UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4351 Fall 2023

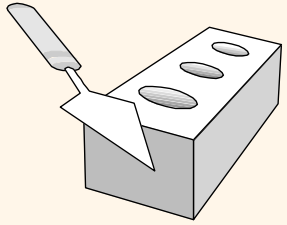
Software Engineering

M & W 4 to 5:30 PM

Prof. Victoria Hilford

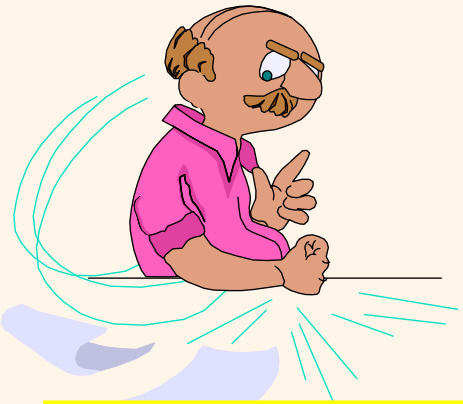
PLEASE TURN your webcam ON

NO CHATTING during LECTURE



COSC 4351

4 to 5:30



PLEASE

LOG IN

CANVAS

Youyi [A-L]

Kevin [M-Z]

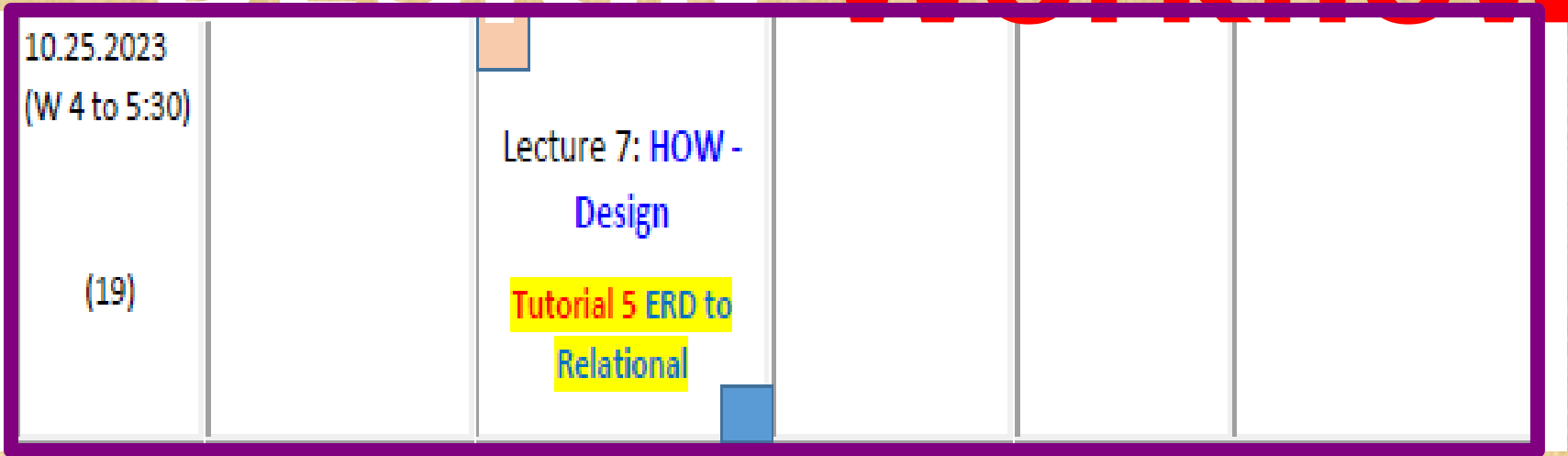
Please close all other windows.

10.25.2023 (W 4 to 5:30) (19)		Lecture 7: HOW - Design Tutorial 5 ERD to Relational			
10.30.2023 (M 4 to 5:30) (20)		EXAM 3 REVIEW (CANVAS)	Download ZyBook: Sections 10-11		
11.01.2023 (W 4 to 5:30) Optional (21)					Q & A Set 3 topics.
11.06.2023 (M 4 to 5:30) (22)					EXAM 3 (CANVAS)

Class 19

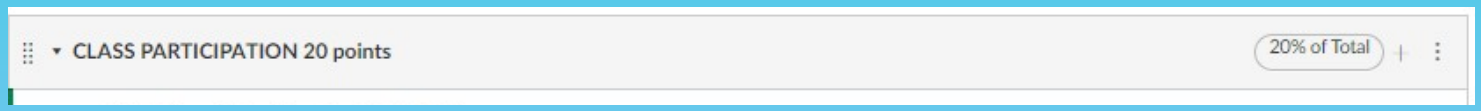
Lecture 7 on

DESIGN Workflow

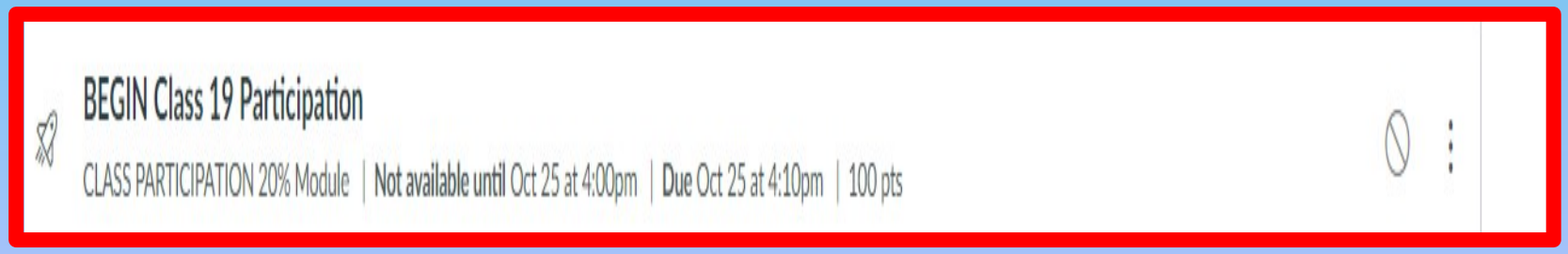


From 4:00 to 4:10 – 10 minutes.

10.25.2023 (W 4 to 5:30) (19)		Lecture 7: HOW - Design Tutorial 5 ERD to Relational			



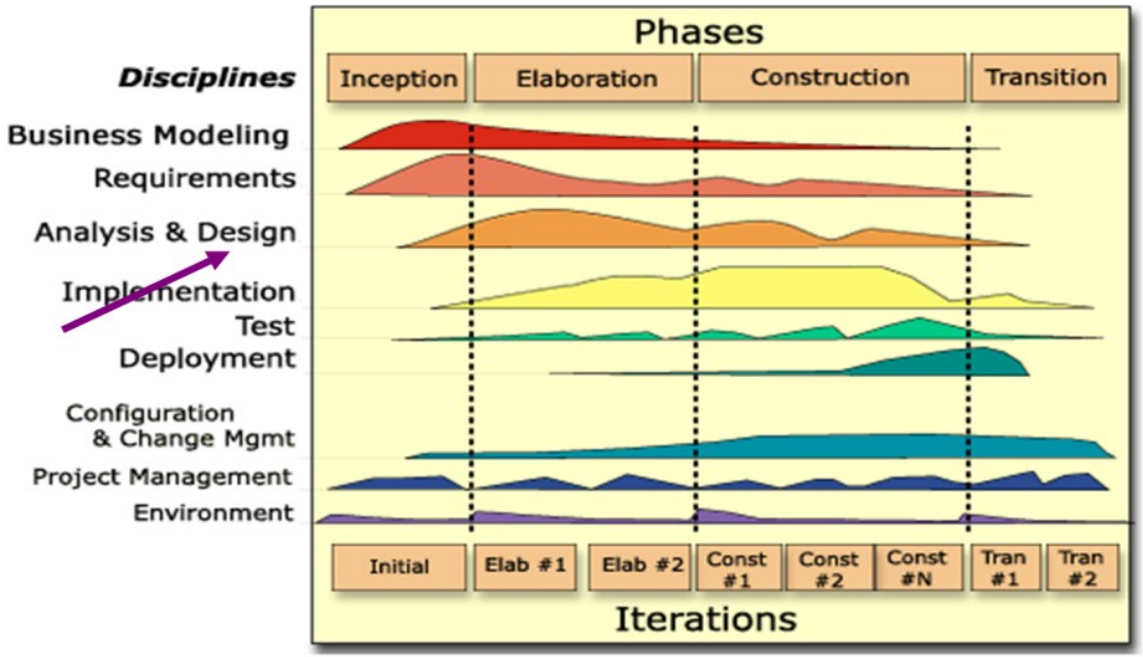
PASSWORD: IN TEAMS



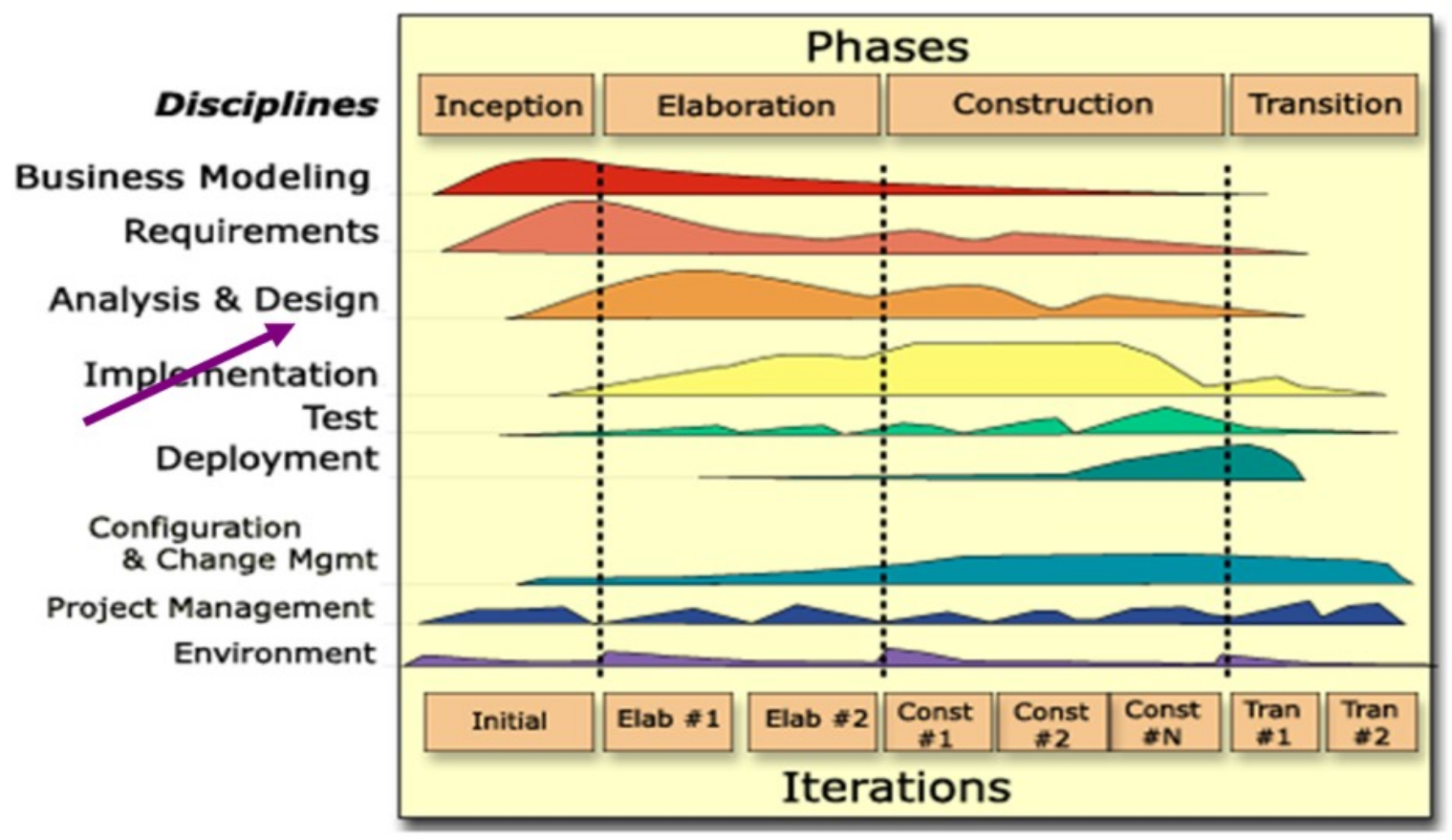
VH– Publish.

