicate.
- Serve as basis for testing.

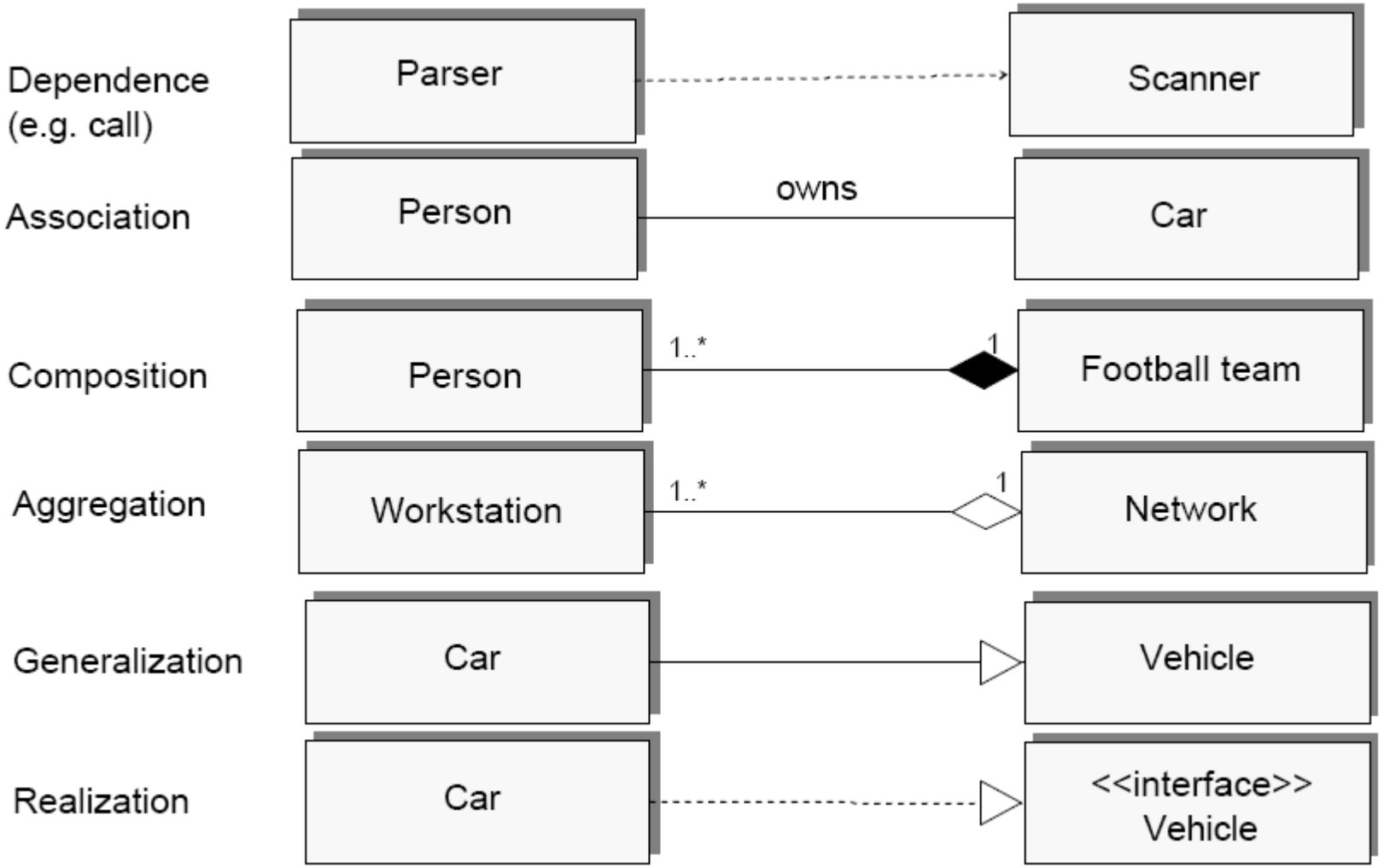
Example : Use Case Diagram



Class Diagrams

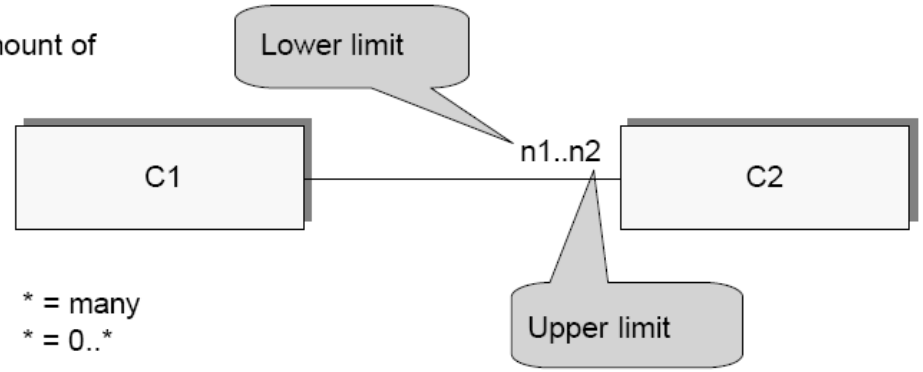


Class Relations

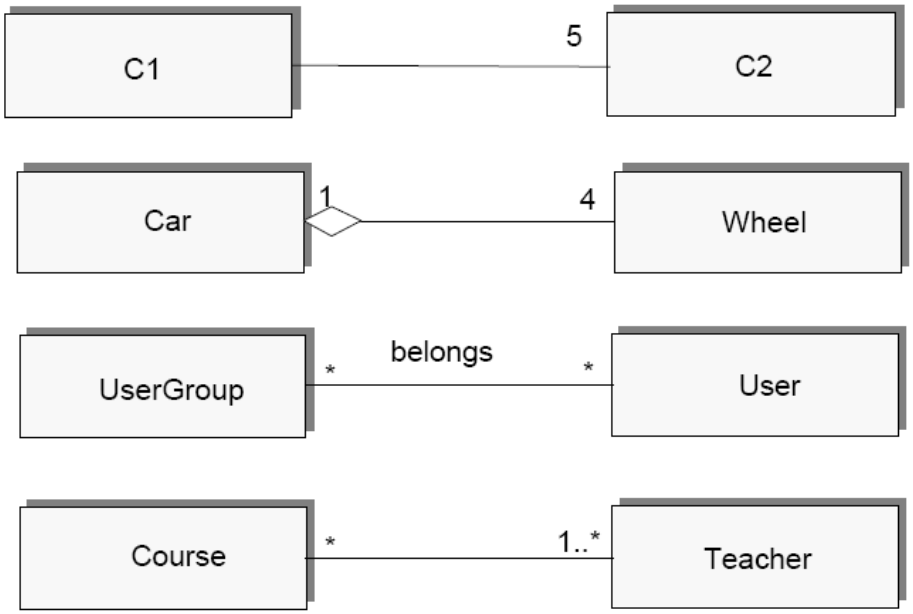


Cardinality

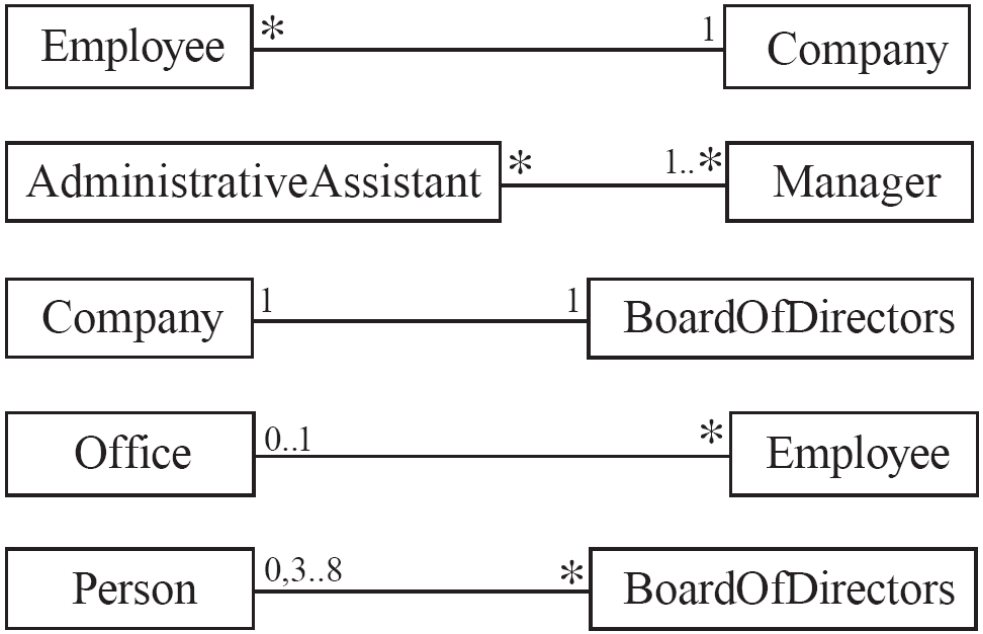
Varying amount of instances

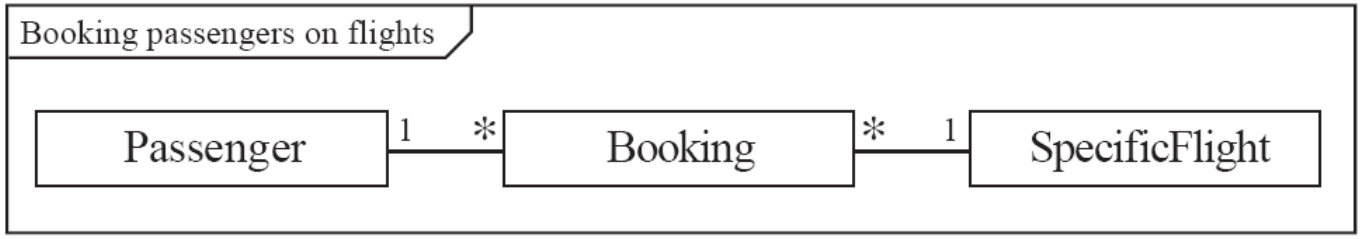


Static amount of instances



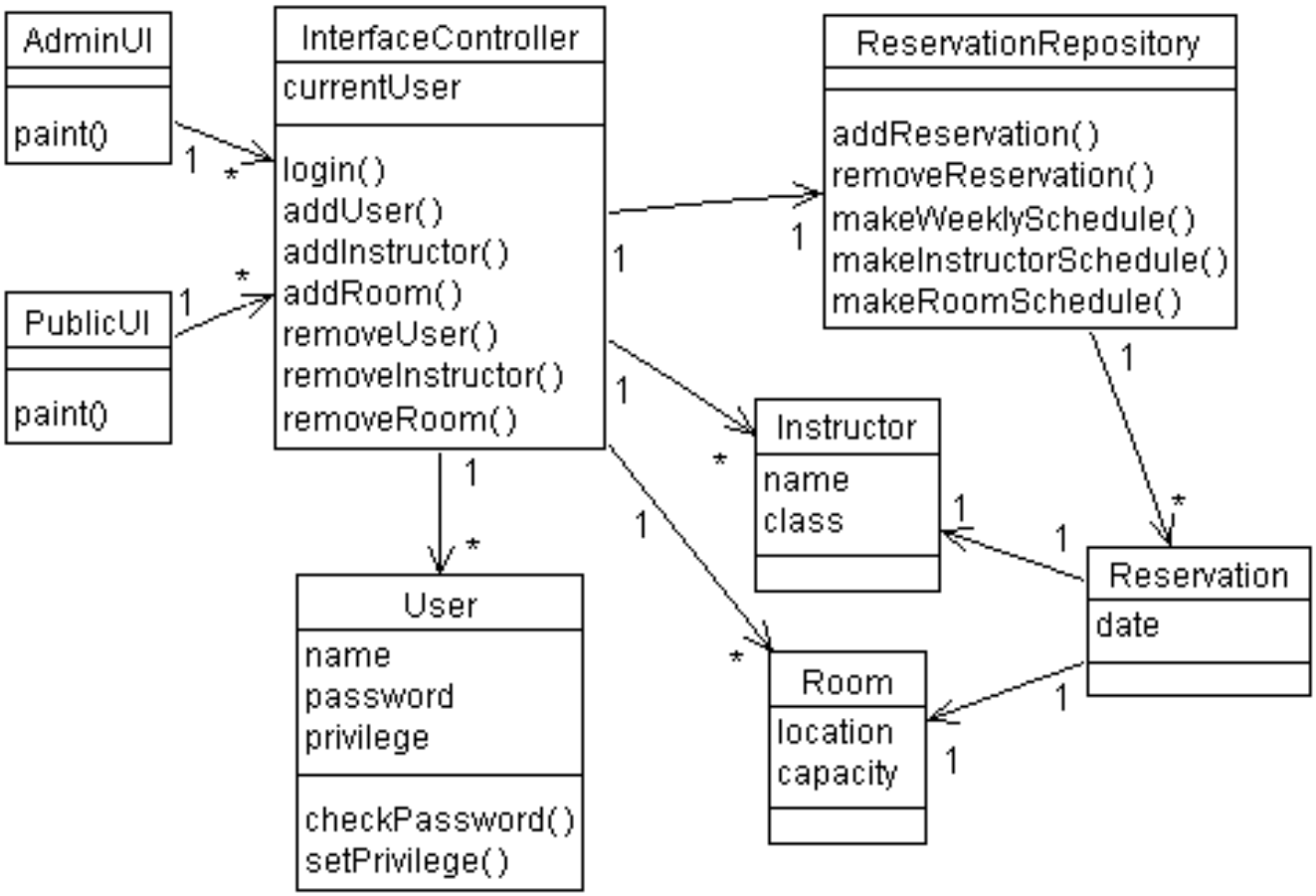
Class relations (examples)



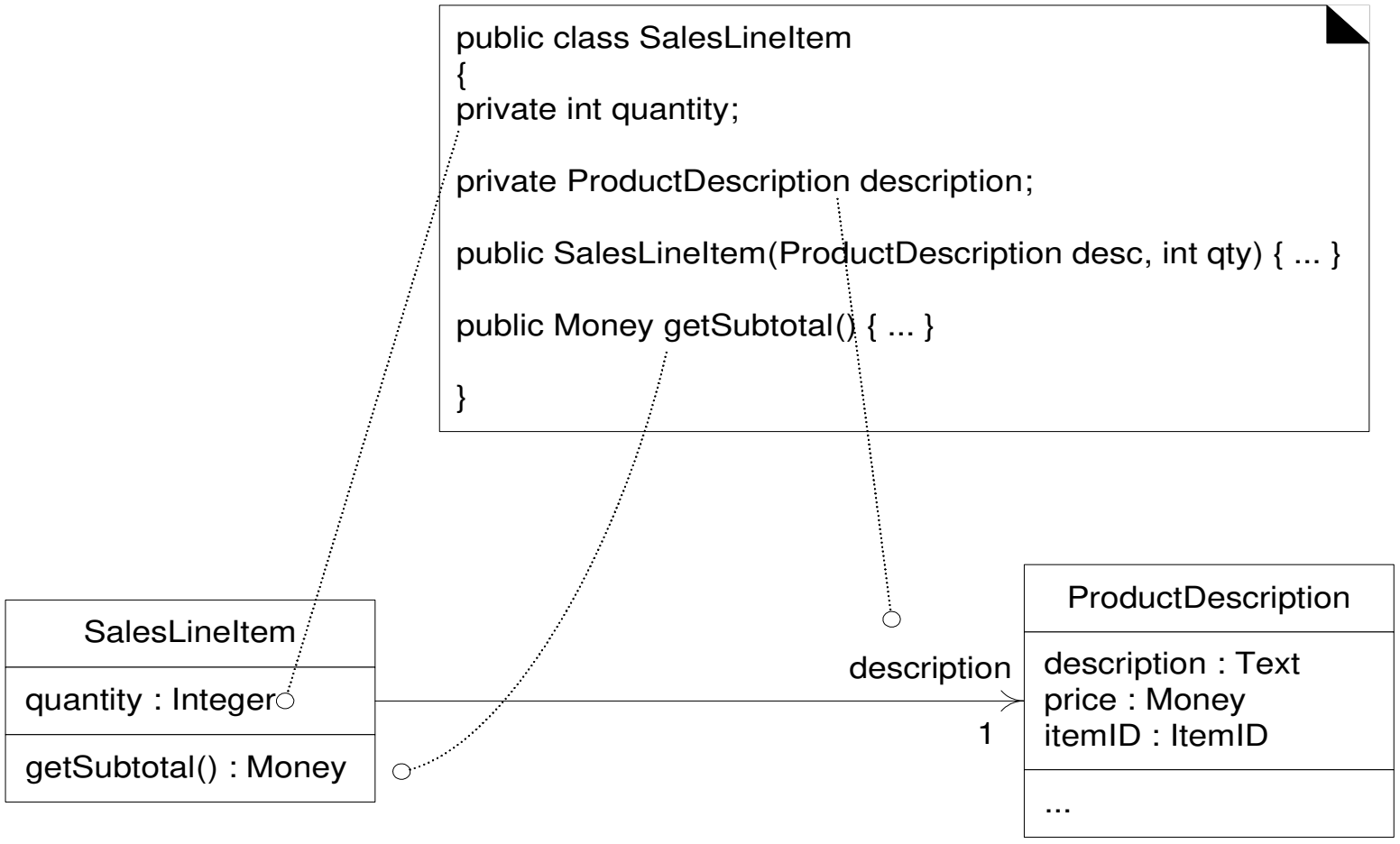


Example : Class Diagram

A classroom scheduling system: specification perspective.