From 4:10 to 4:50 – 40 minutes.

10.25.2023 (W 4 to 5:30) (19)		Lecture 7: HOW - Design Tutorial 5 ERD to Relational			

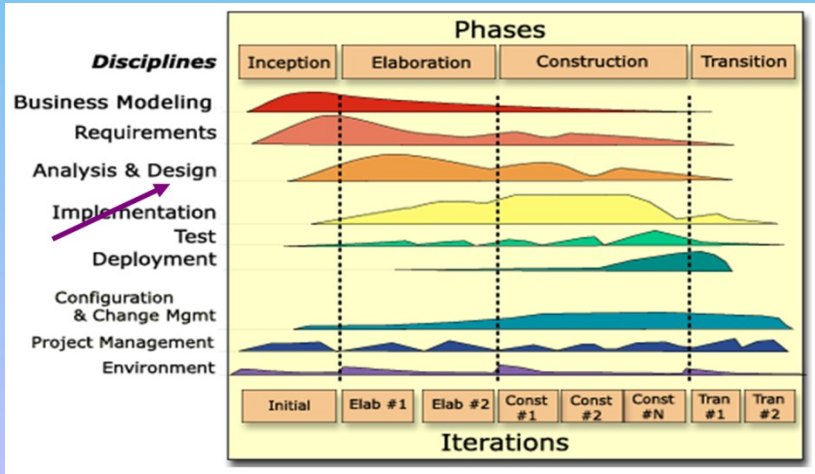


Design Workflow

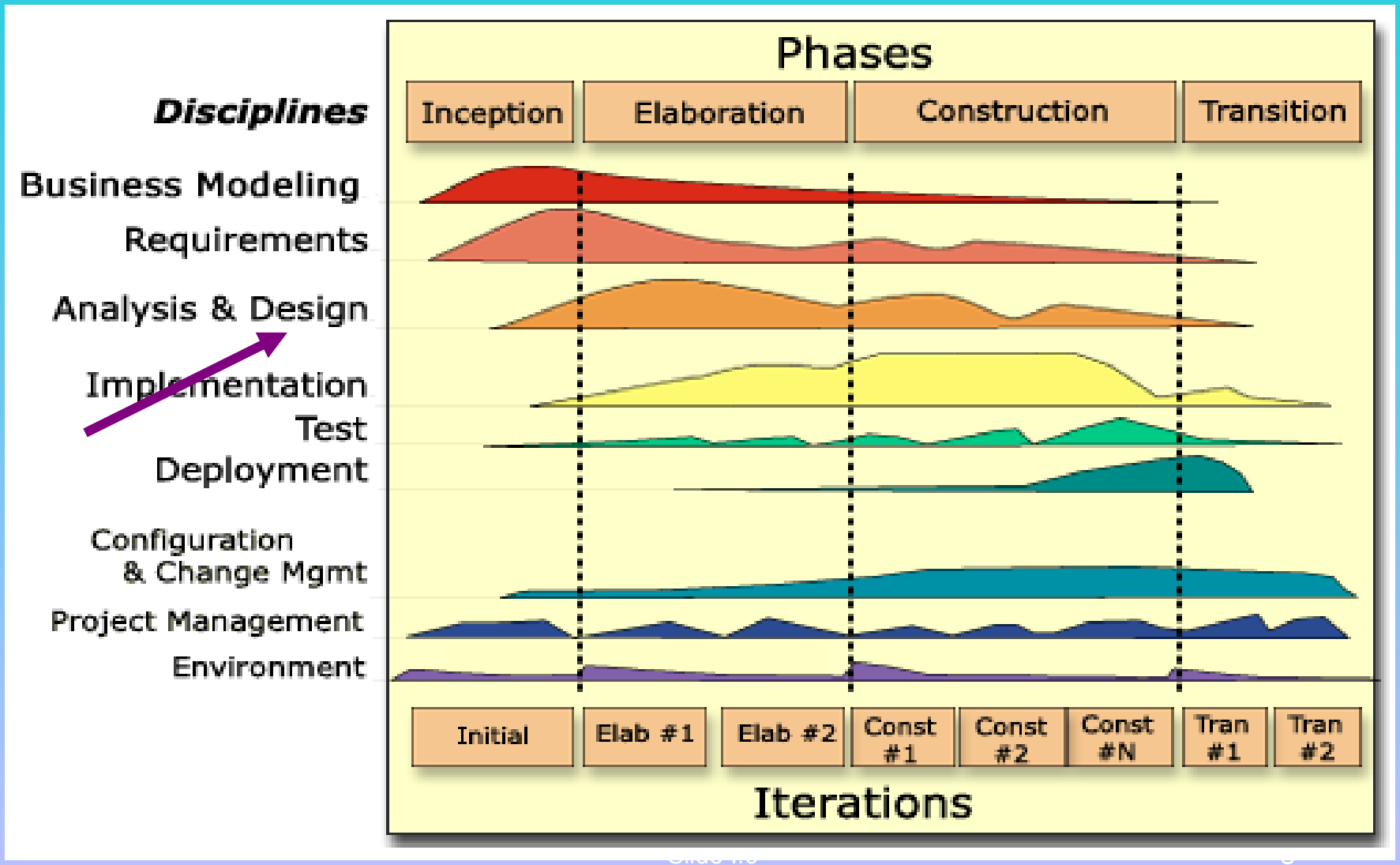


Classical DESIGN

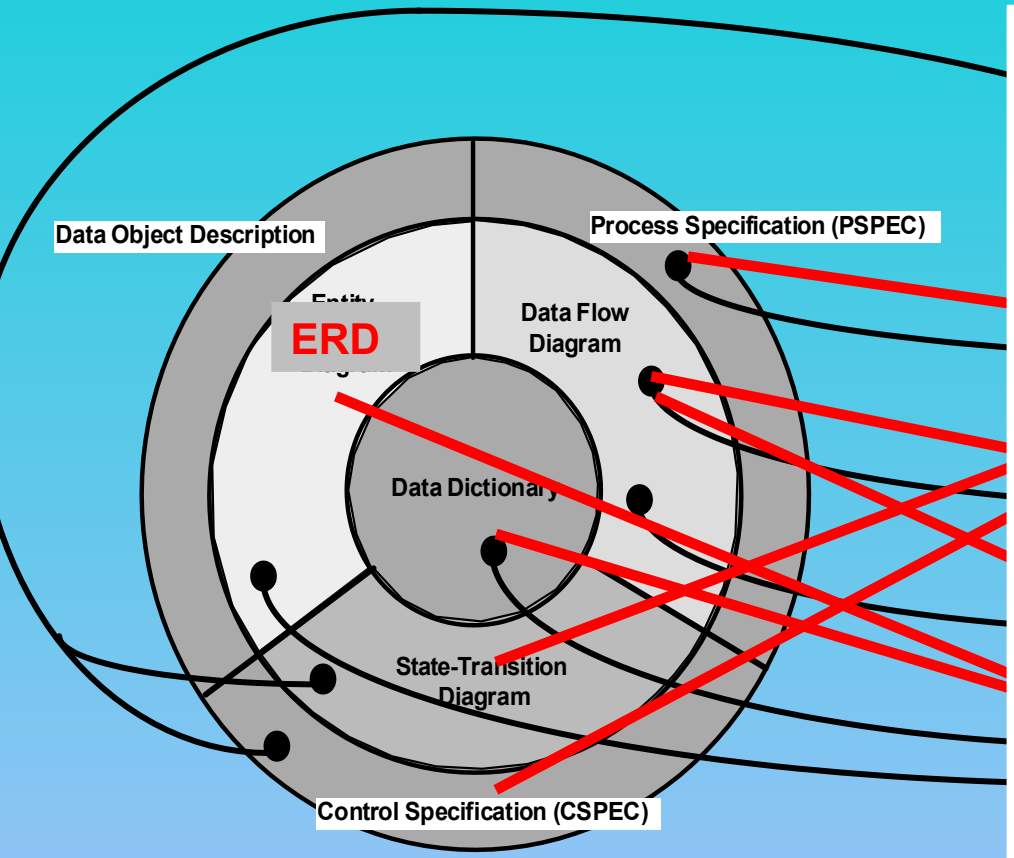
OBJECT ORIENTED DESIGN



The Unified Process



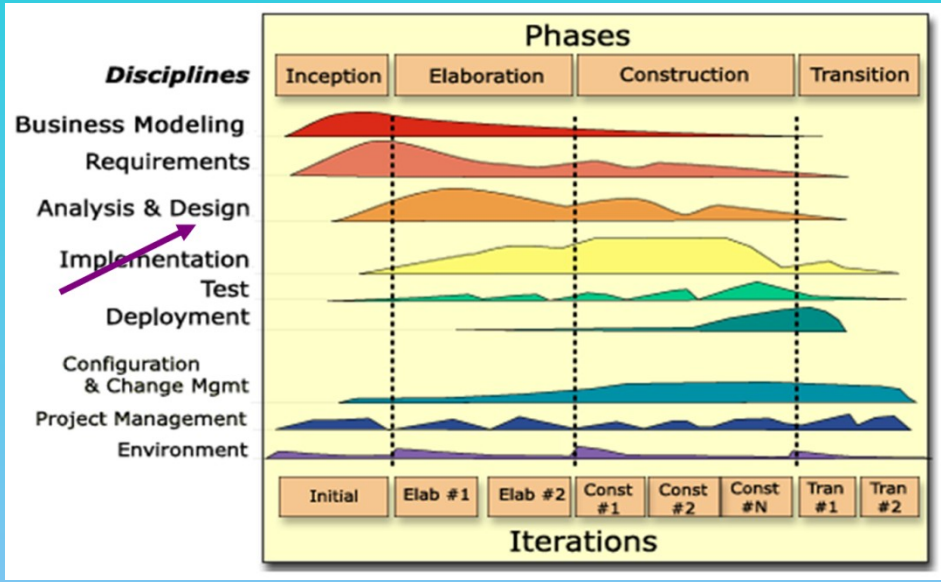
Analysis Workflow - WHAT to Design Workflow - HOW



THE ANALYSIS MODEL

Overview

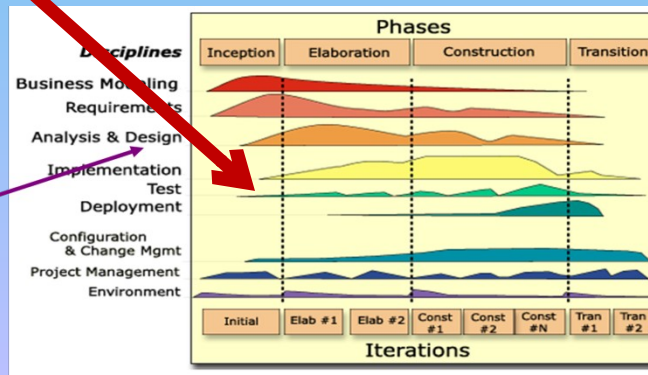
- **Classical Design** and abstraction
- **operation**-oriented **Design**
- Data Flow **Analysis**
- Transaction **Analysis**



- **O**bject **O**riented **D**esign: **The Elevator problem** case study

Overview

- The **Design Workflow**
- The **Test Workflow**: **Design**
- Formal techniques for **Detailed Design**
- **CASE** tools for **Design**
- **Metrics** for **Design**
- **Challenges** of the **Design Workflow**
- **S**oftware **P**roject **M**anagement **P**lan



Design – HOW Workflow

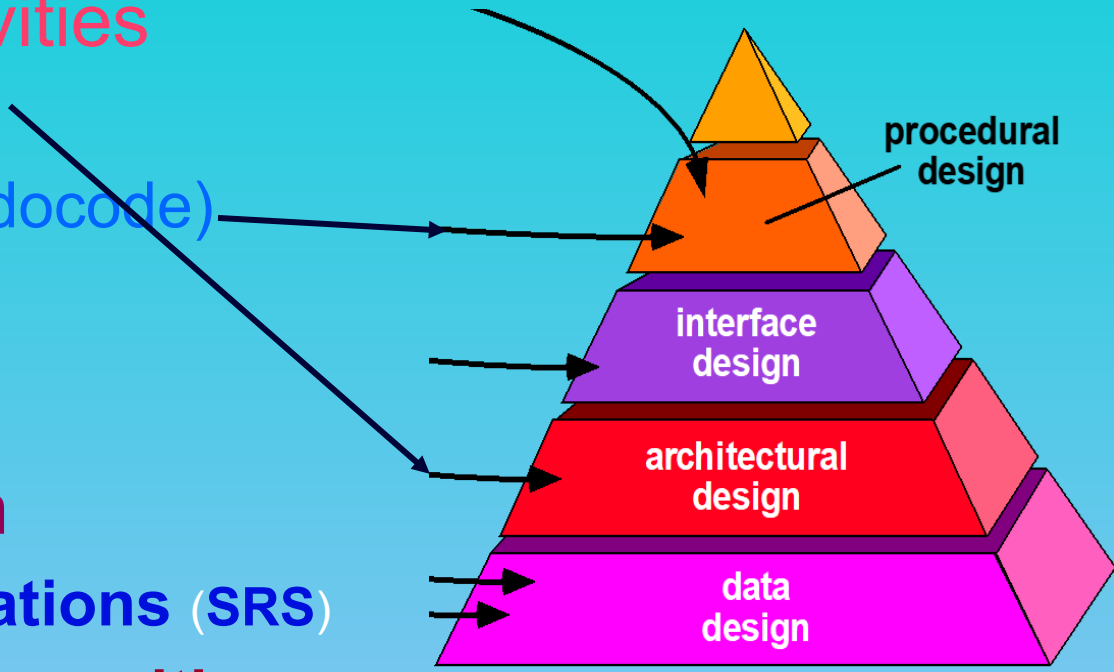
- Two aspects of a **product**
  - **actions** that operate on **data**
  - **data** on which **actions** operate
- The first basic ways of **Designing** a **product**
  - **Operation**-Oriented **Design**
- Second way of **Designing** a **product**
  - Hybrid methods: keep together **Data** and **Actions**

Object Oriented Paradigm

Classical Design and Abstraction

Classical Design Activities

1. Architectural Design
2. Detailed Design (pseudocode)
3. Design Testing



THE DESIGN MODEL

1. Architectural Design

- Input: Specifications (SRS)
- Output: modular Decomposition

2. Detailed Design (pseudocode)

- Each module is Designed
  - » Specific algorithms, data structures

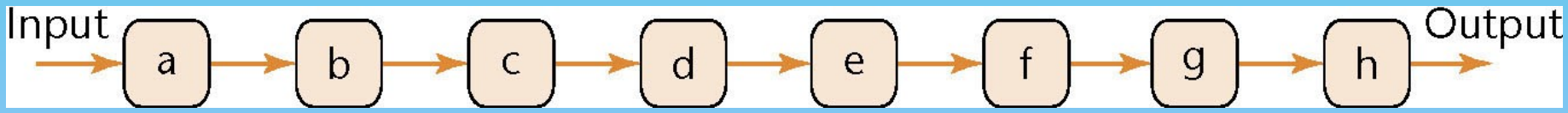
EXAM 3 REVIEW

Classical Design

Operation-Oriented Design

Operation-Oriented Design - HOW

- Key point: We have detailed **action** information from the **DFD** (**D**ata **F**low **D**iagram – **Analysis - WHAT**)



Data Flow Analysis (DFD)

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

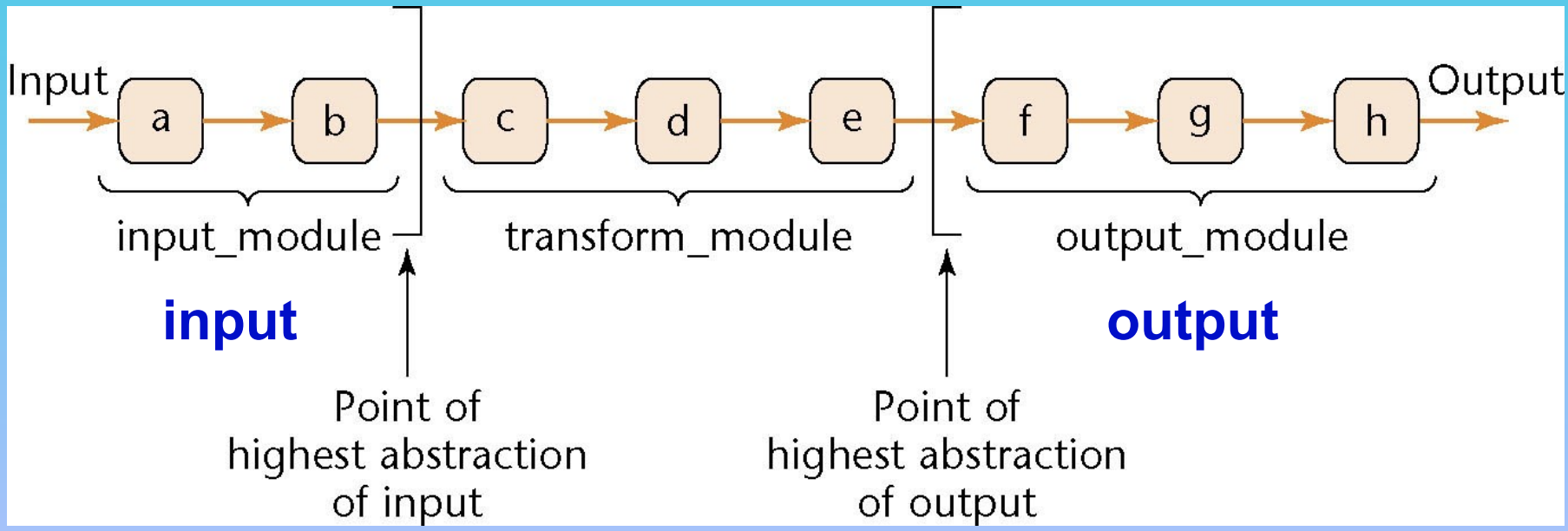


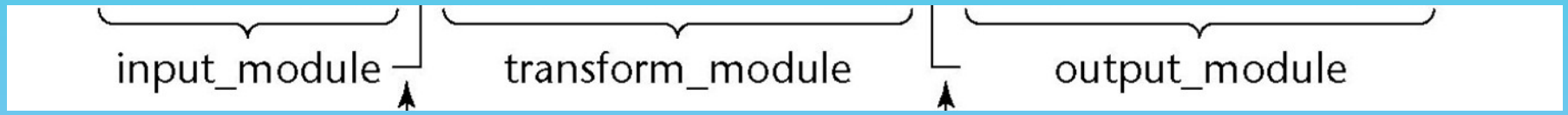
Figure 13.2

Data Flow Analysis (DFD)

Decompose the **product** into **three modules**
(remember Miller's 7 now 5 plus/minus is the limit?