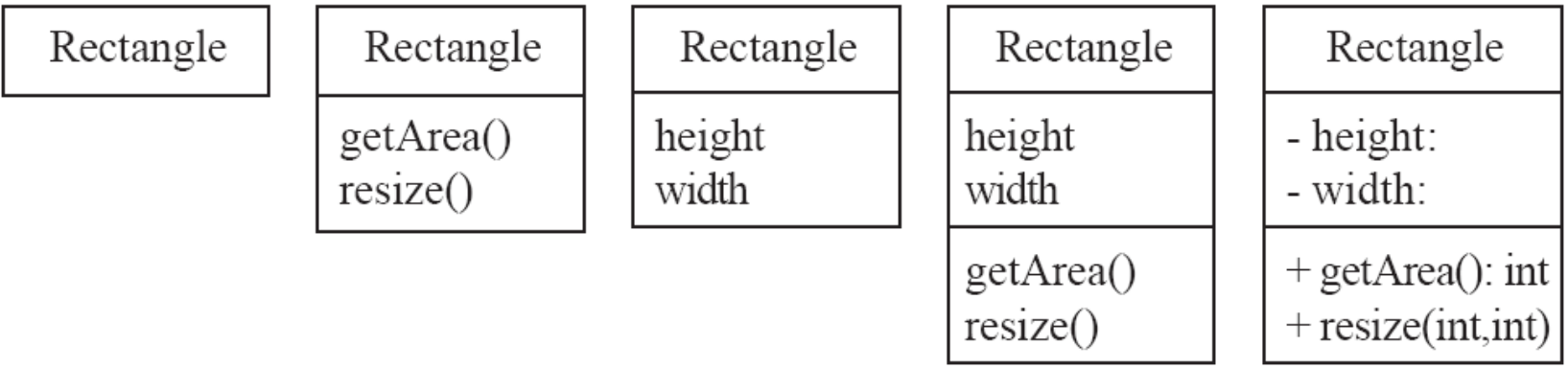
Example : From Class Diagram to Code



Class diagrams

- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType



Notice the character in front of the variables and functions

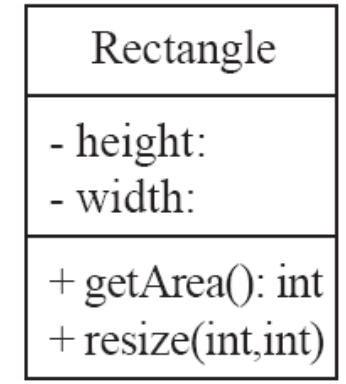
Plus (+) denotes **public**

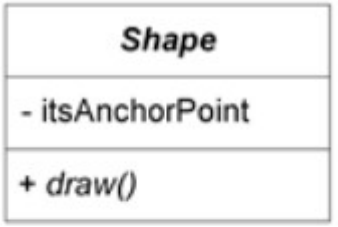
Minus (-) denotes **private**

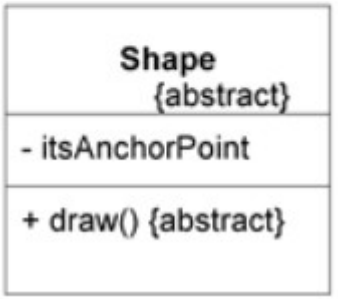
Hash (#) denotes **protected**

Class diagrams

```
public class Rectangle {
        private int height;
        private int width;

        public int getArea() { ... }
        public void resize(int x, int y) { ... }
}
public abstract class Shape {
        private Point itsAnchorPoint;
        public abstract void draw();
}
```







Propagation

```
public class Vehicle{
    private Collection <VehiclePart> part;

   public int getWeight(){
            int weight = 0;
            for(Iterator<VehiclePart> it = part.iterator(); it.hasNext(); ){
                    weight += (it.next()).getWeight();
            }
            return weight;
    }
}
```


```
public class VehiclePart{
        private int weight;
        private Vehicle vehicle;
         public int getWeight(){
                 return weight;
         }
}
```

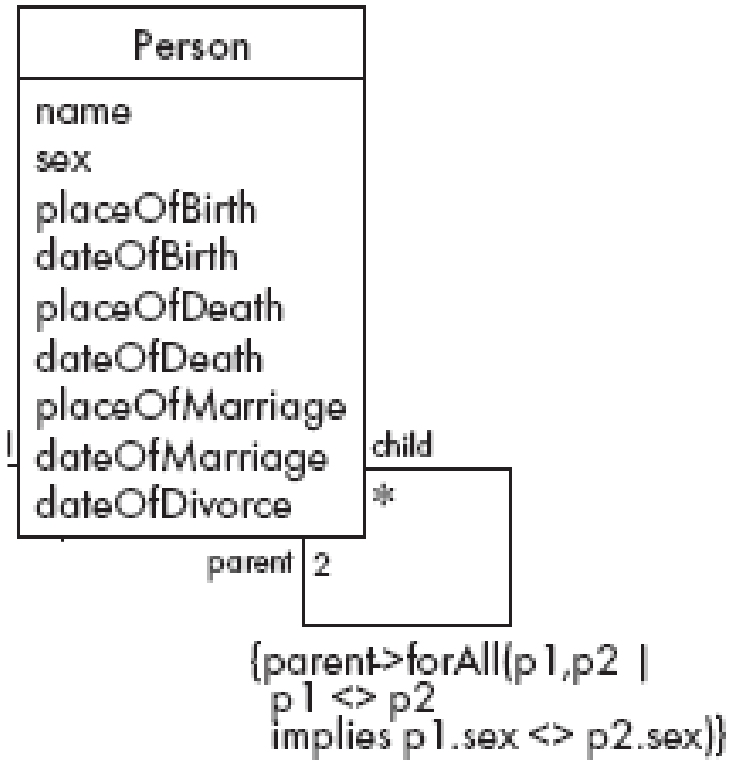
A detailed example for Genealogy problem

- Problem: Imagine you are developing a genealogy system, in which you have to keep track of

1) unions (e.g., marriages and partnerships)

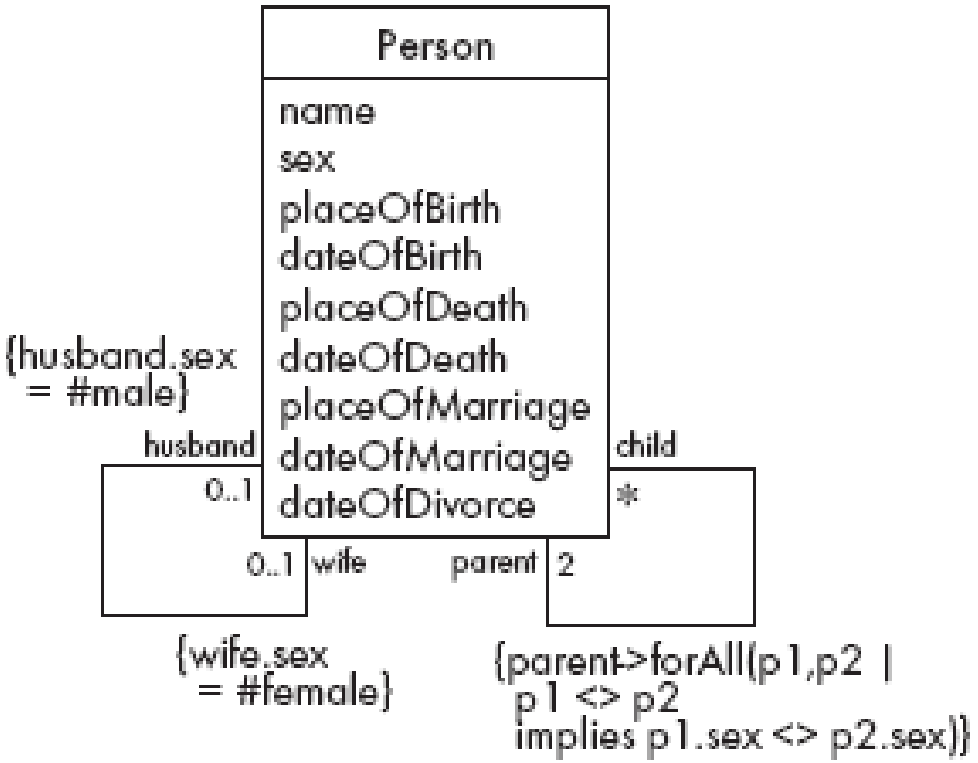
2) parent-child relationships

Detailed Example: A Class Diagram for Genealogy



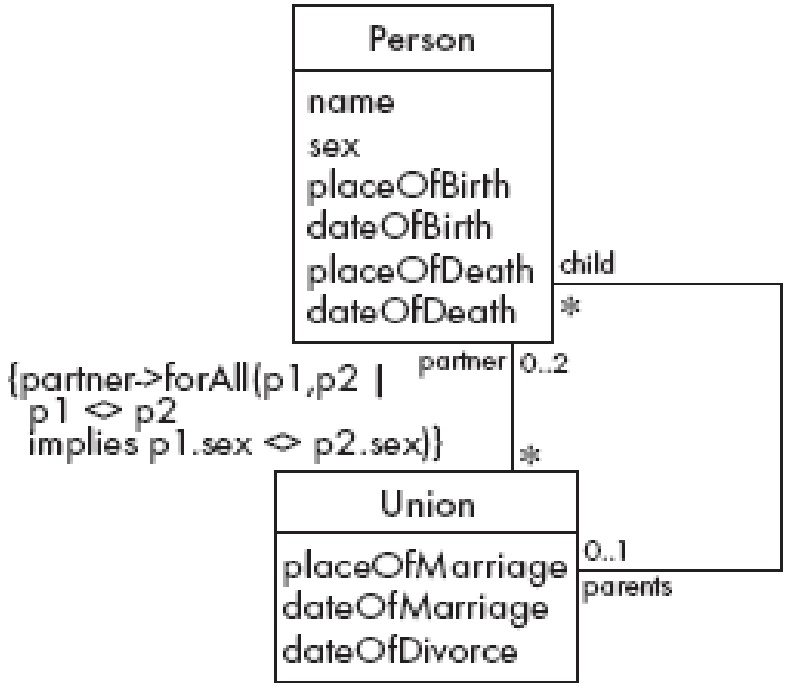
- Problems?

Detailed Example: A Class Diagram for Genealogy

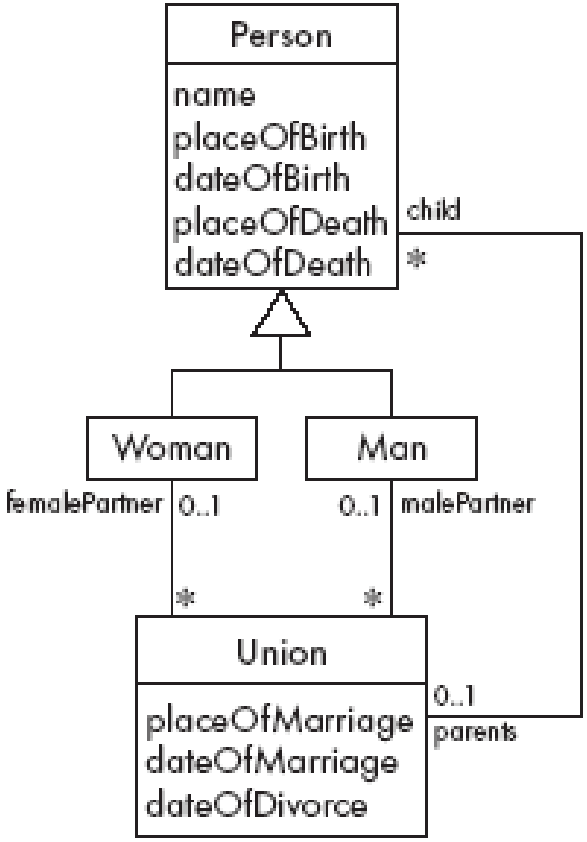


- Problems
  - Marriages not properly accounted for, i.e., a person may have multiple marriages
  - Marriage information (e.g., placeOfMarriage) is duplicated
  - A person must have two parents

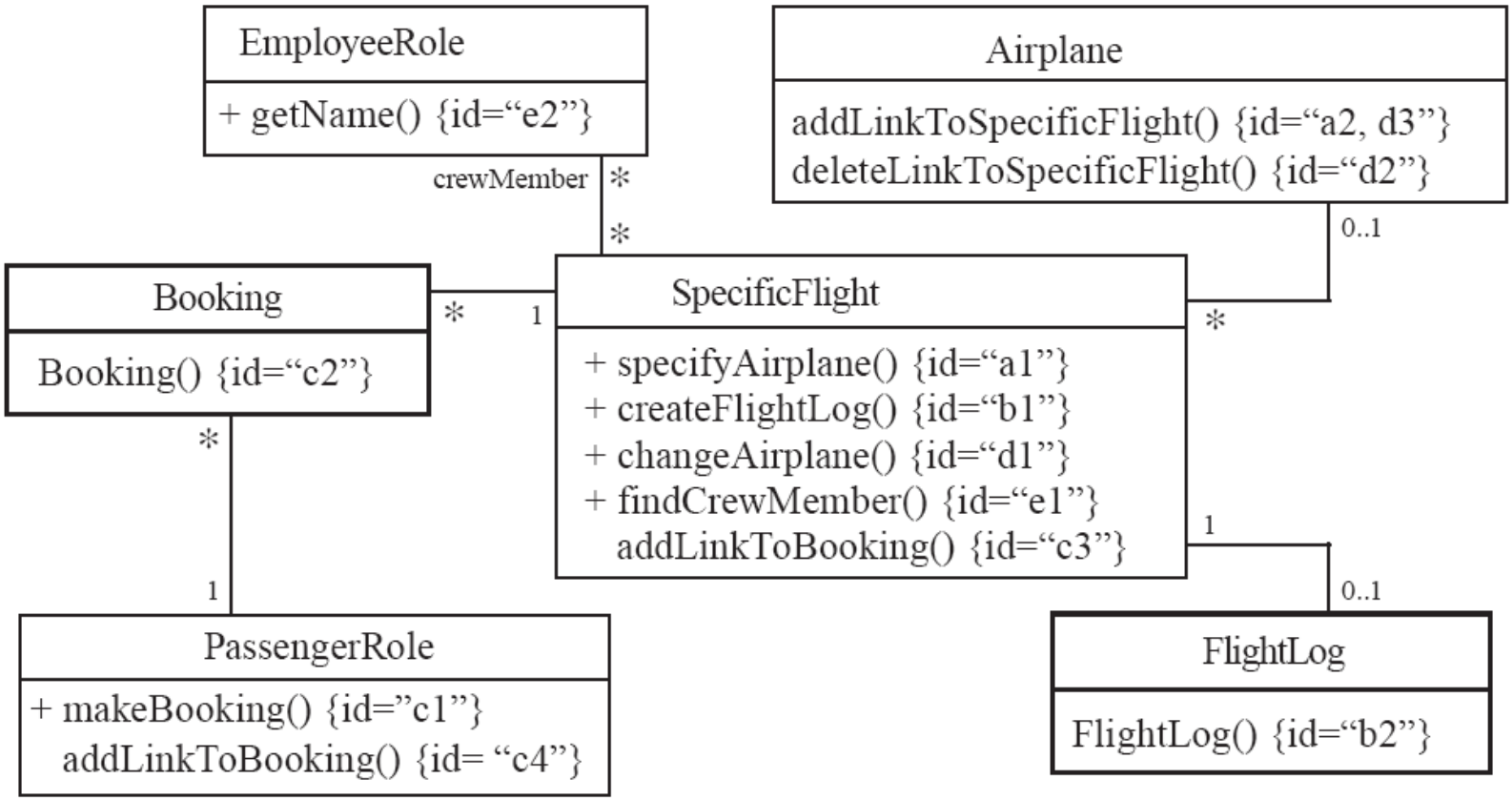
Genealogy example: Possible solutions



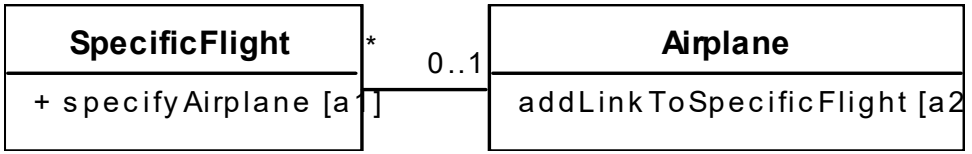
Genealogy example: Possible solutions



Class collaboration



Class collaboration 'a'