3 three modules

)



input **process** **output**

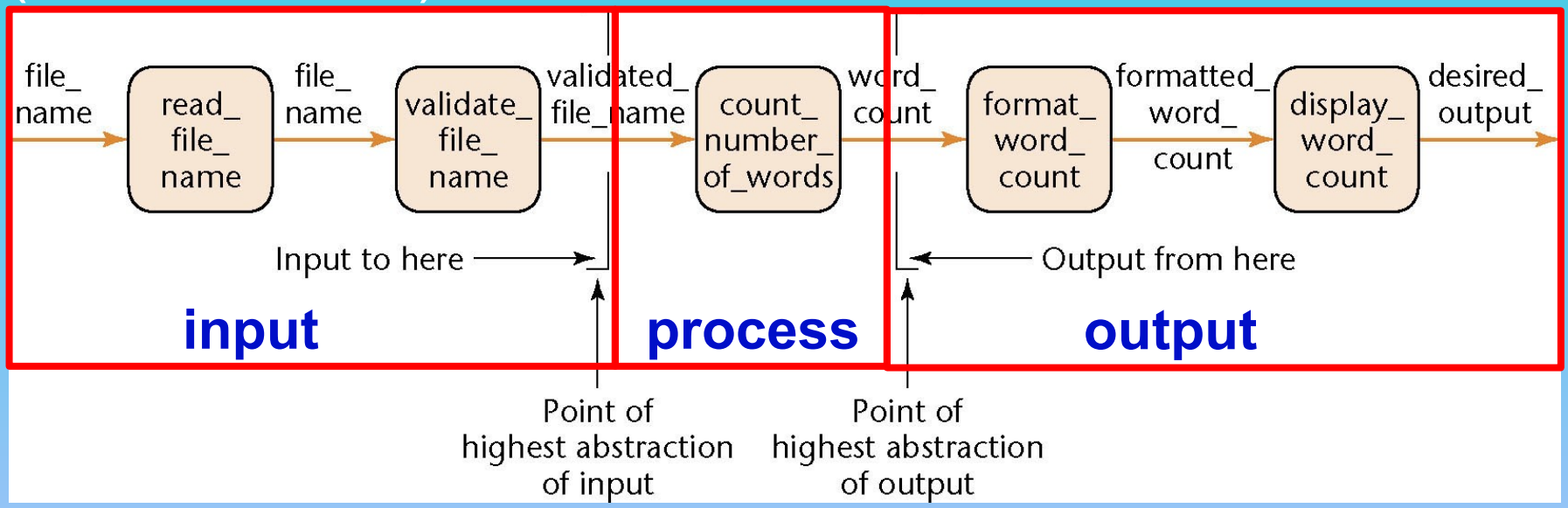
Repeat stepwise until **each module** has ***high cohesion***

Functional cohesion

- Minor modifications may be needed to **lower the coupling**

Mini Case Study: Word Counting DFD

Given the **Specification** (**Analysis - WHAT**) below, **Design - HOW** a **product** which takes as **input** a file name, and returns the **number of words** in that file (like UNIX *wc*)

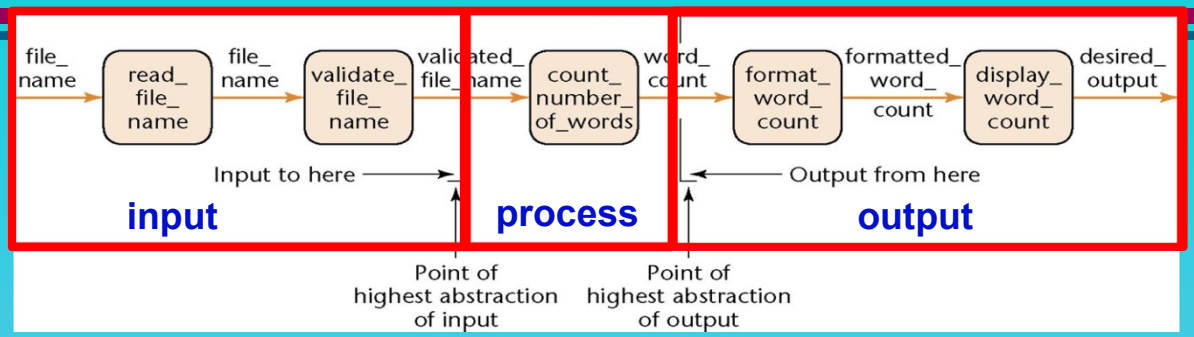


How many **modules**?

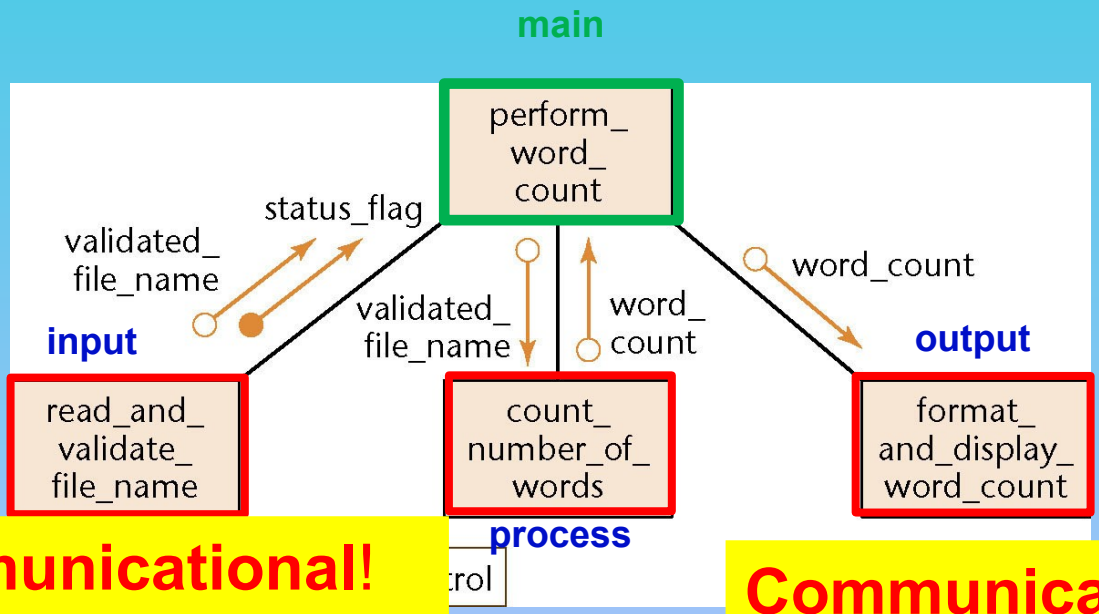
4

HOW?

Mini Case Study: Word Counting



First refinement

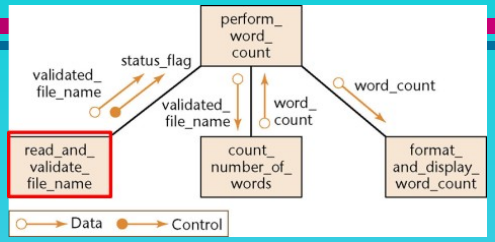


Communicational!

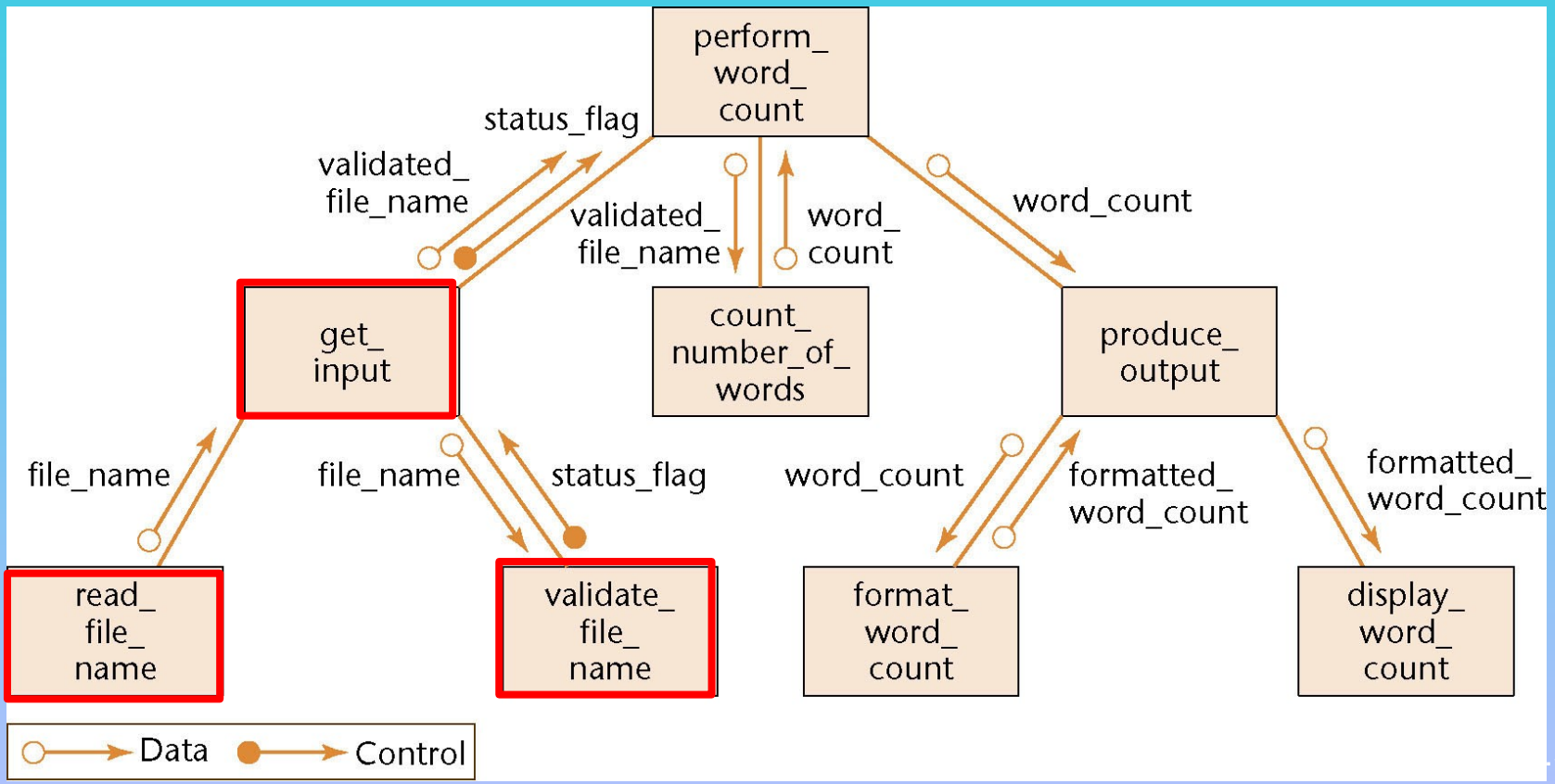
Communicational!
Functional!

What kind of **cohesion we want**?

Mini Case Study: **Word Counting**



- **Second refinement**



- All **eight modules** now have **functional cohesion**

Design and Abstraction

- **Classical Design Activities**
  1. Architectural Design
  2. **Detailed Design** (pseudocode) "**blue print**"
  3. Design testing

- 1. Architectural Design
  - Input: Specifications (**Software Requirements Specificatio**
    SRS)
  - Output: Modular decomposition

- 2. **Detailed Design** (pseudocode) "**blue print**"
  - Each **module** is **Designed**
    - » Specific algorithms, data structures

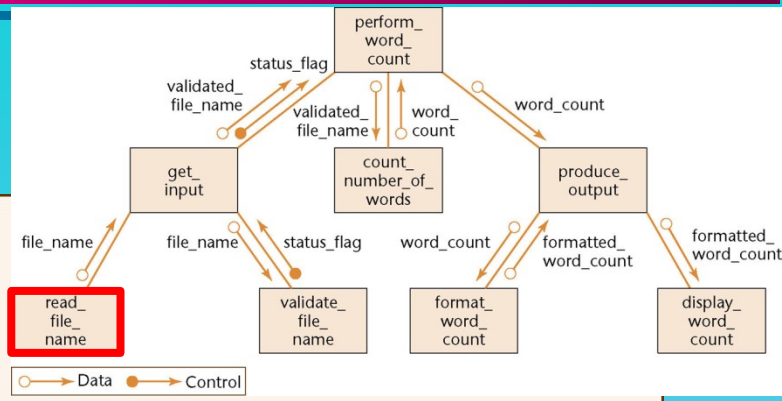
Word Counting: Detailed Design (pseudocode) "blue print"

1. The Architectural Design is complete
   - So proceed to the Detailed Design (pseudocode)

2. Two formats for representing the Detailed Design (pseudocode) "**blue print**" :
   - **Tabular**
   - **PDL** (PDL — Program Design Language)

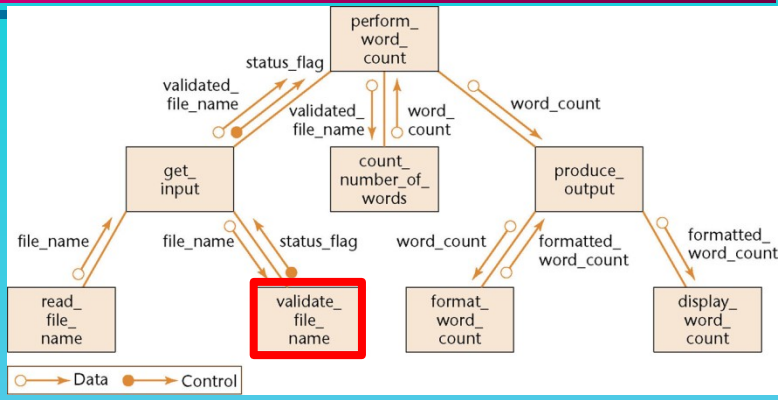
EXAM 3 REVIEW

Detailed **Design** (pseudocode) : **Tabular** Format



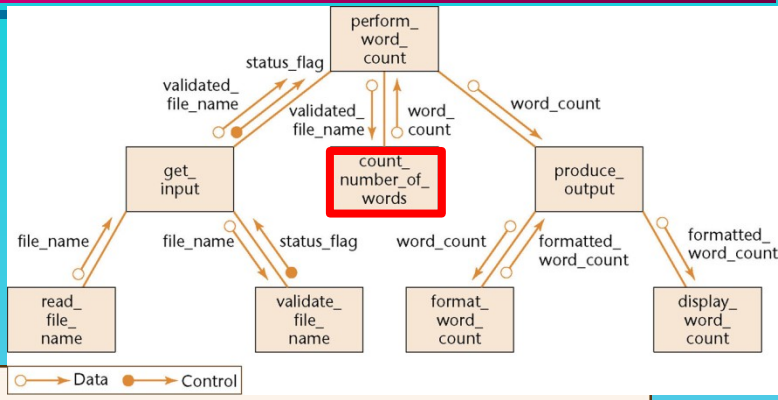
Module name	**read_file_name**
Module type	Function
Return type	**string**
Input arguments	None
Output arguments	**file_name : String**
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	The product is invoked by the user by means of the command string **word_count <file_name>** Using an operating system call, this module accesses the contents of the command string input by the user, extracts **<file_name>**, and returns it as the value of the module.

Detailed **Design** (pseudocode) : **Tabular** Format



Module name	**validate_file_name**
Module type	Function
Return type	**Boolean**
Input arguments	**file_name : string**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	This module makes an operating system call to determine whether file **file_name** exists. The module returns **true** if the file exists and **false** otherwise.

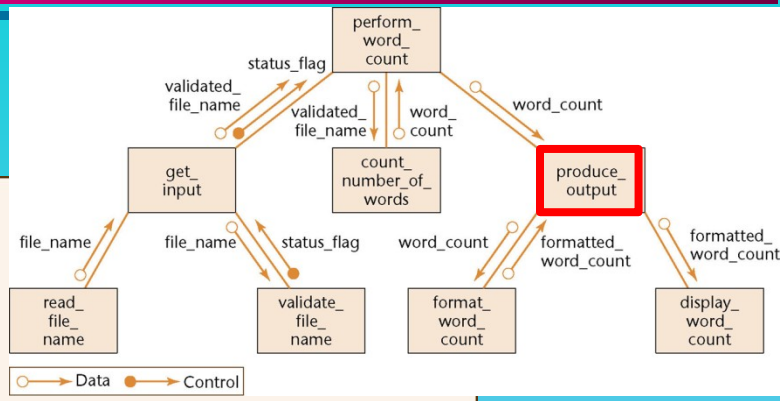
Detailed **Design** (pseudocode) : **Tabular** Format



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**.

6(c)

Detailed Design (pseudocode) : Tabular Format



Module name	**produce_output**
Module type	Function
Return type	**void**
Input arguments	**word_count : integer**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	**format_word_count** arguments: **word_count : integer** **formatted_word_count : string** **display_word_count** arguments: **formatted_word_count : string**
Narrative	This module takes the integer **word_count** passed to it by the calling module and calls **format_word_count** to have that integer formatted according to the specifications. Then it calls **display_word_count** to have the line printed.

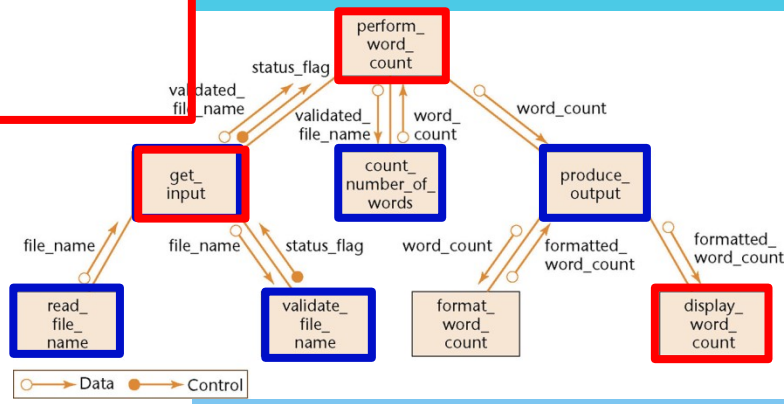
Detailed Design (pseudocode) : PDL Format

(Program Design Language)

```
void perform_word_count()
{
  String                    validate_file_name;
  int                       word_count;

  if (get_input (validate_file_name) is false)
     print "error 1: file does not exist";
  else
  {
     set word_count equal to count_number_of_words (validate_file_name);
     if (word_count is equal to –1)
        print "error 2: file is not a text file";
     else
        produce_output (word_count);
  }
}
```

```
Boolean get_input (String validate_file_name)
{
  String                    file_name;

  file_name = read_file_name ();
  if (validate_file_name (file_name) is true)
  {
     set validate_file_name equal to file_name;
     return true;
  }
  else
     return false;
}
```

```
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";
}
```