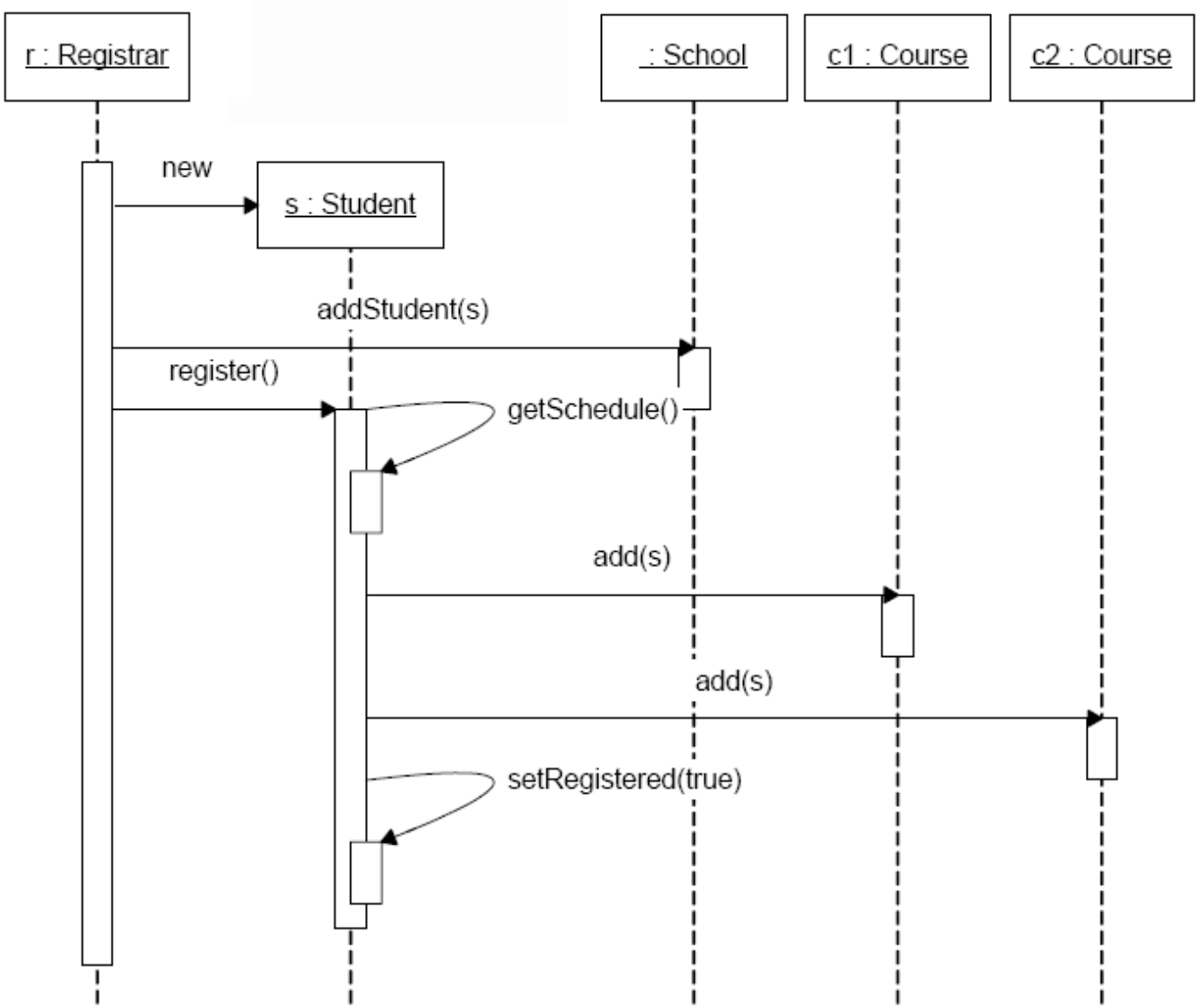



```
package flight.booking;
public class SpecificFlight{
        private Airplane airplane;

    public void specifyAirplane(Airplane airplane){
                this.airplane = airplane;
        this.airplane.addLinkToSpecificFlight(this);
   }
}
package flight.booking;
public class Airplane{
    private Collection <SpecificFlight>  specificFlight;

      void   addLinkToSpecificFlight(SpecificFlight   f){
    specificFlight.add(f);
   }
}
```

Interaction Diagrams

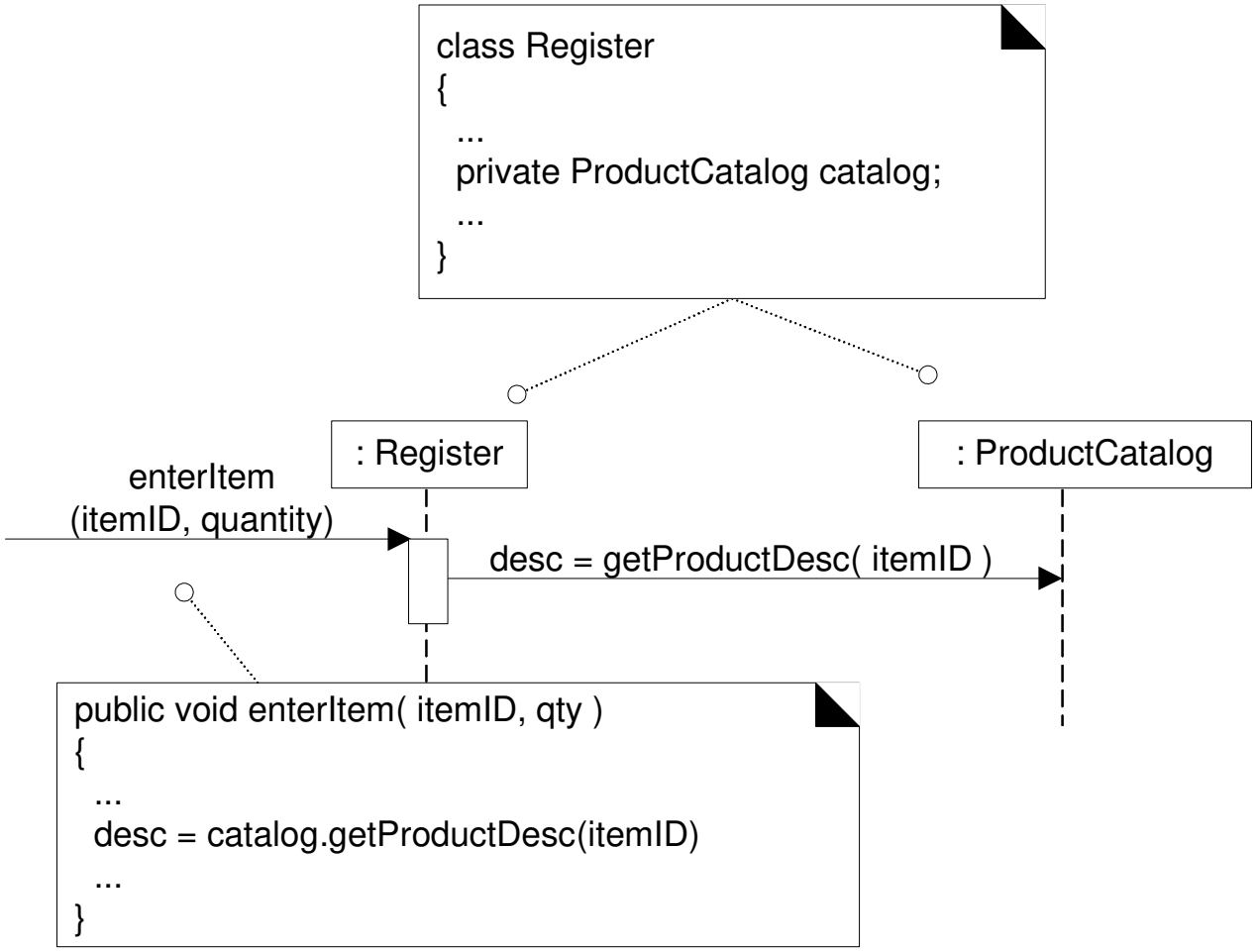
- UML presents two types of interaction diagrams.
  1. Sequence Diagrams
  2. Collaboration Diagrams