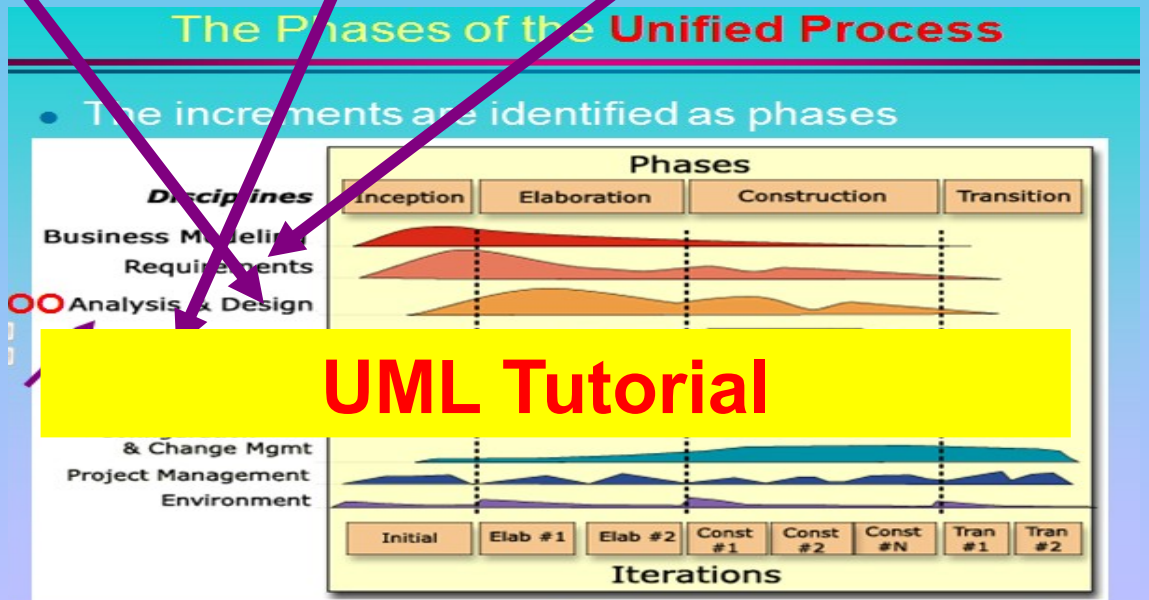
EXAM 3 REVIEW

Object-Oriented Design

Recap: Object Oriented Analysis Workflow

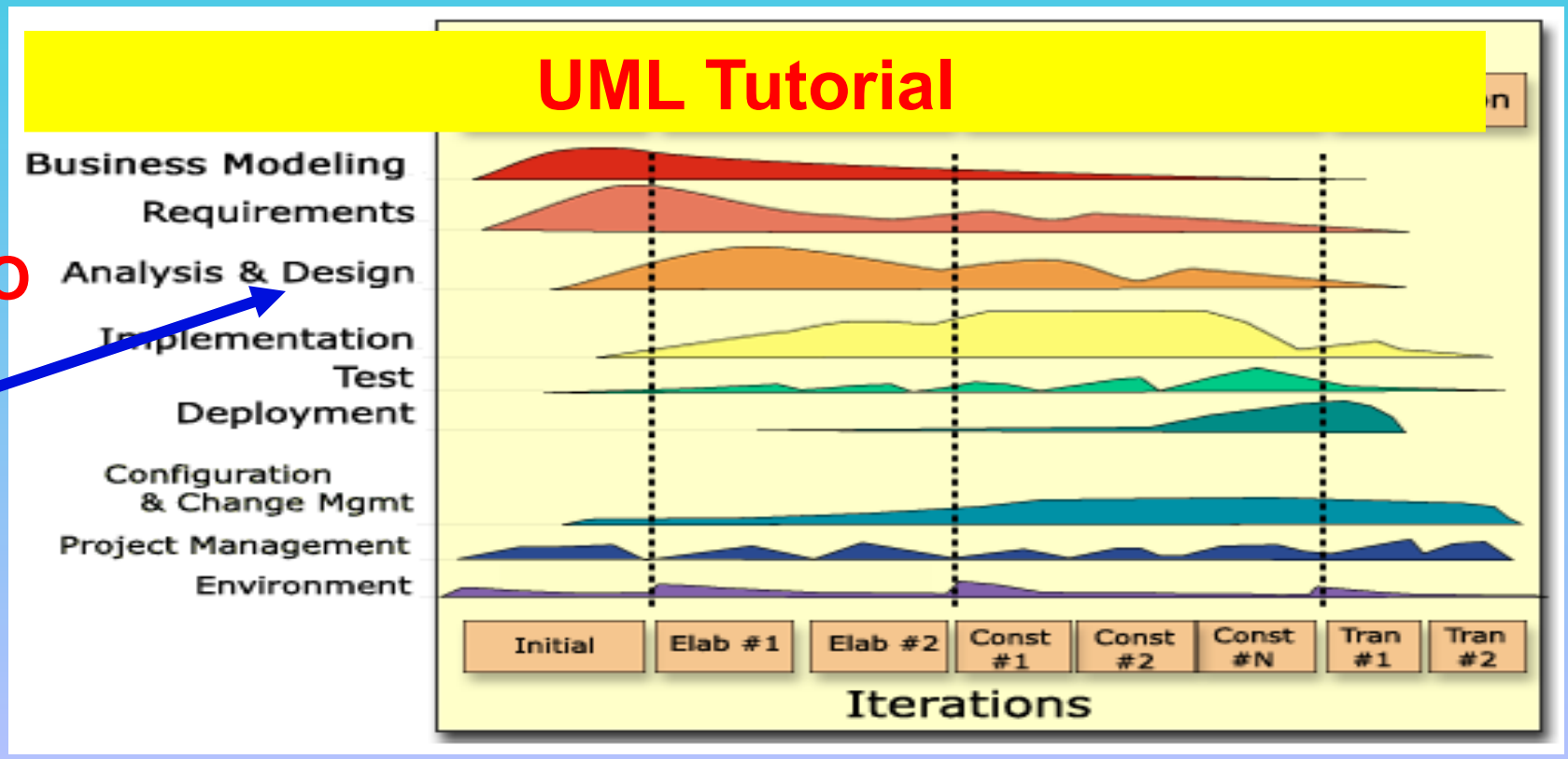
The **Object Oriented Analysis Workflow** has two aims:

- Obtain a *deeper understanding of the* ***Requirements***
- Describe ***Requirements*** in a way that will result in a **Maintainable OO Design** and **OO Implementation Workflows**



OO Design Workflow – "blue print"

- **OO Design Workflow – HOW "blue print" :**
  - The **OO Analysis Workflow artifacts are iterated and incremented until the Software Engineer can utilize them**



Object Oriented Design - HOW Steps

Aim

- OO Design the product in terms of the Classes extracted during OOA

OOD consists of two steps:

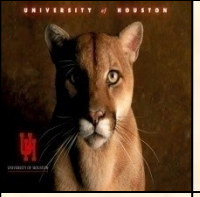
Step 1. Complete the UML MVC Class Diagram

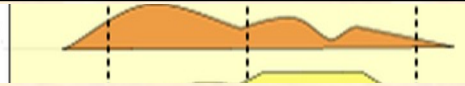
a. Determine the formats of the attributes

b. 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 (pseudocode)

EXAM 3 REVIEW





UML OOD Modeling Steps

OO Design Workflow

Textual Analysis for

Second UML MODEL: UML MVC OOD Classes Diagram + Methods Pseudocode

Step 4: Complete UML MVC Classes Diagram

Step 5: Detailed Design Pseudocode for methods (pseudocode)

Object Oriented Design Steps

- Step 1. Complete the **UML Class Diagram**
  a. The formats of the **attributes** can be directly deduced from the **Analysis** artifacts

- Example: **Dates**
  - U.S. format (mm/dd/yyyy)
  - European format (dd/mm/yyyy)
  - In both instances, 10 characters are needed

- The **formats** could be added during **OO Analysis**
  - To minimize **rework**, ***never*** add an **item** to a **UML Class Diagram** until strictly necessary

Object Oriented Design Steps

- Step 1. Complete the **UML Class Diagram**
  - **b. 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**

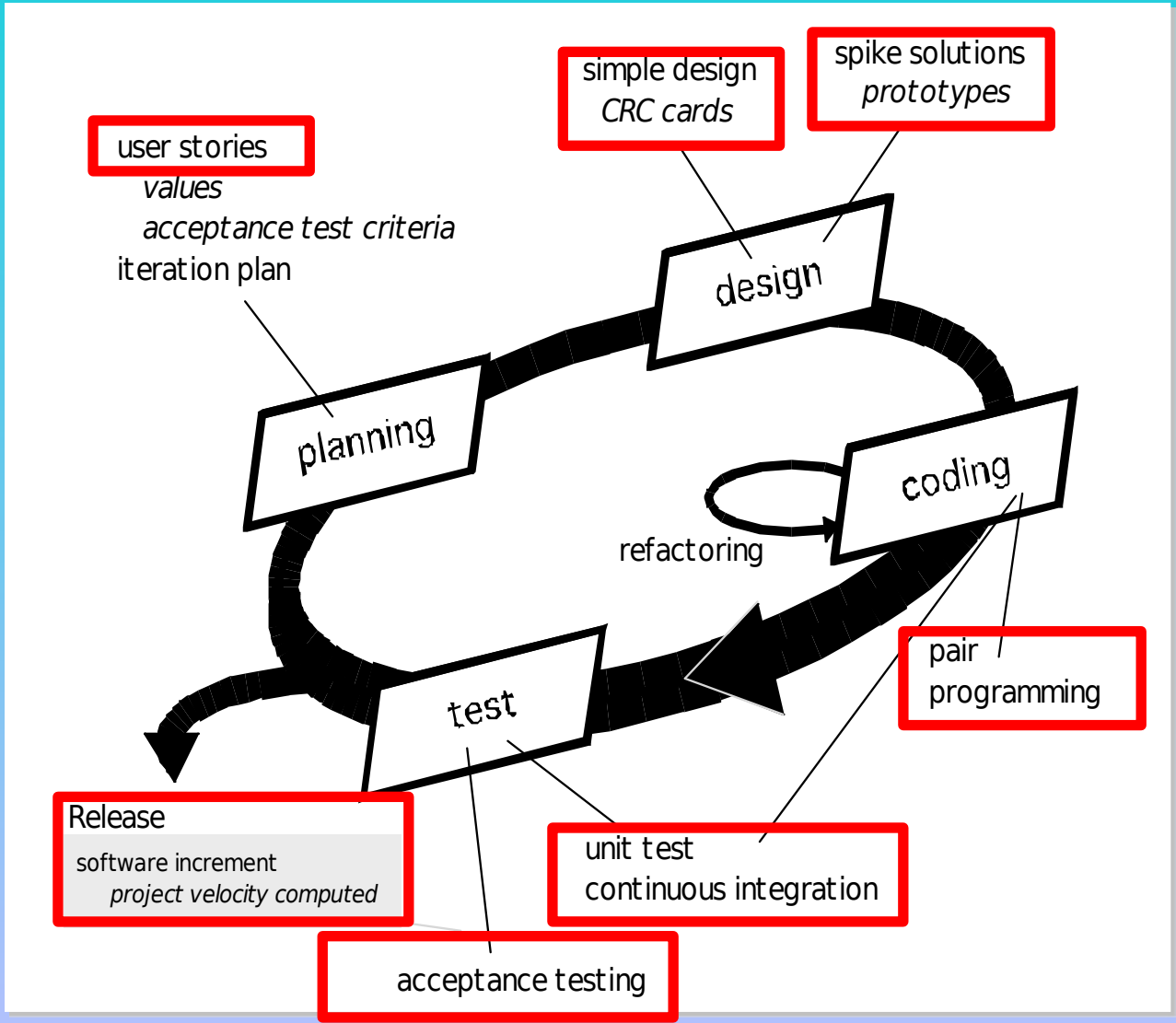
Object Oriented Design: The Elevator Problem Case Study

REVIEW:
Specialized Process Models: EXtreme Programming (XP) Model

- Widely used **Agile Process**, proposed by Kent Beck
- XP Planning
  - Begins with the creation of "user stories" **UML Use Case Description**
  - Agile **team** assesses each **story** and assigns a **cost**
  - **Stories** are grouped to form a deliverable Increment
  - A commitment is made on **delivery date**

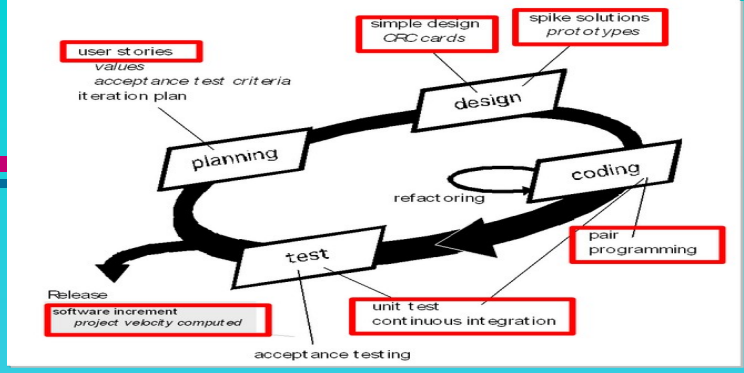
REVIEW:

EXtreme Programming (XP) Model



REVIEW:

EXtreme Programming (XP)



XP **Design**

- Follows the KIS**S** principle
- Encourage the use of CRC cards (**C**lass-**R**esponsibility-**C**ollaboration)
- For difficult design problems, creation of "spike solutions"—a design prototype
- Encourages "refactoring"—an Iterative refinement of the internal **program Design**

XP **Coding**

- Recommends the construction of a unit test *before* coding commences (**TDD**)
- Encourages "pair programming"

XP **Testing**

- All unit tests are executed **daily** (**Continuous Integration**)
- "Acceptance tests" defined by **Customer**; executed to assess **Customer** functionality

After the first Increment "project velocity" is used to help define subsequent **delivery dates** for other Increments

REVIEW:

Describe some features of the Extreme Programming (XP) Life-Cycle Model.