Sequence Diagrams

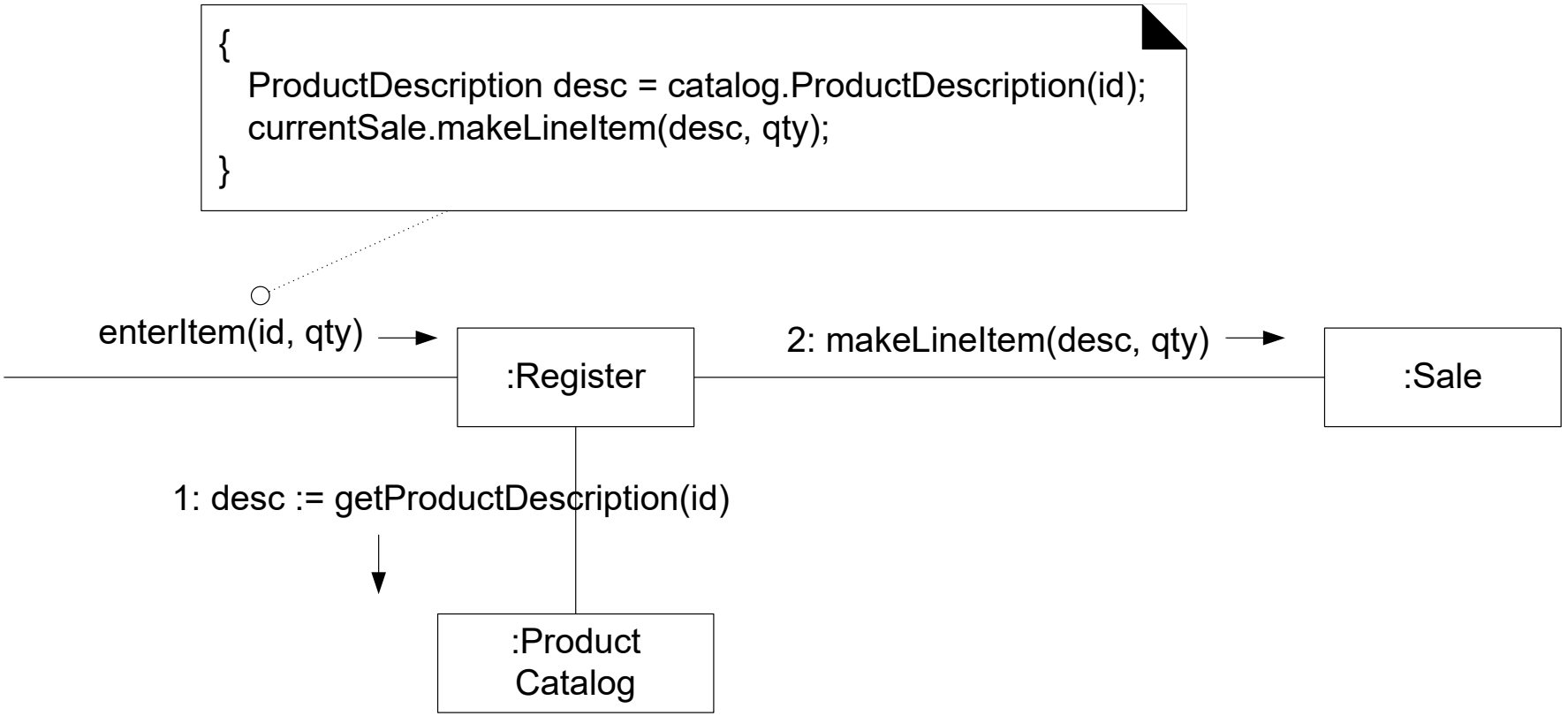


Example : Mapping Diagram to Code

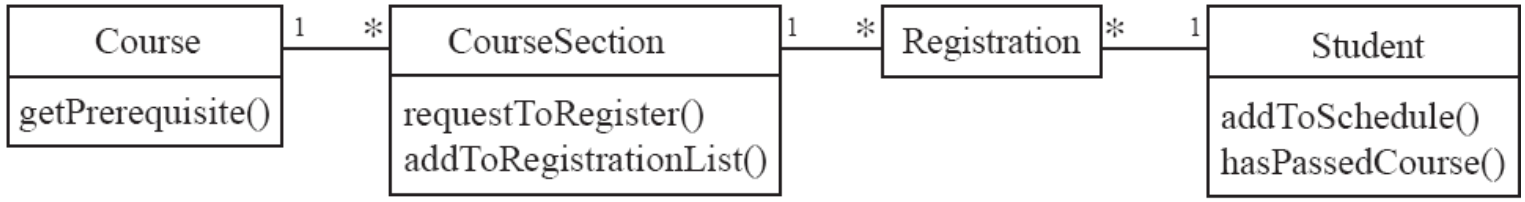
- Mapping sequence diagram to Java code

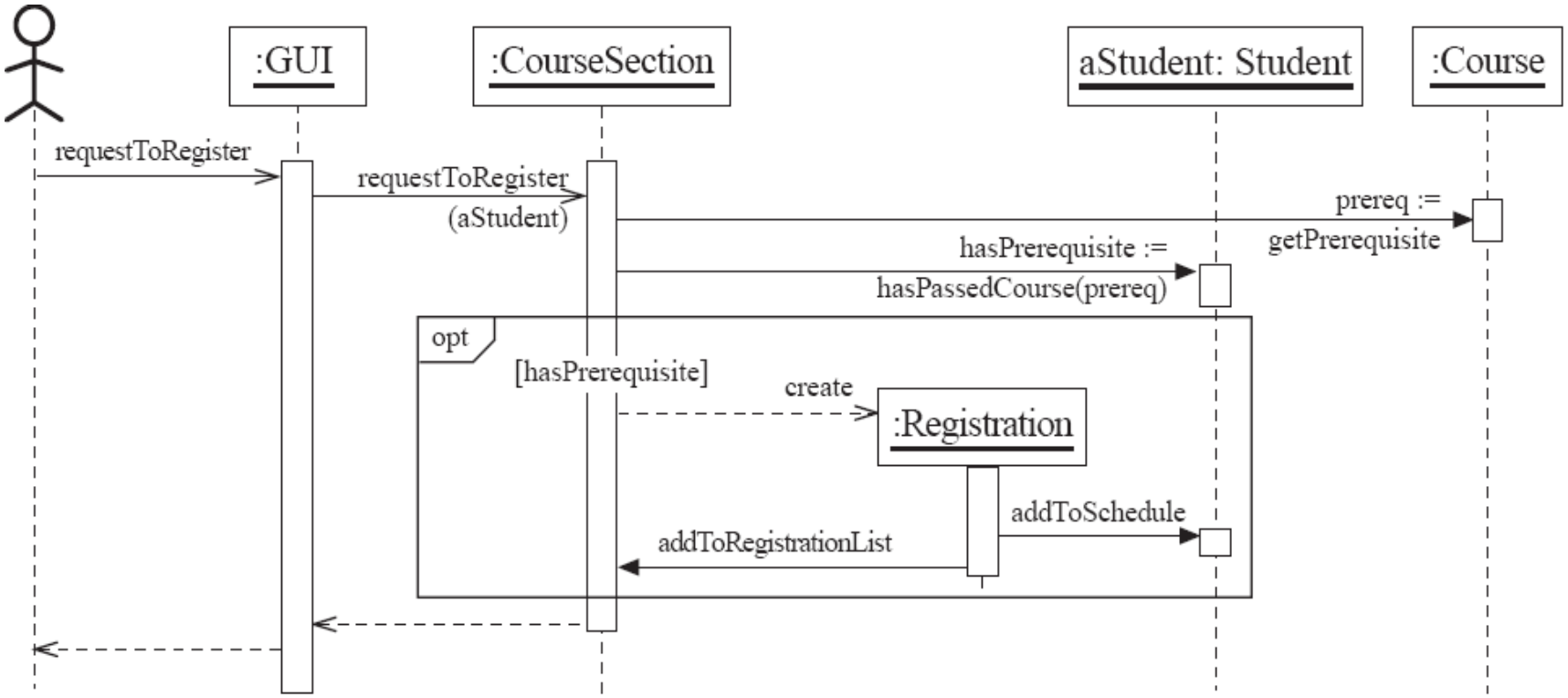


Collaboration Diagrams

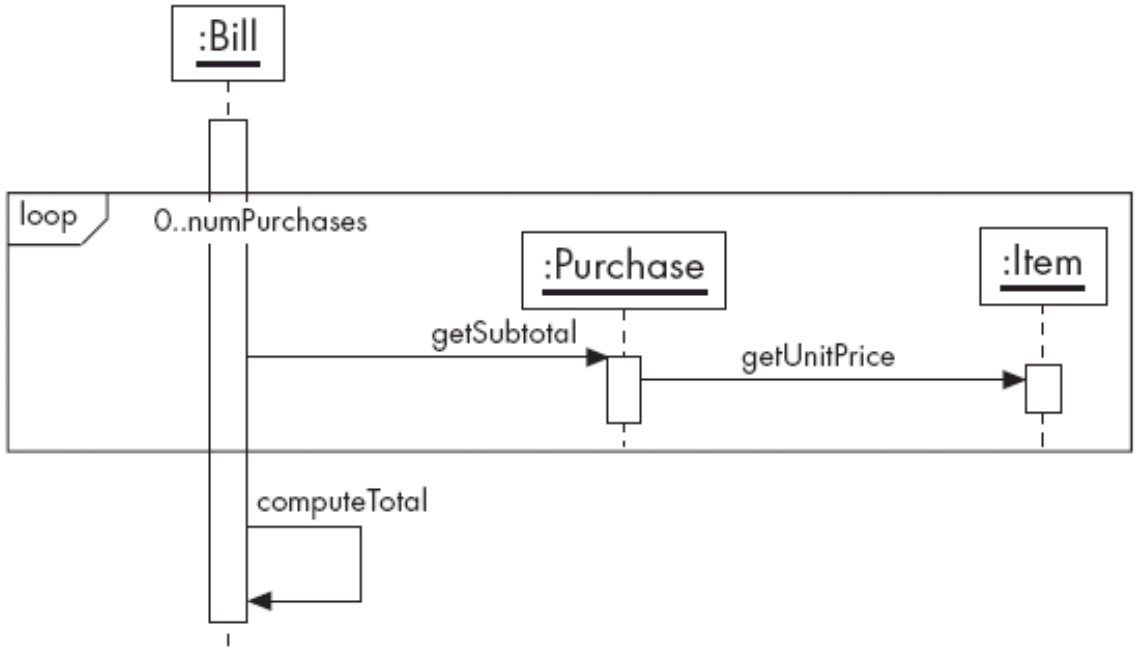


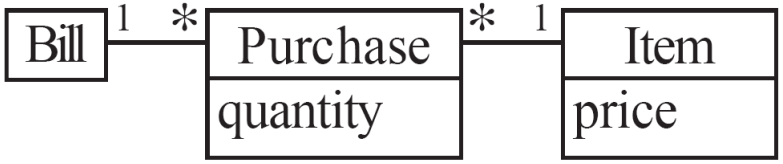
Conditional cases





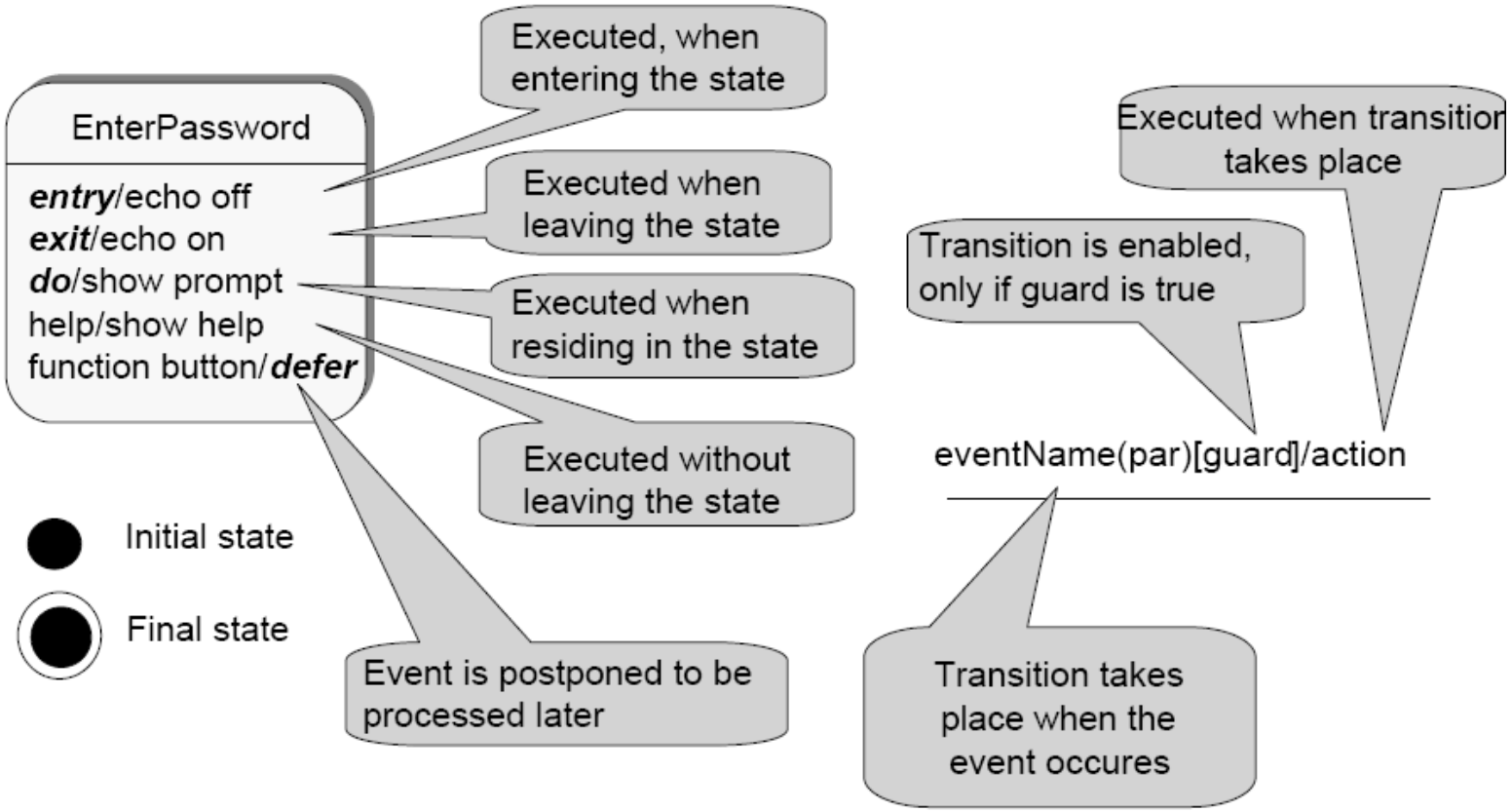
Loop conditions



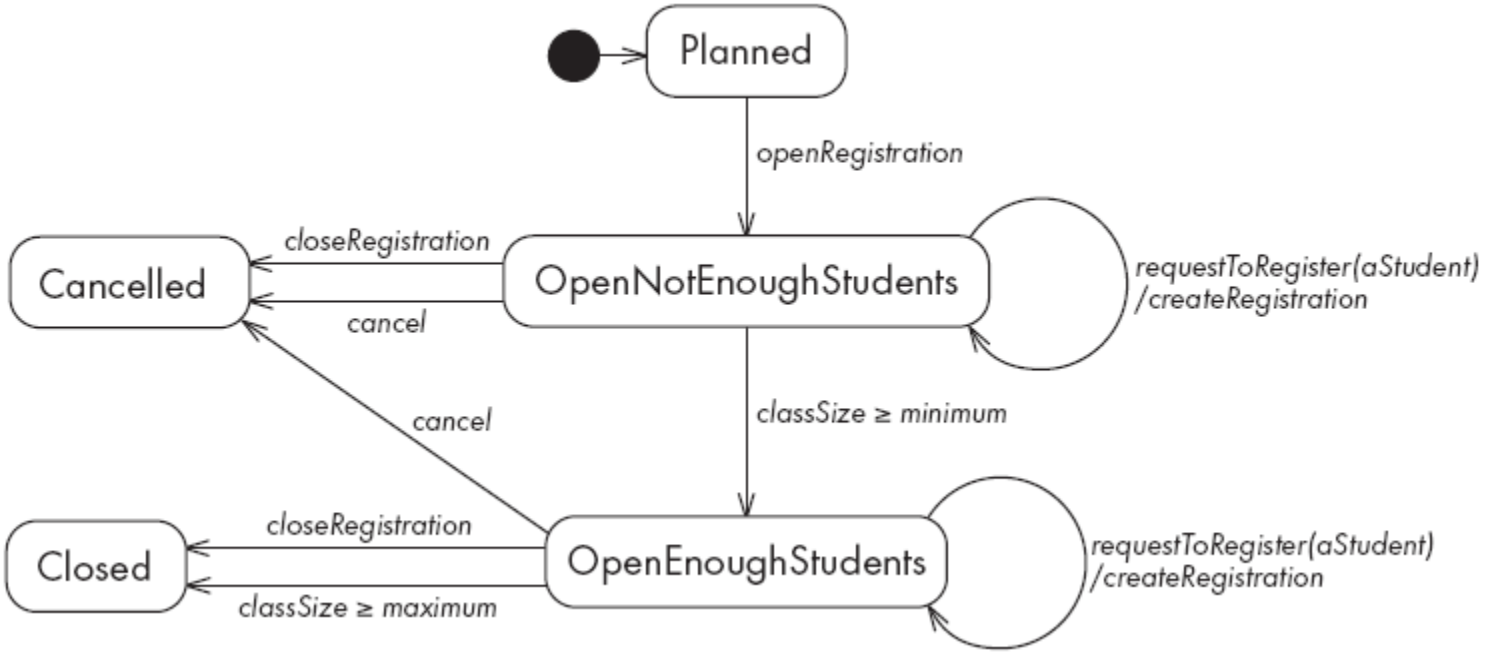


Behavioral models: State Chart Notation

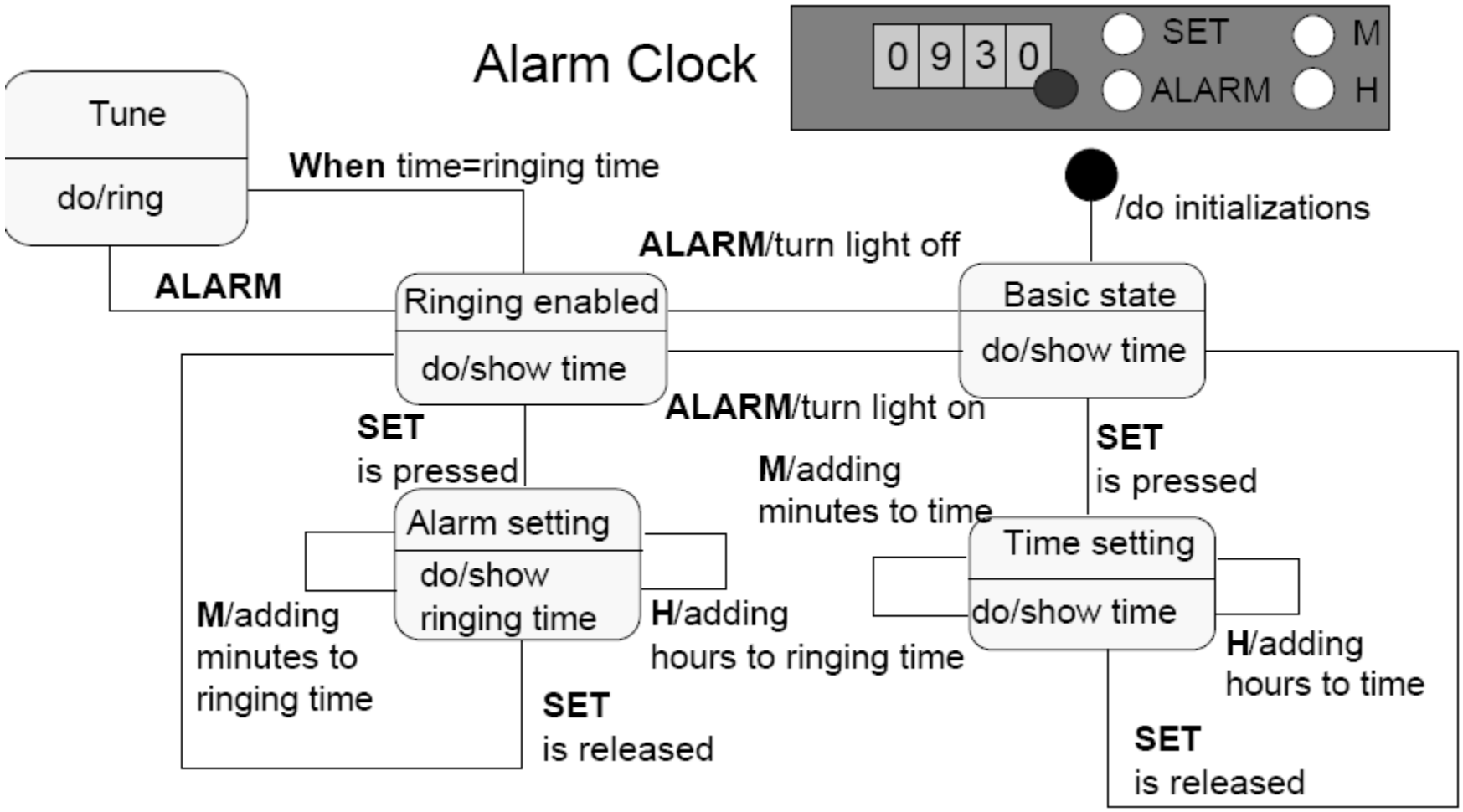
- State transition diagram shows
  - The events that cause a transition from one state to another
  - The actions that result from a state change



State diagram of CourseSection Class

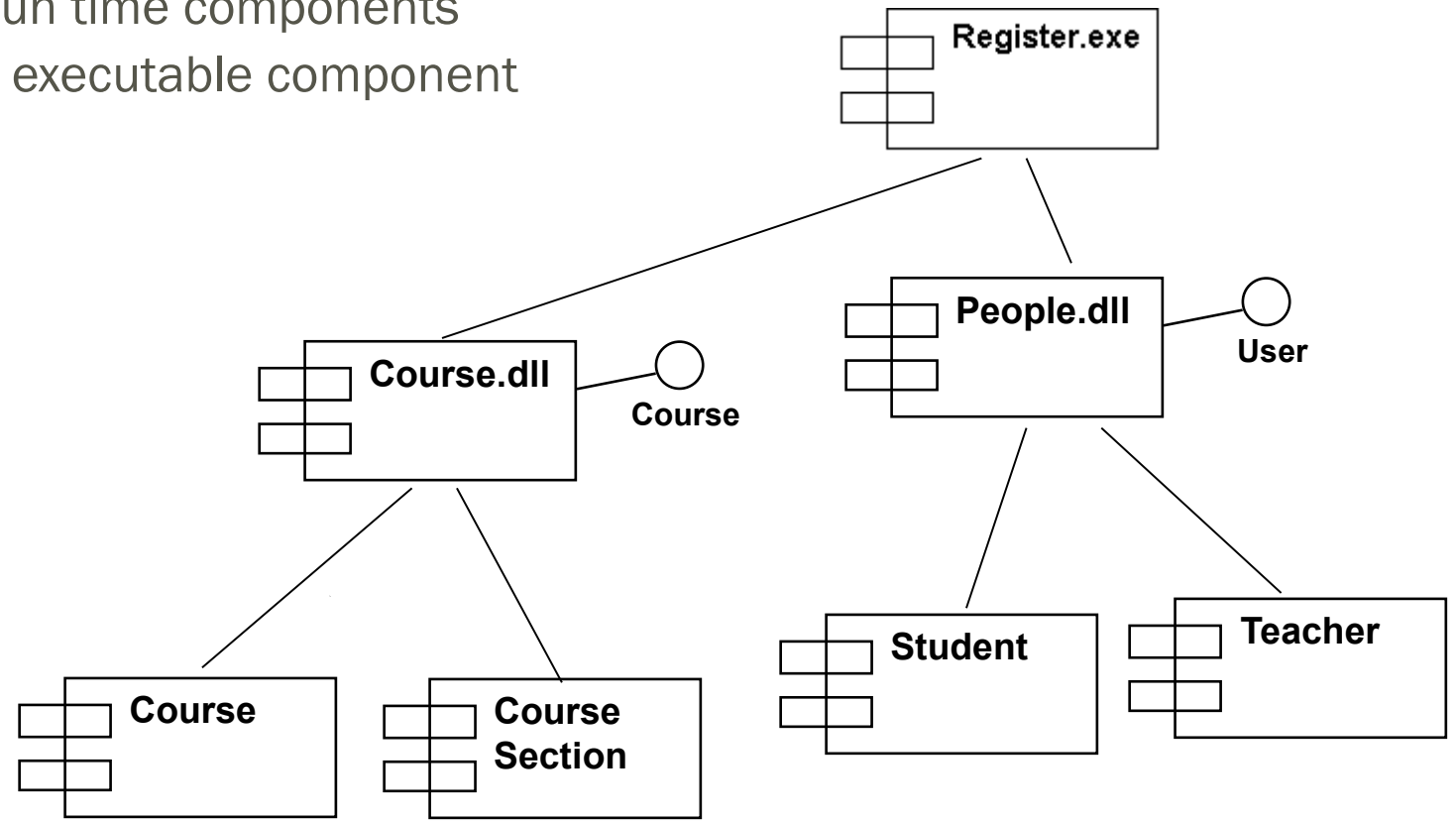


Example : State Chart Diagram



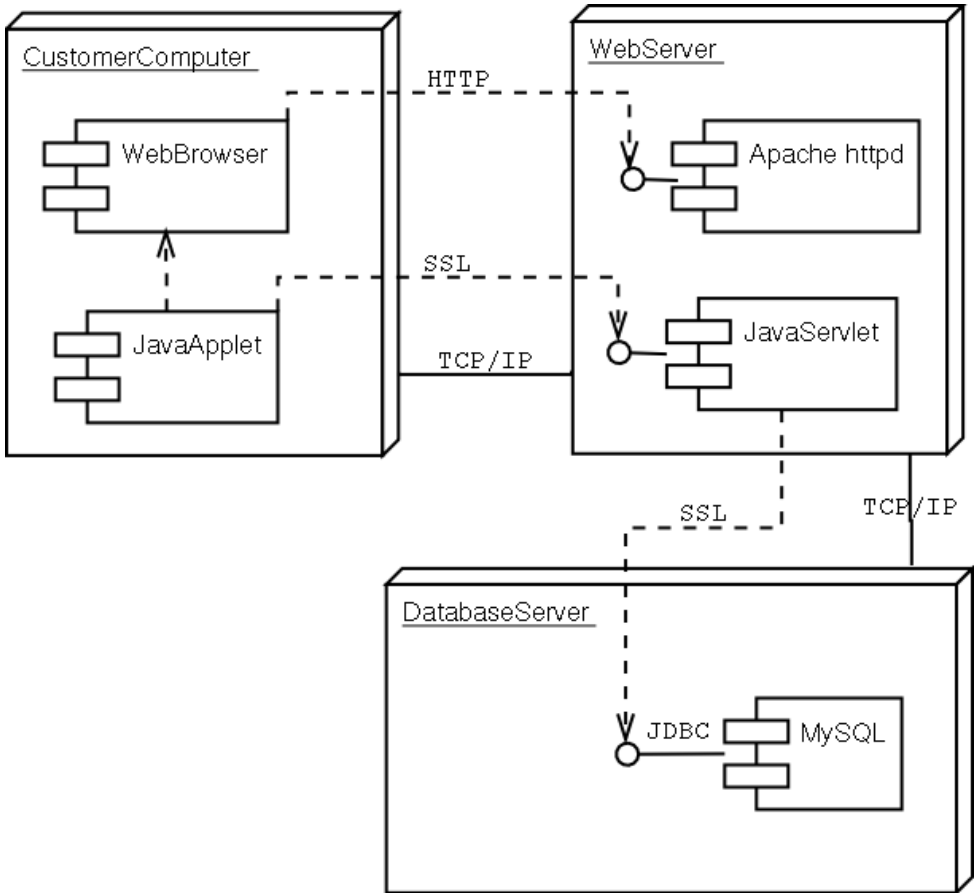
Package Diagrams

- Component diagrams illustrate the organizations and dependencies among software components in physical world.
- A component may be
  - A source code component
  - A run time components
  - An executable component

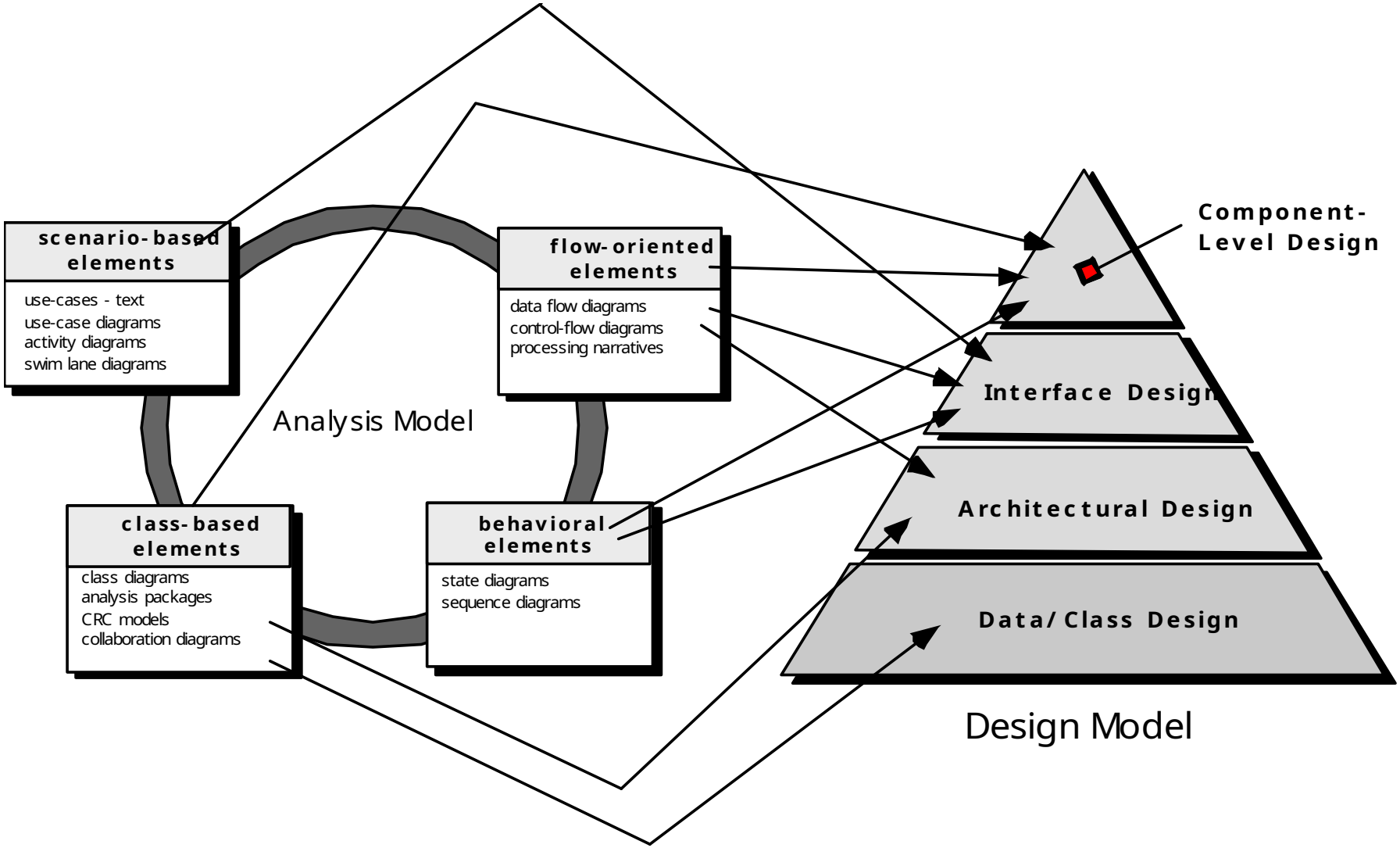


Deployment Diagrams

- Captures the distinct number of computers involved
- Shows the communication modes employed
- Component diagrams can be embedded into deployment diagrams effectively



Analysis Model ⟶ Design Model



1. Software Design Concepts
2. Structured Design
   2.1. Case Study: SafeHome
3. Object Oriented Design Principles
   3.1. Unified Modeling Language
   3.2. Case Study: Elevator
   3.3. Case Study: Manufacturing Plant
4. User Interface Design