Extreme programming [Beck, 2000] is a somewhat controversial new approach to software development based on the iterative-and-incremental model. The first step is that the software development team determines the various features (**stories**) the client would like the product to support. For each such feature, the team informs the client how long it will take to implement that feature and how much it will cost. This first step corresponds to the requirements and analysis workflows of the iterative-and-incremental model (Figure 2.4).

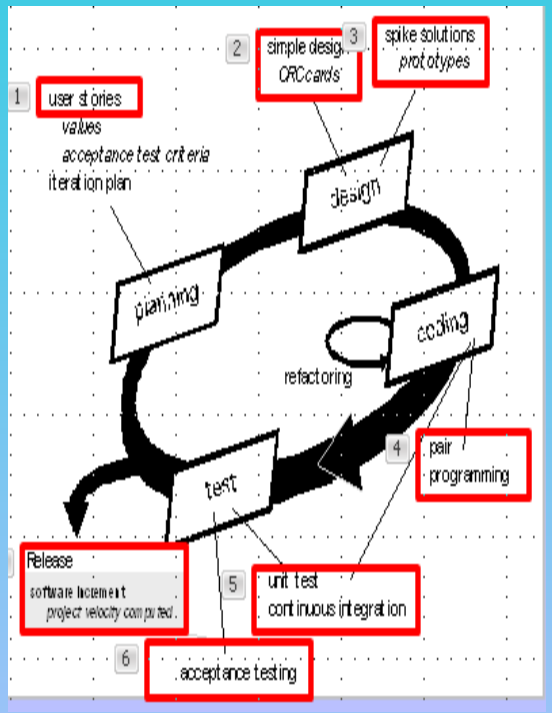
The client selects the features to be included in each successive build using cost–benefit analysis (Section 5.2), that is, on the basis of the duration and the cost estimates provided by the development team as well as the potential benefits of the feature to his or her business. The proposed build is broken down into smaller pieces termed **tasks**. A programmer first [illegible] test cases for a task; this is termed **test-driven development** (TDD). [illegible] mers work together on one computer (**pair programming**) [Williams, Kessler, Cunningham, and Jeffries, 2000], implementing the task and ensuring that all the test cases work correctly. The two programmers alternate typing every 15 or 20 minutes; the programmer who is not typing carefully checks the code while it is being entered by his or her partner. The task is then integrated into the current version of the product. Ideally, implementing and integrating a task should take no more than a few hours. In general, a num[illegible] implement tasks in parallel, so integration is essentially continuous [illegible] change coding partners daily, if possible; learning from the other team members increases everyone's skill level. The TDD test cases used for the task are retained and utilized in all further integration testing.

Some drawbacks to pair programming have been observed in practice [Drobka, Noftz, and Raghu, 2004]. For example, pair programming requires large blocks of uninterrupted time, and software professionals can have difficulty in finding 3- to 4-hour blocks of time. In addition, pair programming does not always work well with shy or overbearing individuals, or with two inexperienced programmers.

UML

TDD

ccnet

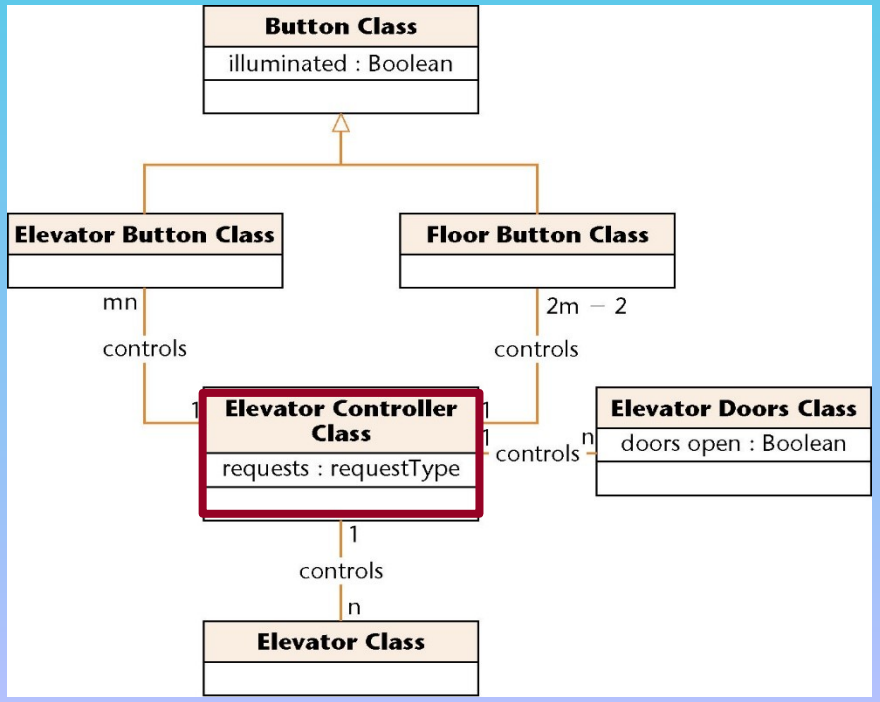


What would you say?

Object Oriented Design: The Elevator Problem Case Study

- Step 1. Complete the **UML Class Diagram**

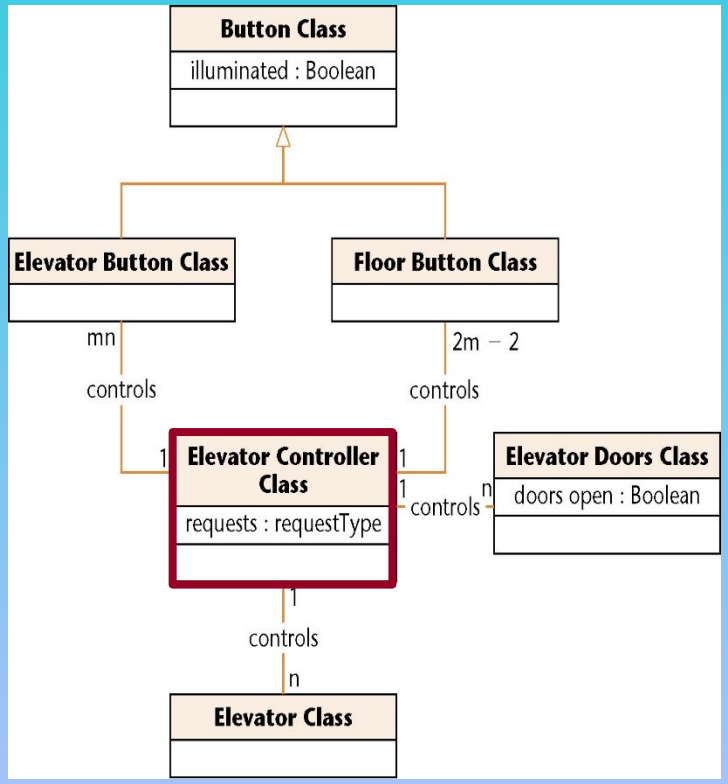
- Consider the **third iteration** of the **CRC card** for the **Elevator Controller Class (OOA artifact)**



C	CLASS **Elevator Controller Class**
R	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
C	COLLABORATION 1. **Elevator Button Class** (subclass) 2. **Floor Button Class** (subclass) 3. **Elevator Doors Class** 4. **Elevator Class**

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**

Object Oriented Design Steps

Aim

- **Design** the product in terms of the classes extracted during **OOA**

OOD consists of two steps:

Step 1. Complete the **UML Class Diagram**

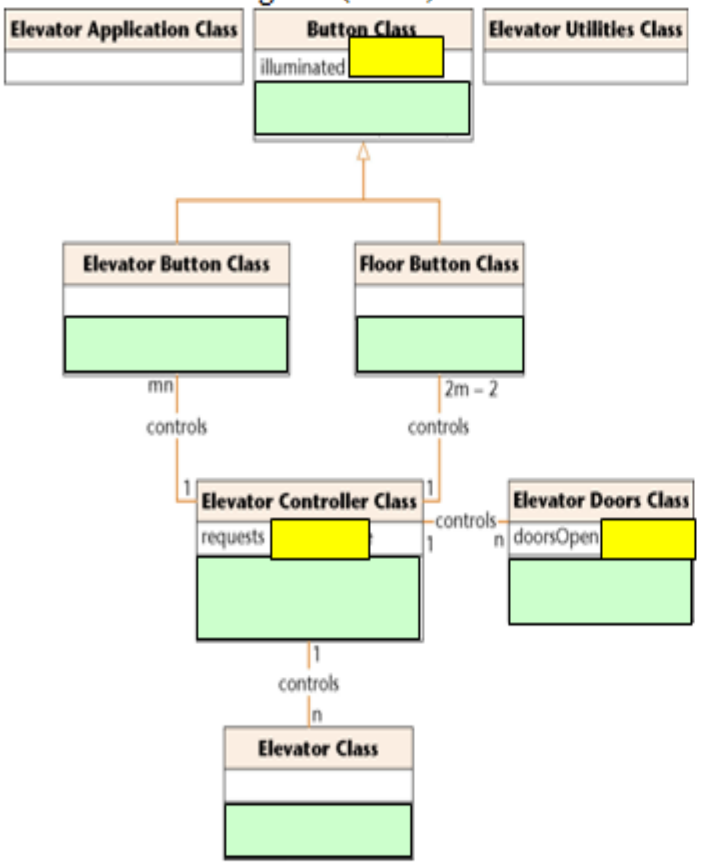
a. Determine the **formats of the attributes**

b. **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 - pseudocode**

Third Iteration of UML Class Diagram

And the Class Diagram (OOA):