Case Study: Elevator

8.3.2

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

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

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 Button Class**

First Iteration of Class Diagram

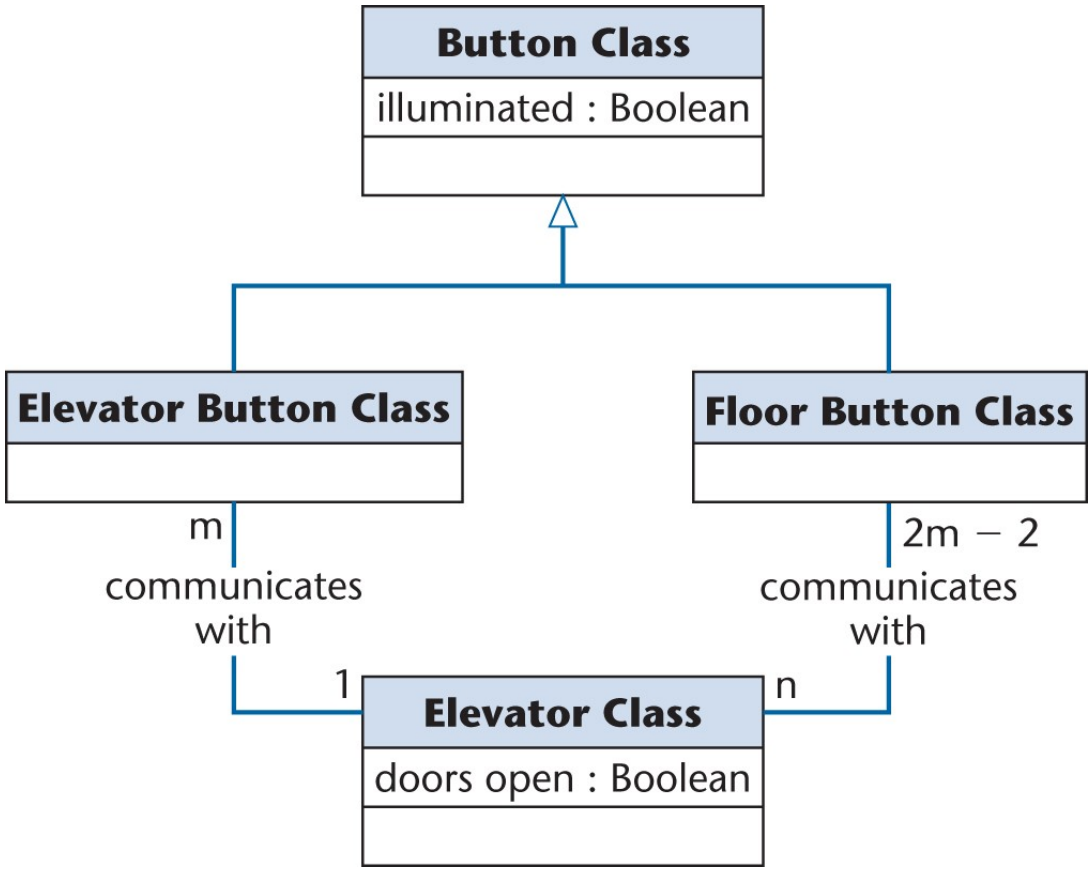


Figure 13.5

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

Second Iteration of Class Diagram

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

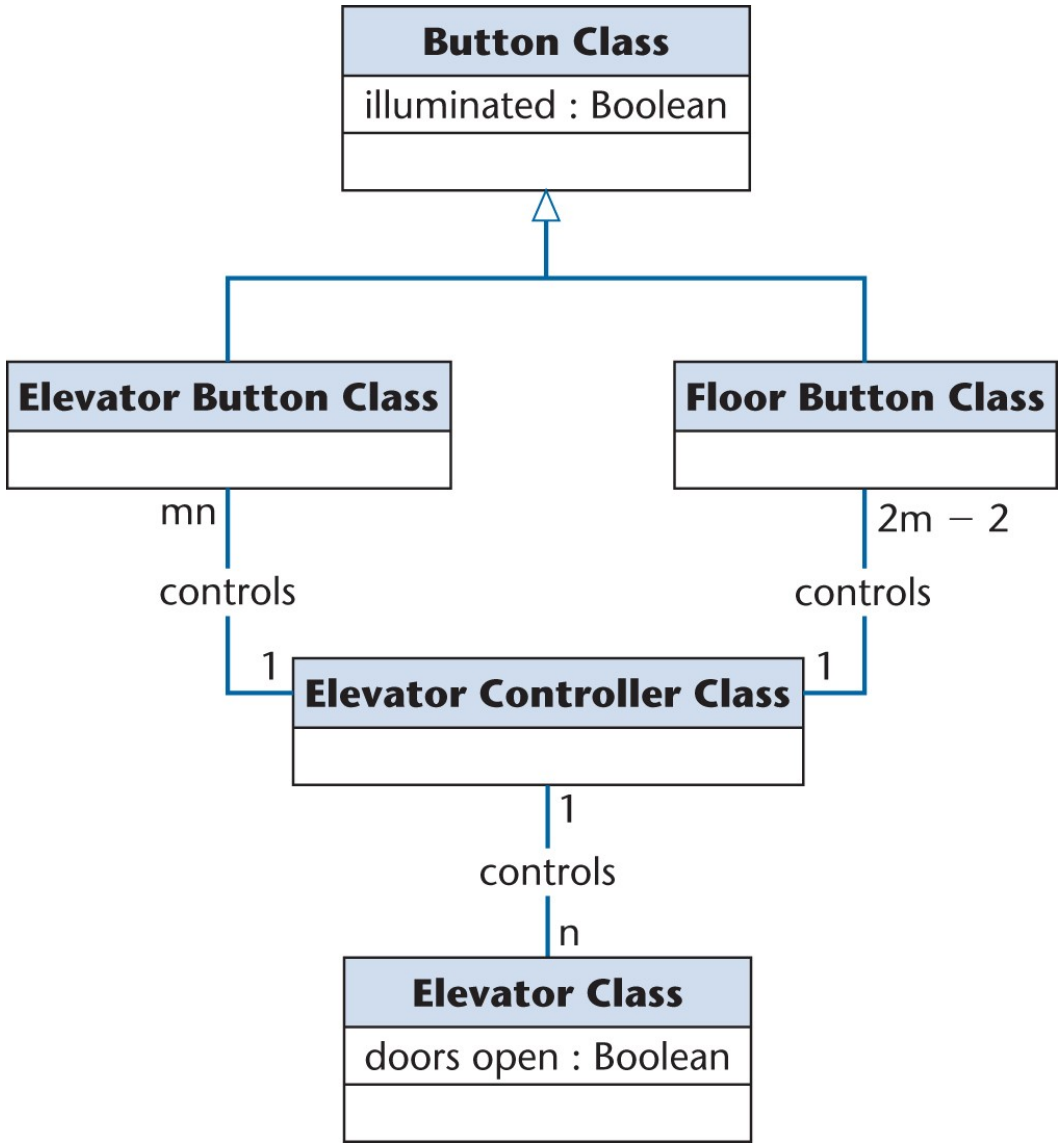


Figure 13.6

Third Iteration of Class Diagram

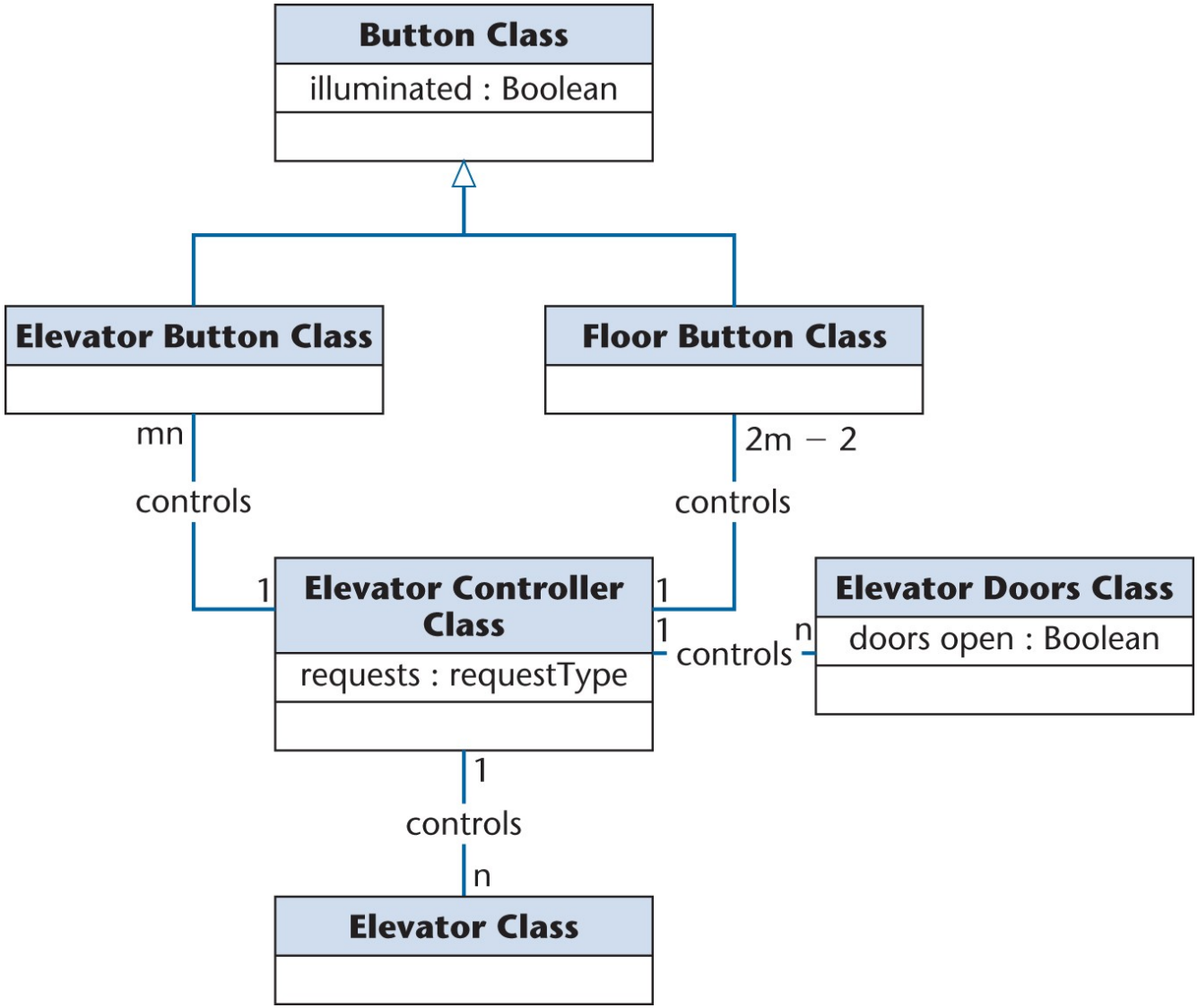


Figure 13.10

Fourth Iteration of Class Diagram

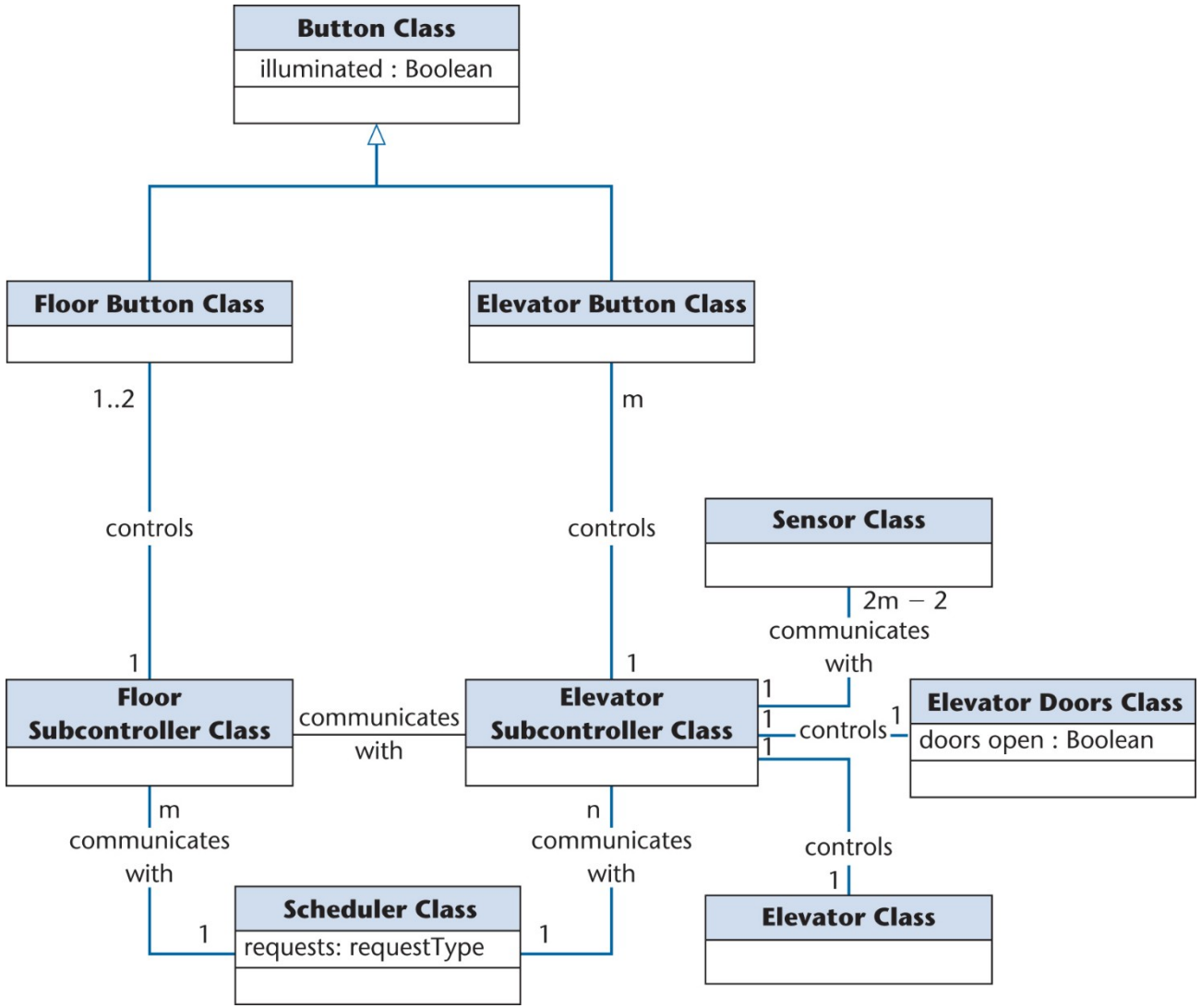


Figure 13.12

Detailed Class Diagram: Elevator Problem

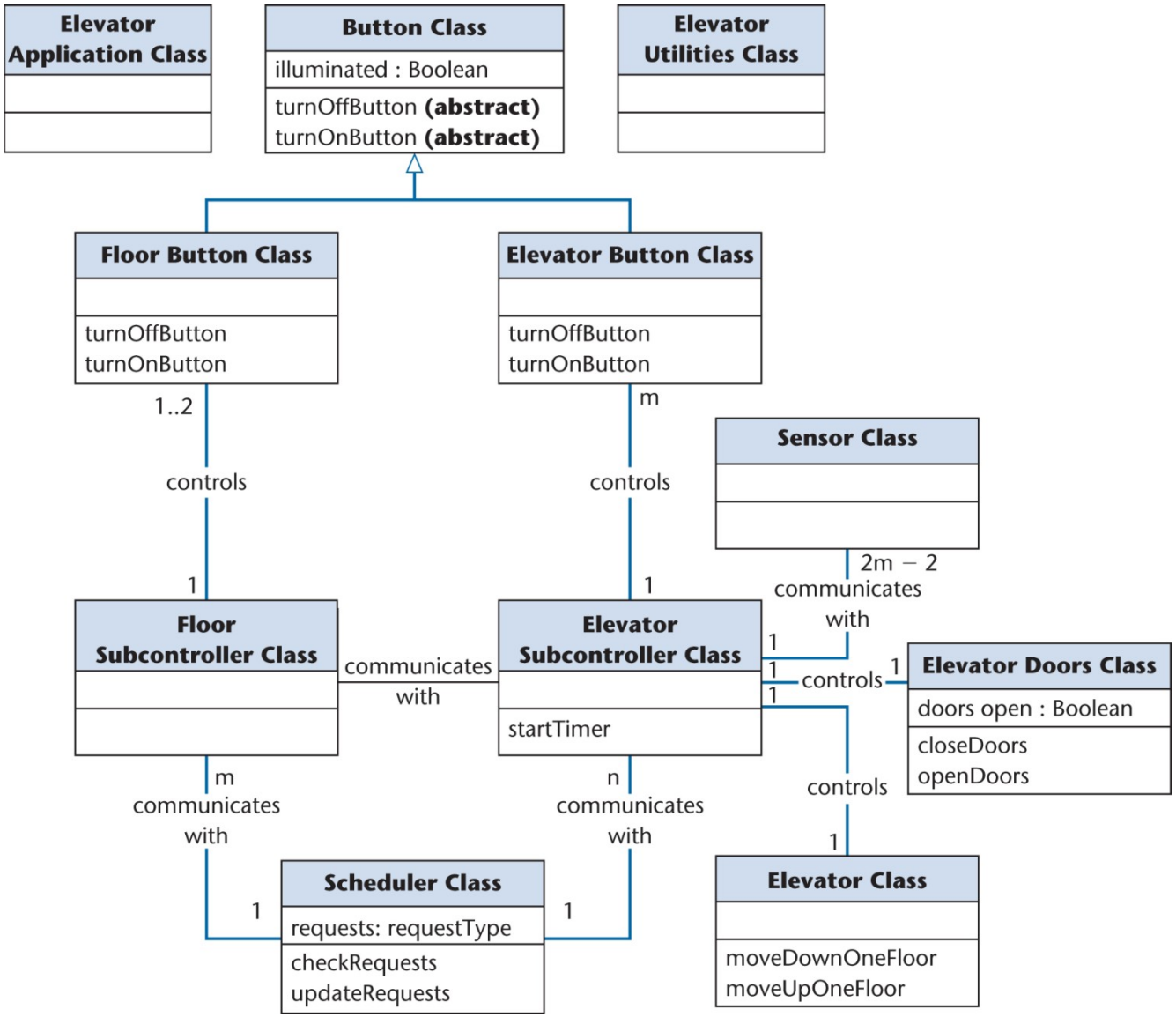


Figure 14.11

Dynamic Modeling: The Elevator Problem Case Study

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

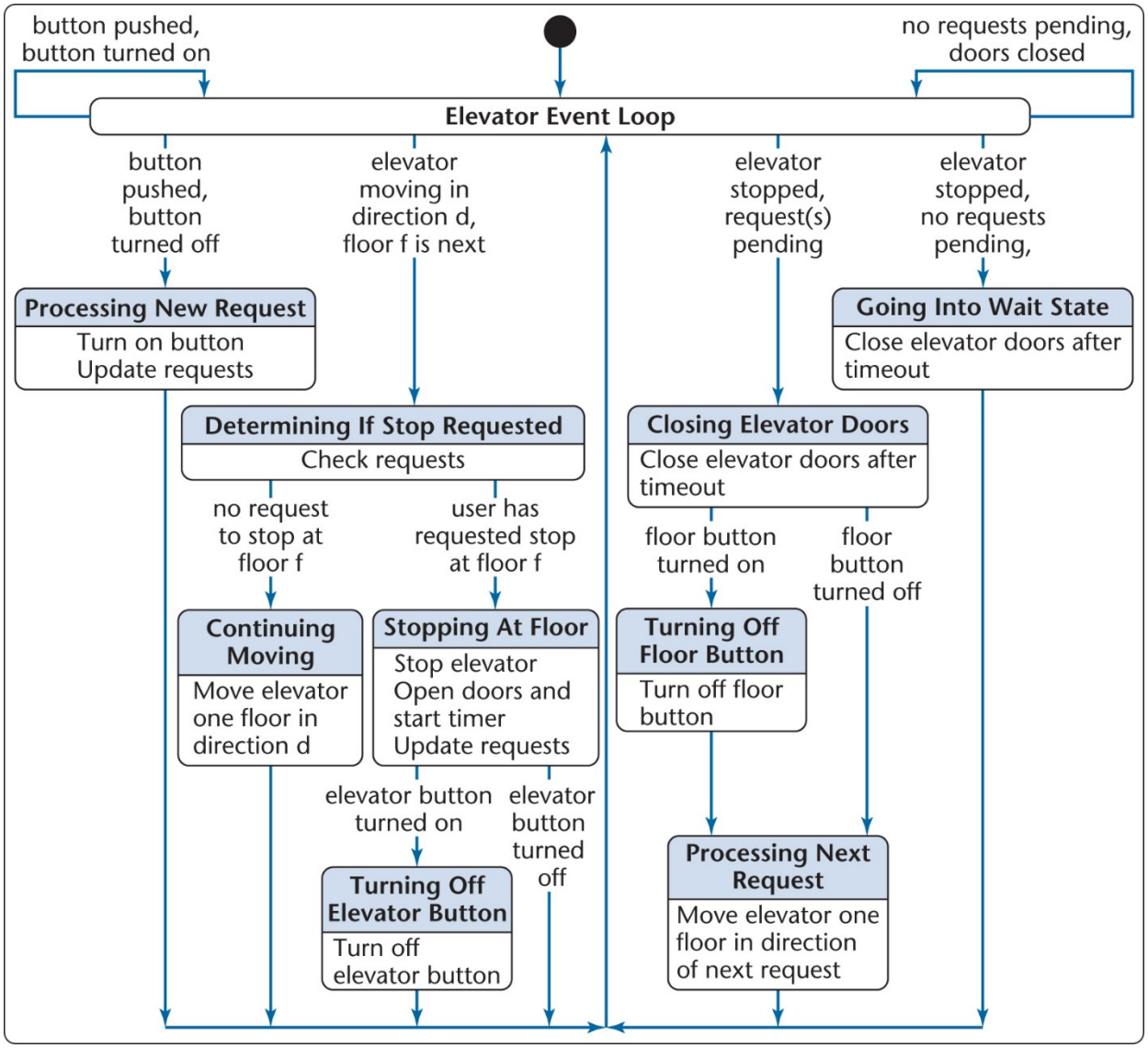


Figure 13.7

Detailed Design: Elevator Problem

- Detailed code 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

Case Study: Manufacturing Plant

8.3.3

Manufacturing Plant's Production Process

- This system will be used to manage and control the production processes at Use Case Industries' manufacturing plant. The plant makes several types of mechanical devices. It has 10 assembly lines, each of which can be used in the manufacturing of any of its products. An assembly line is allocated to a Product for a fixed period of time (anywhere from a few hours to a few days) – this is called a product run. During a product run, th s of the product.

Lets find the classes.

- Each product is assembled in several steps. As the product-under construction moves down the assembly line, it will be worked on in turn by a series of robots. Each robot completes one step before the product moves on to the next step (and a different robot). Each robot is dedicated to just one manufacturing step.
- Each product is composed of parts. Parts may be bought from suppliers, or they may in fact be smaller products that are built by this company (in earlier product runs). In each manufacturing step, a given subset of these parts is put together. Parts waiting for assembly are kept in numbered bins; the robots know which bins to go to in order to get the required parts.
- Each completed assembly is given a serial number. When orders for products are filled, the serial numbers of the products sold are recorded with the order.

Manufacturing Plant's Production Process

- This system will be used to manage and control the production processes at Use Case Industries' manufacturing plant. The plant makes several types of mechanical devices. It has 10 assembly lines, each of which can be used in the manufacturing of any of its products. An assembly line is allocated to a Product for a fixed period of time (anywhere from a few hours to a few days) – this is called a product run. During a product run, the assembly line makes a specified number of units of the product.
- Each product is assembled in several steps. As the product-under construction moves down the assembly line, it will be worked on in turn by a series of robots. Each robot completes one step before the product moves on to the next step (and a different robot). Each robot is dedicated to just one manufacturing step.
- Each product is composed of parts. Parts may be bought from suppliers, or they may in fact be smaller products that are built by this company (in earlier product runs). In each manufacturing step, a given subset of these parts is put together. Parts waiting for assembly are kept in numbered bins; the robots know which bins to go to in order to get the required parts.
- Each completed assembly is given a serial number. When orders for products are filled, the serial numbers of the products sold are recorded with the order.

Classes

- This system will be used to manage and control the production processes at Use Case Industries' manufacturing plant. The plant makes several types of mechanical devices. It has 10 assembly lines, each of which can be used in the manufacturing of any of its products. An assembly line is allocated to a Product for a fixed period of time (anywhere from a few hours to a few days) – this is called a product run. During a product run, the assembly line makes a specified number of units of the product.
- Each product is assembled in several steps. As the product-under construction moves down the assembly line, it will be worked on in turn by a series of robots. Each robot completes one step before the product moves on to the next step (and a different robot). Each robot is dedicated to just one manufacturing step.
- Each product is composed of parts. Parts may be bought from suppliers, or they may in fact be smaller products that are built by this company (in earlier product runs). In each manufacturing step, a given subset of these parts is put together. Parts waiting for assembly are kept in numbered bins; the robots know which bins to go to in order to get the required parts.
- Each completed assembly is given a serial number. When orders for products are filled, the serial numbers of the products sold are recorded with the order.

Revision on Classes

- Production Process
- Manufacturing Plant ?
- Mechanical Devices →Product?
- Assembly Line
- Product → **ProductType** (description)
- Product Run
- Product Unit → **Product** (serial number)
- Step
- Robot
- Part?
- Supplier
- Bin
- Assembly?
- Order

https://cruise.umple.org/umpleonline/

Select from Examples (Manufacturing Plan)

1. Software Design Concepts
2. Structured Design
   2.1. Case Study: SafeHome
3. Object Oriented Design Principles
   3.1. Unified Modeling Language
   3.2. Case Study: Elevator
   3.3. Case Study: Manufacturing Plant
4. User Interface Design

User Interface Design

§ 8.4 §

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.

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.

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).

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*.

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.

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

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.

GUI Elements and Principles

Elements

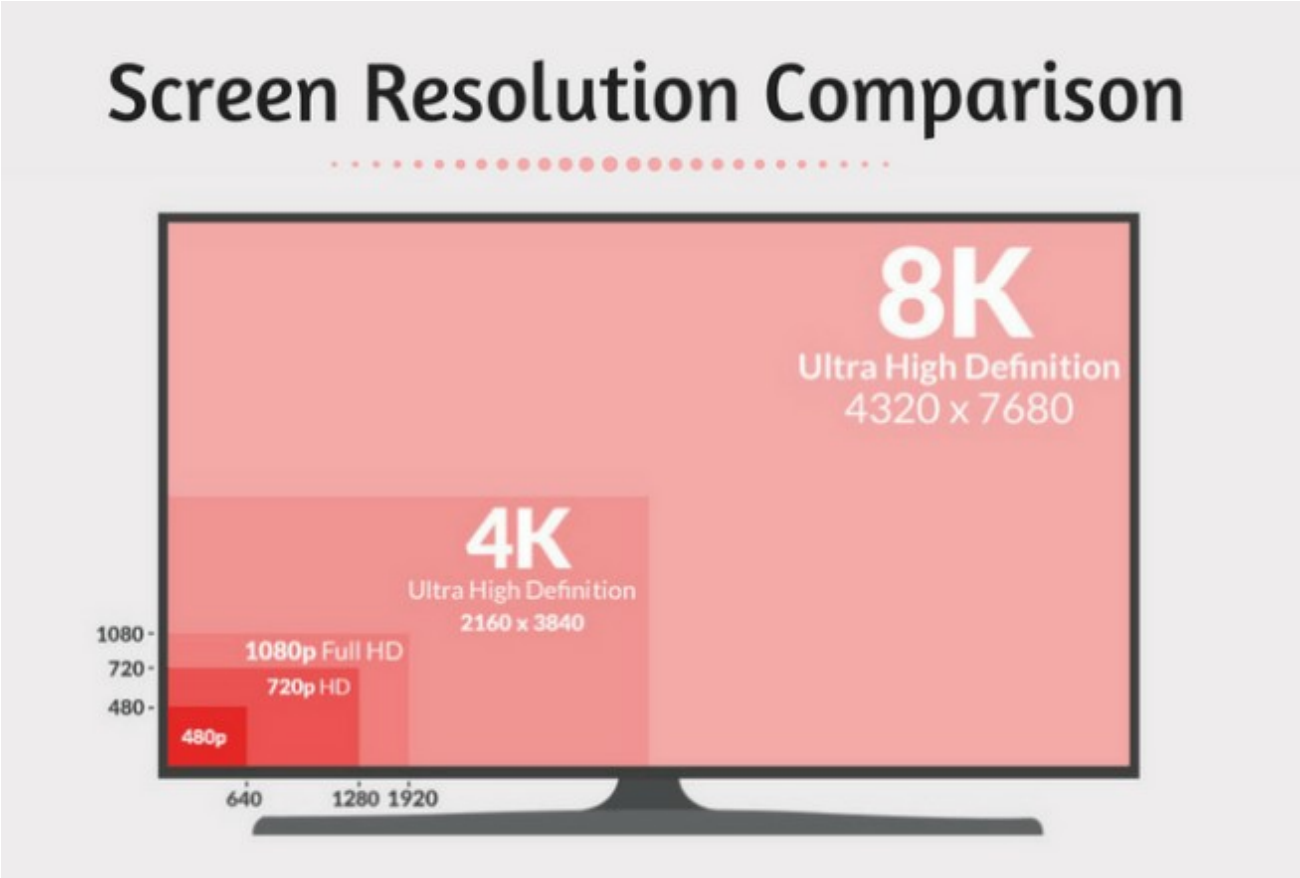
- Icons
- Windows
- Menus
- Pointers
- Text
- Color
- Graphics
- Audio
- Animation
- Video

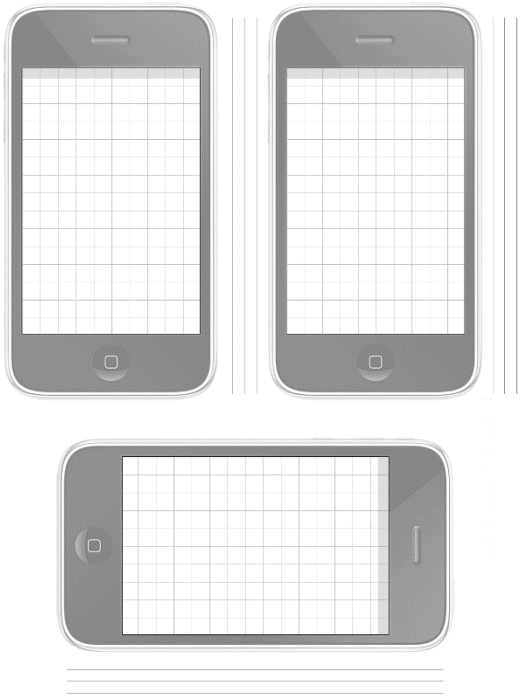
Principles

- Consistency
- Clarity
- Predictability
- Economy
- Transparency
- Modality
- Sensitivity
- Understandability

Product

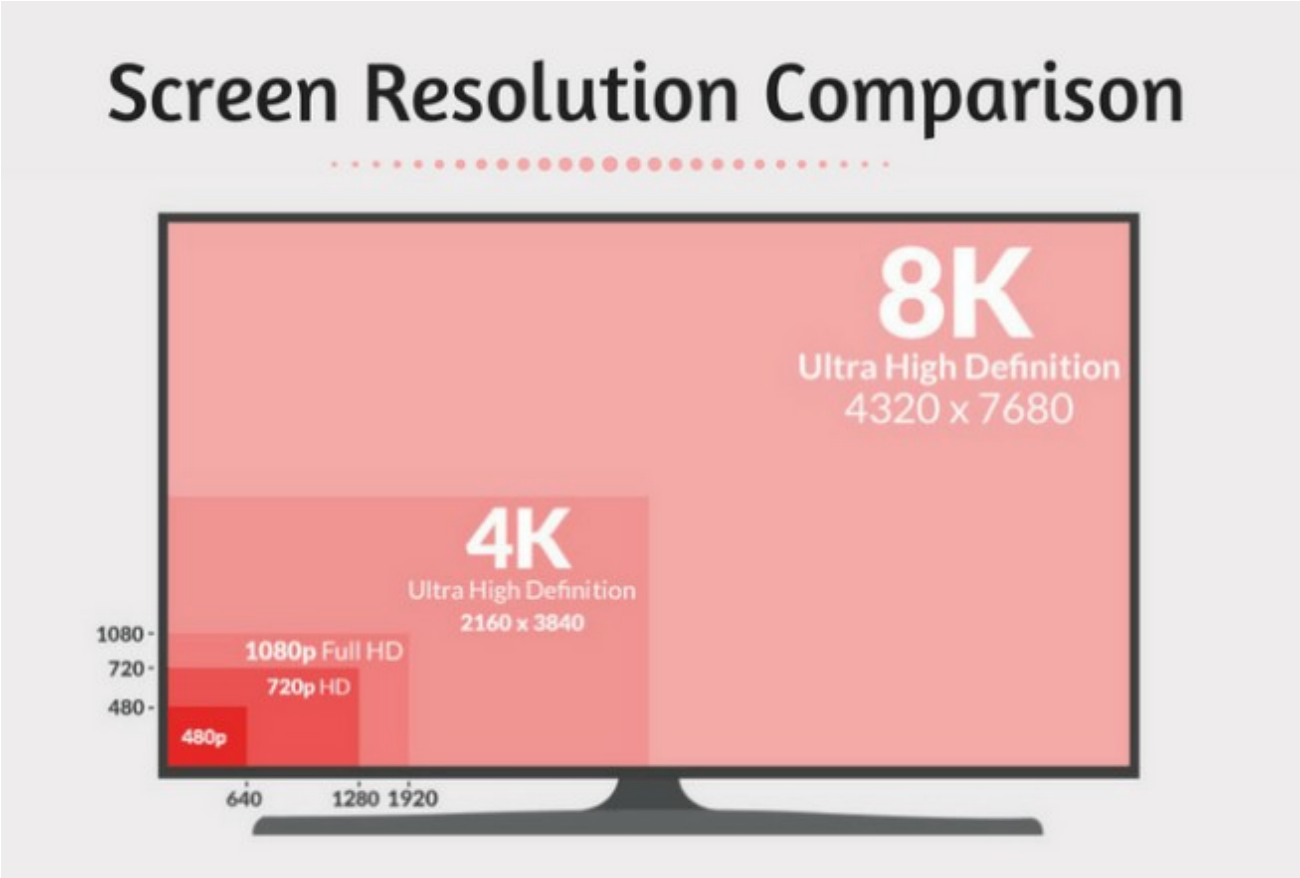
Screen Aspect Ratio & Orientation

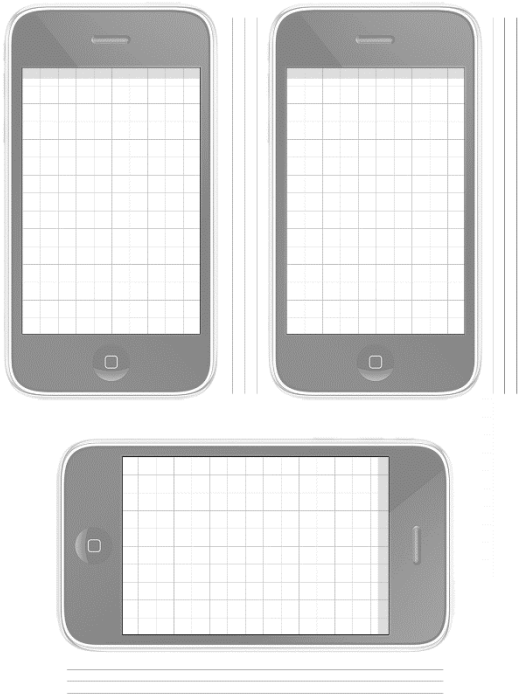




The web and mobile devices make this challenging

Screen Aspect Ratio & Orientation





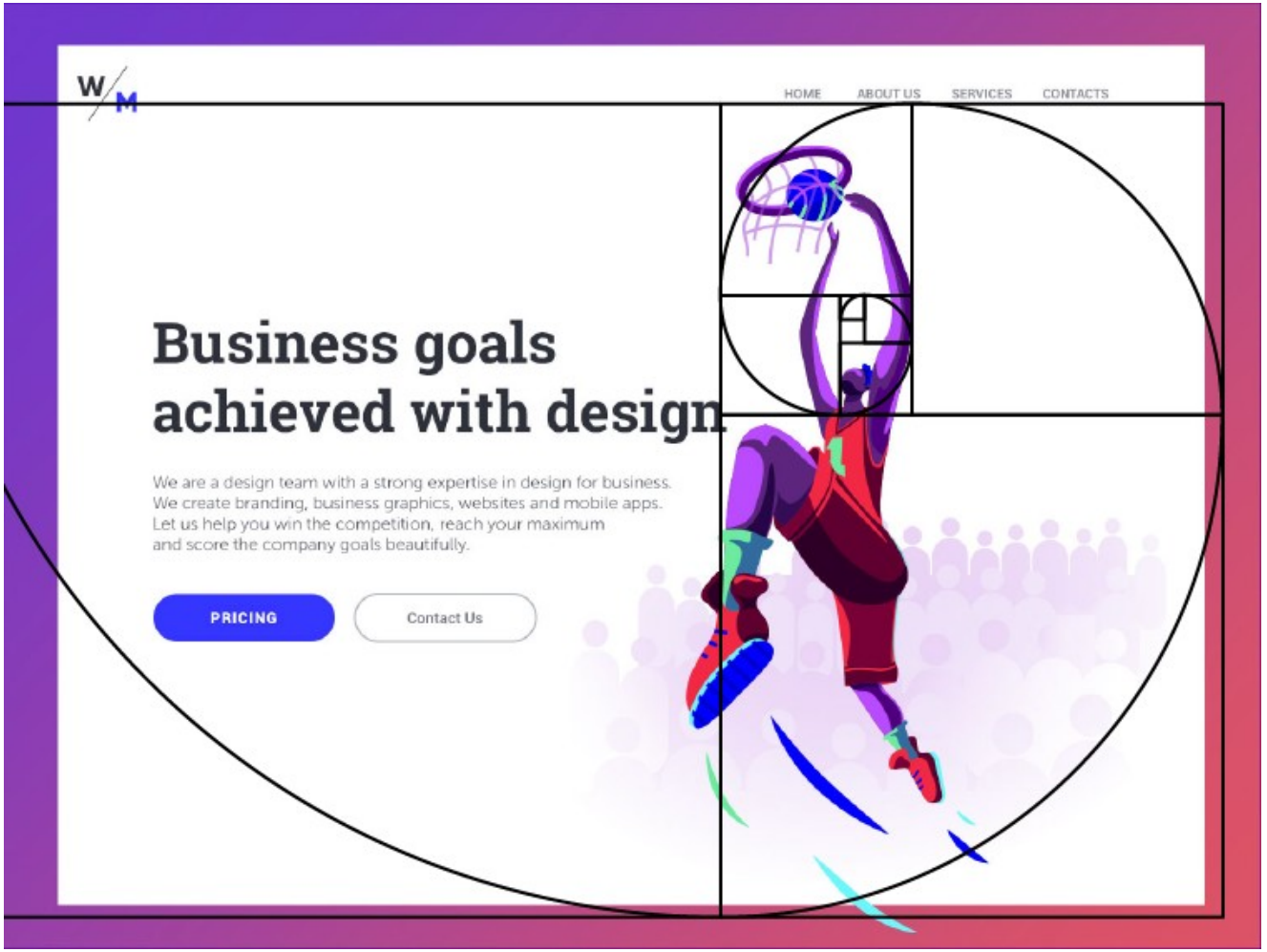
The web and mobile devices make this challenging

Golden Rectangle

The Fibonacci Series yields a ratio of 1:1.61 (Phi) which is found throughout natural forms.

The rectangle based on that ratio is called the Golden Rectangle.

Software tools now help designers use the Golden Mean.

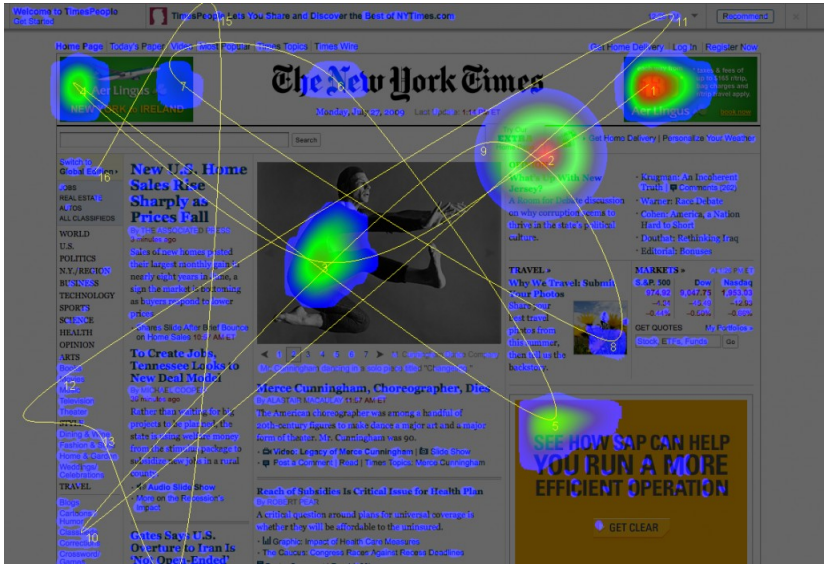


The Rule of Thirds

The guideline proposes that an image should be imagined as divided into nine equal parts by two equally spaced horizontal lines and two equally spaced vertical lines, and that important compositional elements should be placed along these lines or their intersections.

Capturing the User's Attention

- Where the eye is drawn:
  - Moving pictures (video, animation)
  - Pictures (photos, graphics, illustrations)
  - Headlines (large, bold, or differently-colored text)
  - Body copy
- Visual prominence accrues to:
  - Larger regions of positive and negative space
  - Regions with greater contrast
  - Higher saturations of color
  - Warmer colors

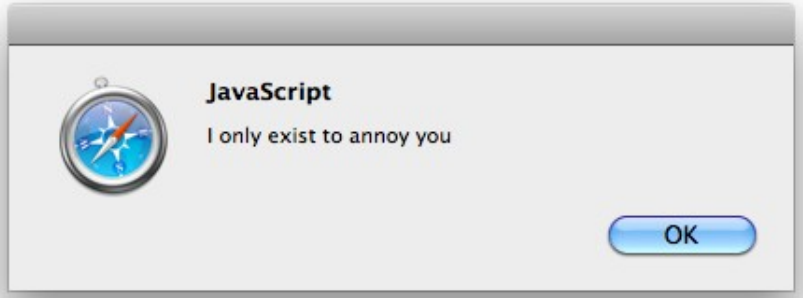


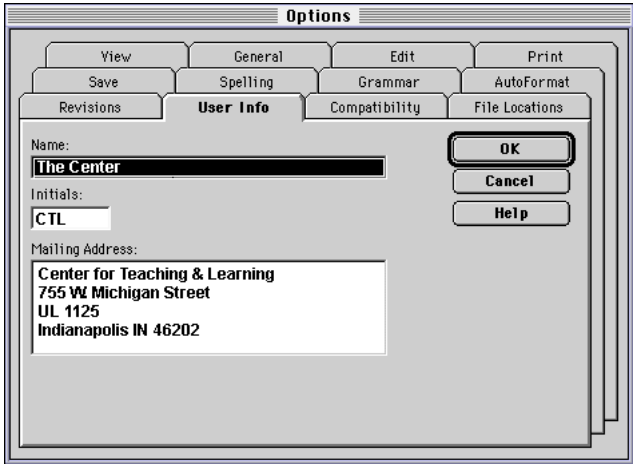
An eye-tracking heat map of the NY Times online indicates where a user's gaze has been directed.

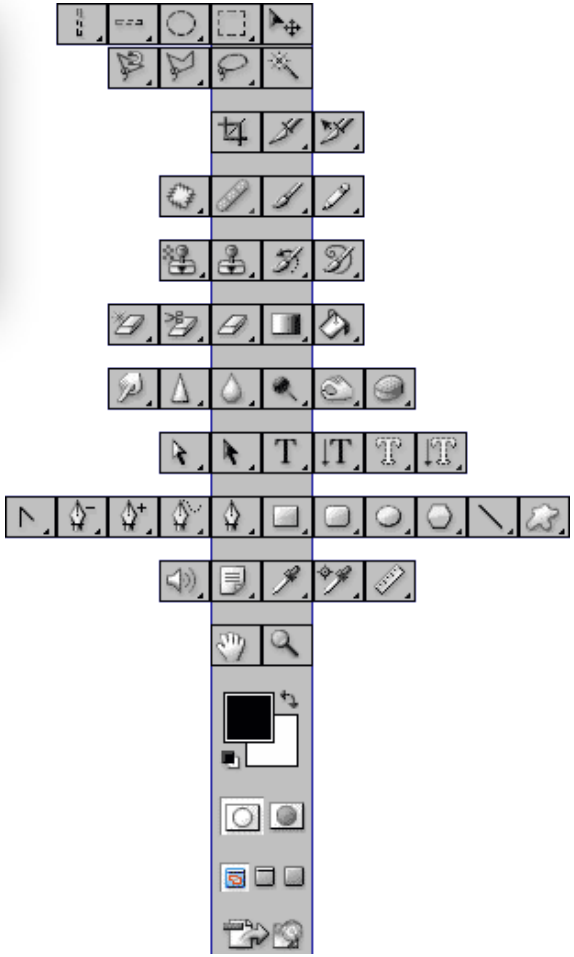
Avoiding Clutter

With dialogs, alerts, and tool bars, avoid complexity and too many choices. Tools should be semantically organized.

Use dialogs and alerts sparingly. Dialogs and alerts can interrupt the flow of interaction and make users lose their place.







Color Guidelines

- Use a limited palette of complementary colors
- Make sure your background *is* the background
- Don’t use light blue for text, but it’s great for backgrounds
- Older users need brighter colors to recognize them, and 10% of males have some form of color blindness
- Don’t use highly saturated colors from opposite ends of the spectrum in close proximity

Use Complementary Colors

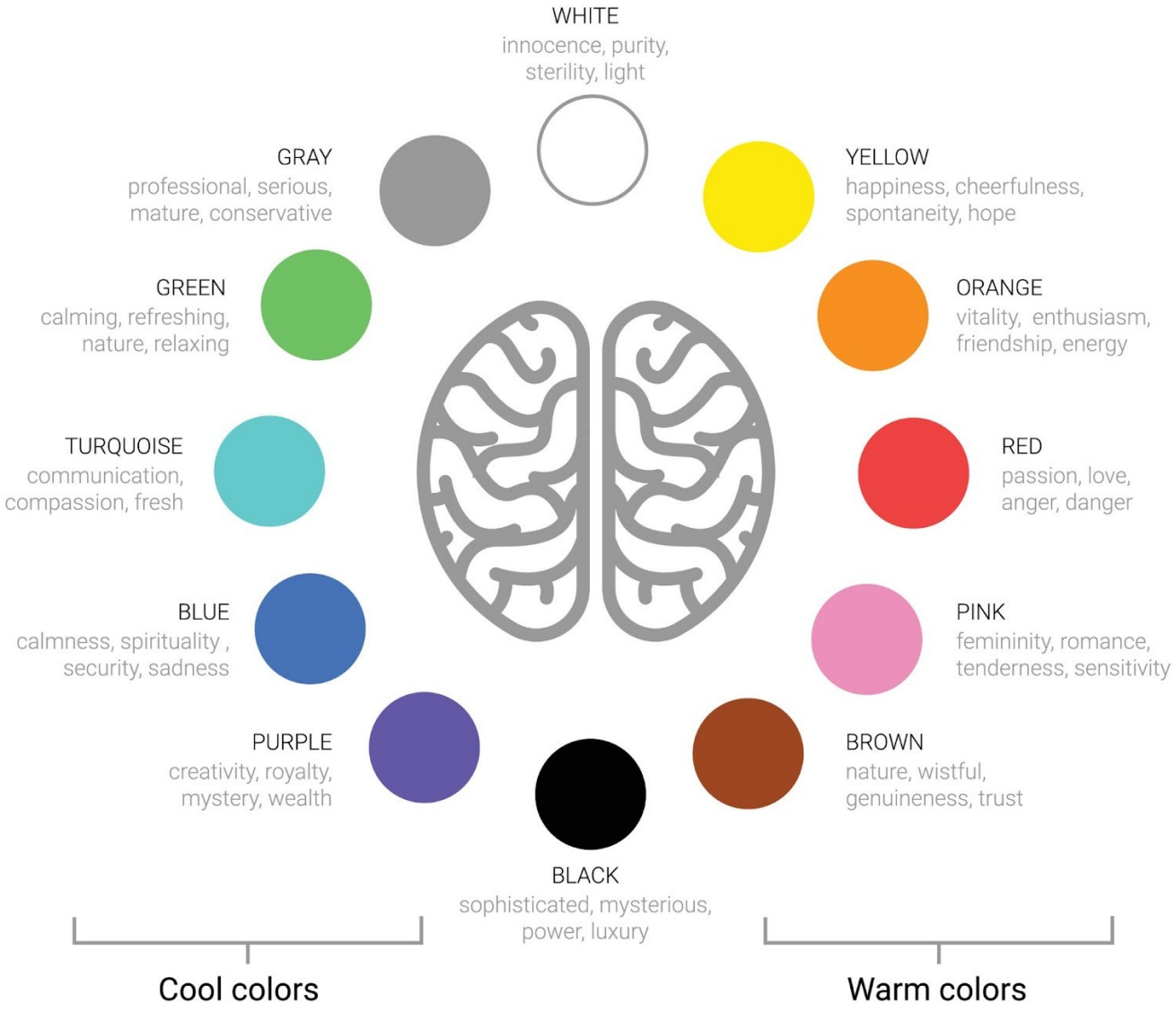
Warm colors move forward

Cool backgrounds recede

This is very hard to read.

Don’t Do This!

Color Psychology



Typography

- Limit the use of styles to at most four: headlines, subheads, body copy, and captions.
- The use of **bold** and *italic* fonts should be consistent and limited.
- Choose fonts that support the theme of the interface.
  - Serif fonts (e.g. **Times**) are classical and formal. They are more legible than sans-serif fonts and are good for body copy.
  - Sans-Serif fonts (e.g. **Arial**) are modern and informal. Usually best for headlines.
- Specialty fonts and typographic designs (e.g. logos) might have to be rasterized for distribution. Some applications, such as Flash, allow developers to embed fonts in their executable files (.swf).

Above, the “core nine” cross-platform (web-safe) fonts.

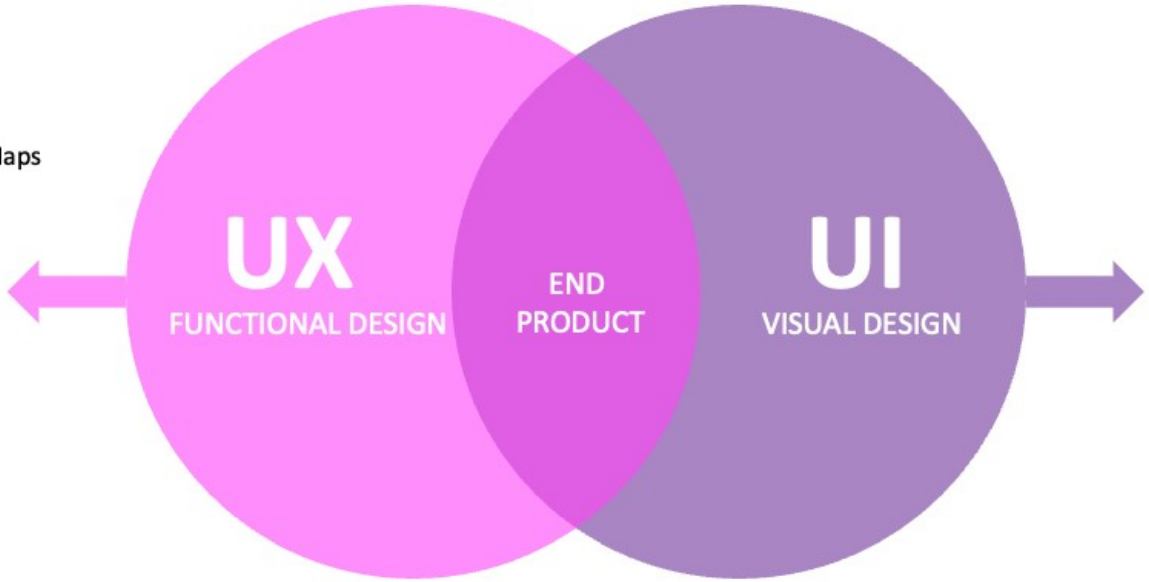
User eXperience vs User Interface

USER RESEARCH
- Primary Research
- Competitor Research

RESEARCH ANALYSIS
- Consumer Insights
- Personas
- Customer Journey Maps
- Empathy Maps
- Site Map
- Task Flows
- User Flows

DESIGN IDEATION
- Concepting
- Wireframes
- Lo-fi Prototypes
- User Testing
- Design Iterations

EXECUTION
- Stakeholder Co-ordination
- UI Design Collaboration



INTERFACE DESIGN
- Wireframe Knowledge Transfer
- Look and Feel
- Branding and Design Systems
- Layout and Responsiveness
- User Testing

DESIGN SPECIFICATIONS
- UI Hi-fi Prototypes
- Visual Design Documentation
- Developers Final Design Handout
- Graphic Elements Set (Icons)
- Adaptation to Form Factors

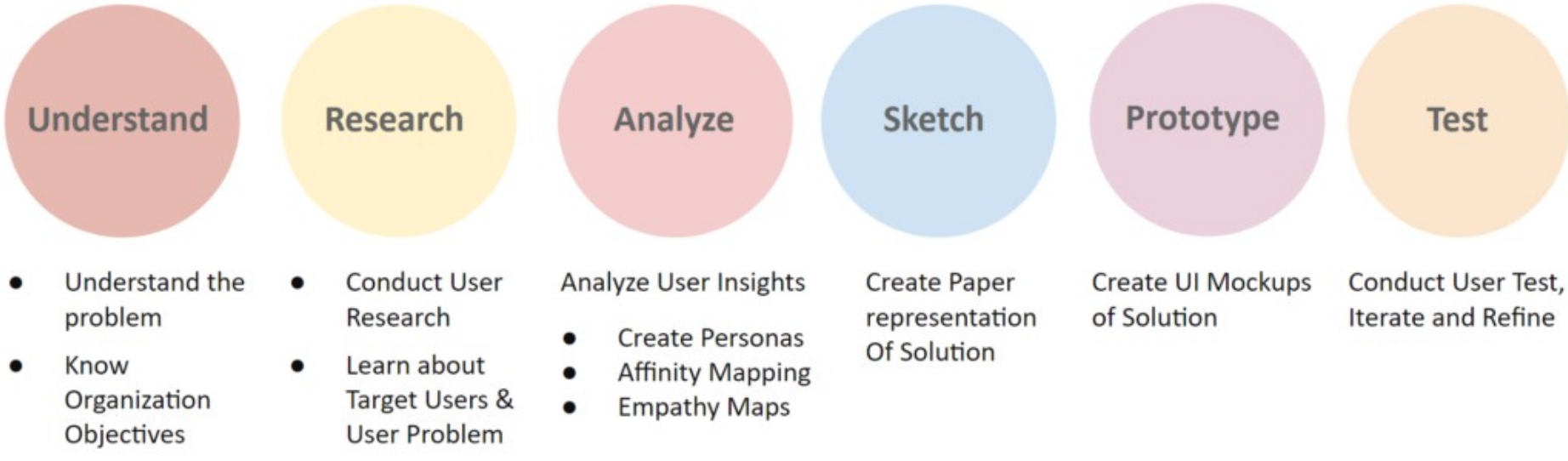
EXECUTION
- UX Design Collaboration
- Developer Collaboration
- Implementation Reviews
- Stakeholder Coordination

User eXperience (UX)

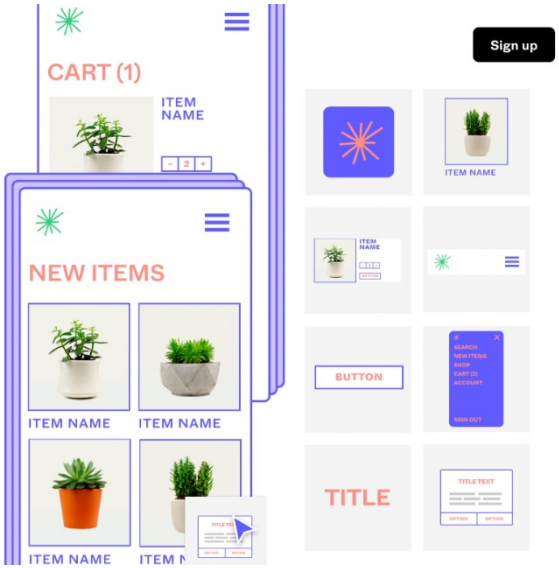
- The User eXperience (UX) relates to how a user feels whenever they interact with a product or service. It's not a physical, tangible thing - it's the ease and user-friendliness of the interaction as a whole.

- The User Interface (UI) relates specifically to the screens, buttons and other visual and interactive features a person uses to interact with a digital product, such as a website or app.

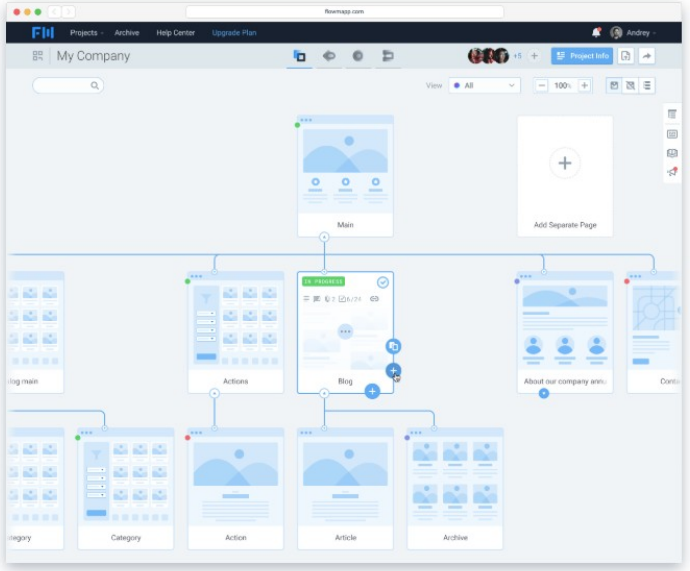
UX Design Process



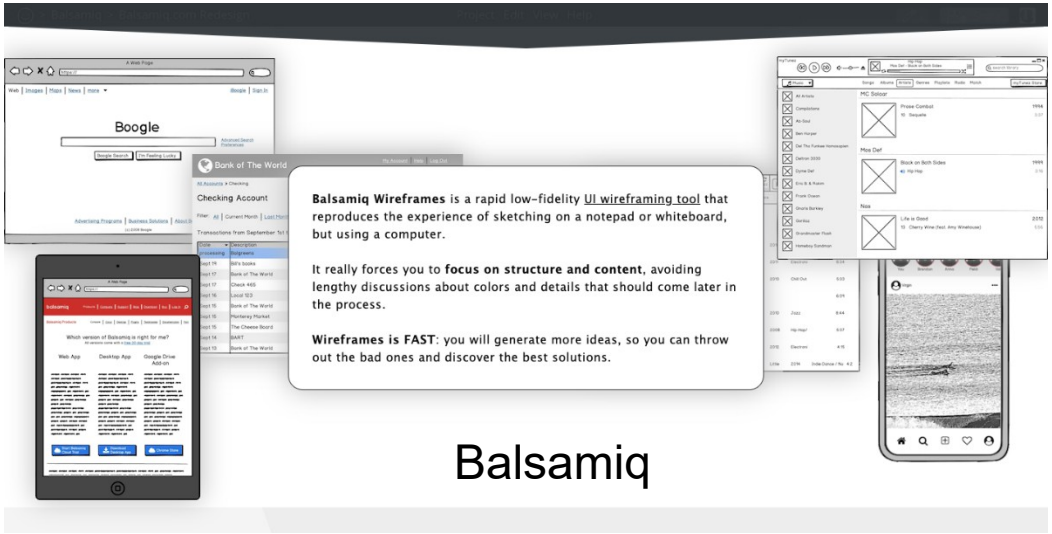
UI/UX Design Tools



Figma



WebFlow



Balsamiq

many more >

User Interface Patterns

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

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

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!!!*