Perform OOD: Assign types to attributes

&

the 11 methods to their corresponding classes

&

write pseudo-code for method turnOnButton

Assign types to attributes

OOD: Elevator Problem Case Study

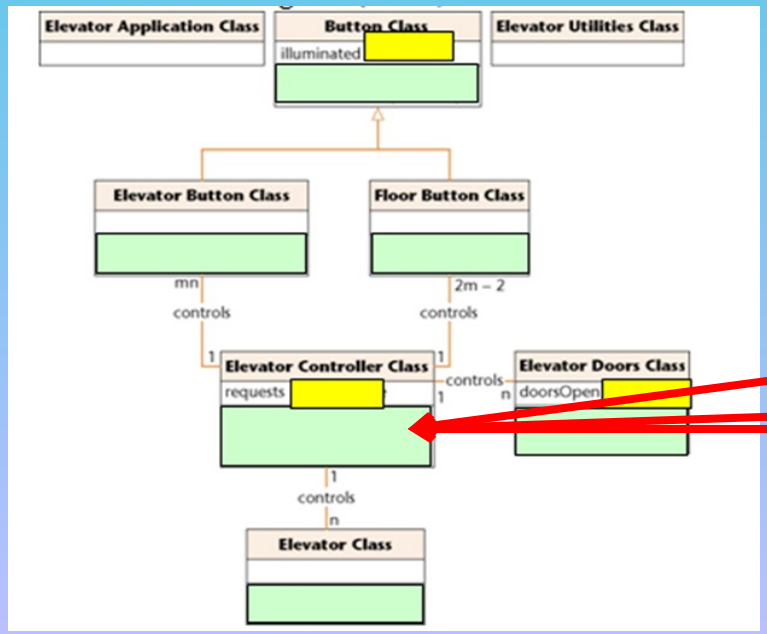
Responsibilities

11 methods to their corresponding classes

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

are assigned to the **Elevator Controller Class**

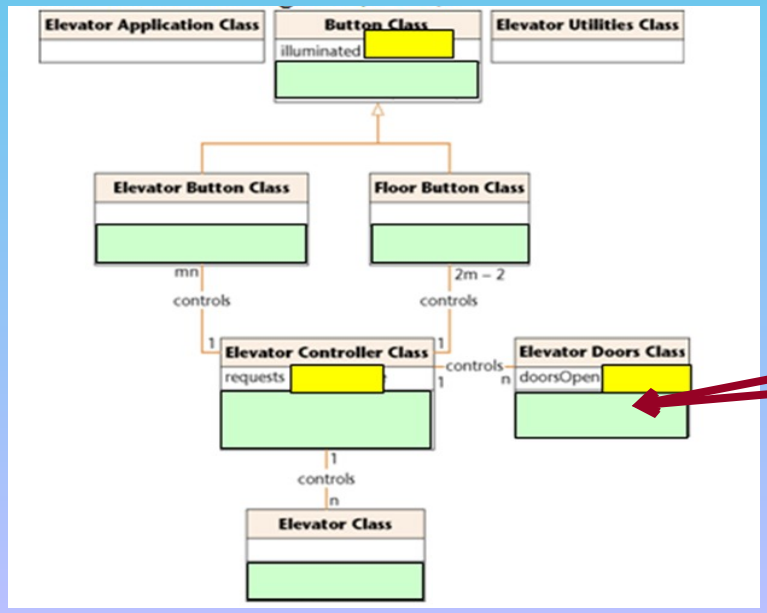
Because they are carried out by the **Elevator Controller**



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**

OOD: Elevator Problem Case Study

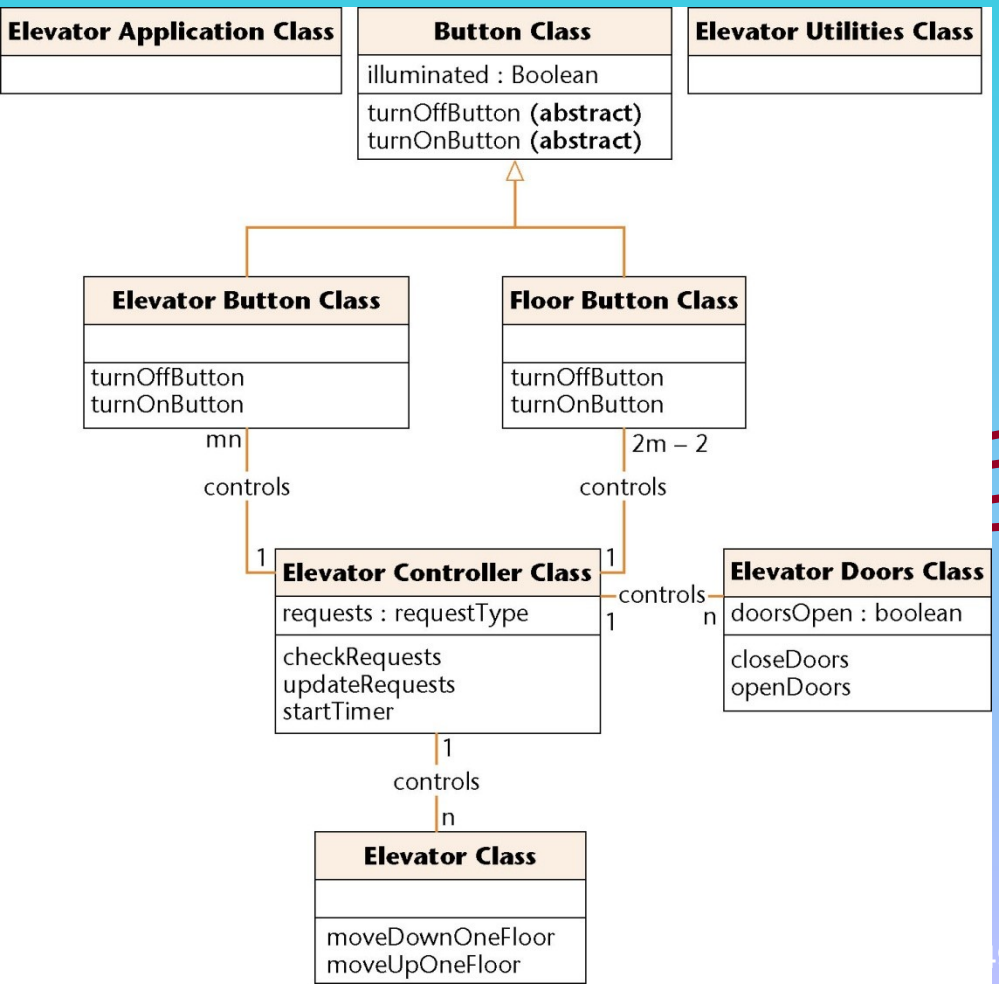
- The remaining eight responsibilities have the form
  - "Send a message to **another Class** to tell it do something"
- These should be **assigned to that other Class**
  - **Responsibility Driven Design**
  - **Safety considerations**
- Methods `7. open doors`, `9. close doors` are assigned to **Elevator Doors Class**



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**

OOD: Elevator Problem Case Study

Methods 2., 4. `turn off button`, 1., 3. `turn on button` are assigned to **classes Floor Button Class** and **Elevator Problem Class**



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**

Object Oriented Design Steps

Aim

- **Design** the product in terms of the classes extracted during **OOA**

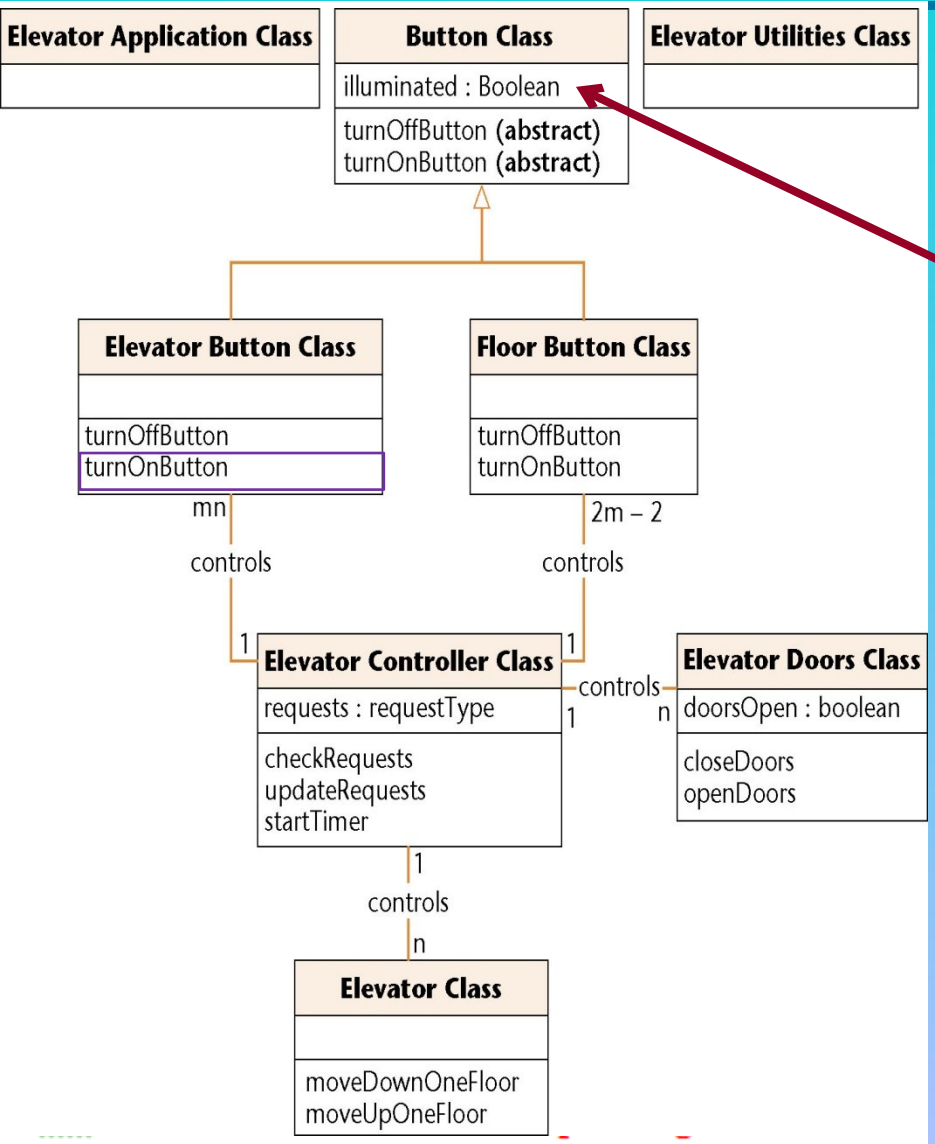
OOD consists of two steps:

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** (pseudocode)

Third Iteration of UML Class Diagram



&

write pseudo-code for method turnOnButton

write pseudo-code for method turnOnButton

```
void turnOnButton ()
{
    set illuminated to TRUE;
}
```

(OOA - WHAT)

- Produced **UML Statechart**
- **(during OOA)**

- State, event, and predicate are distributed over the **UML Statechart**

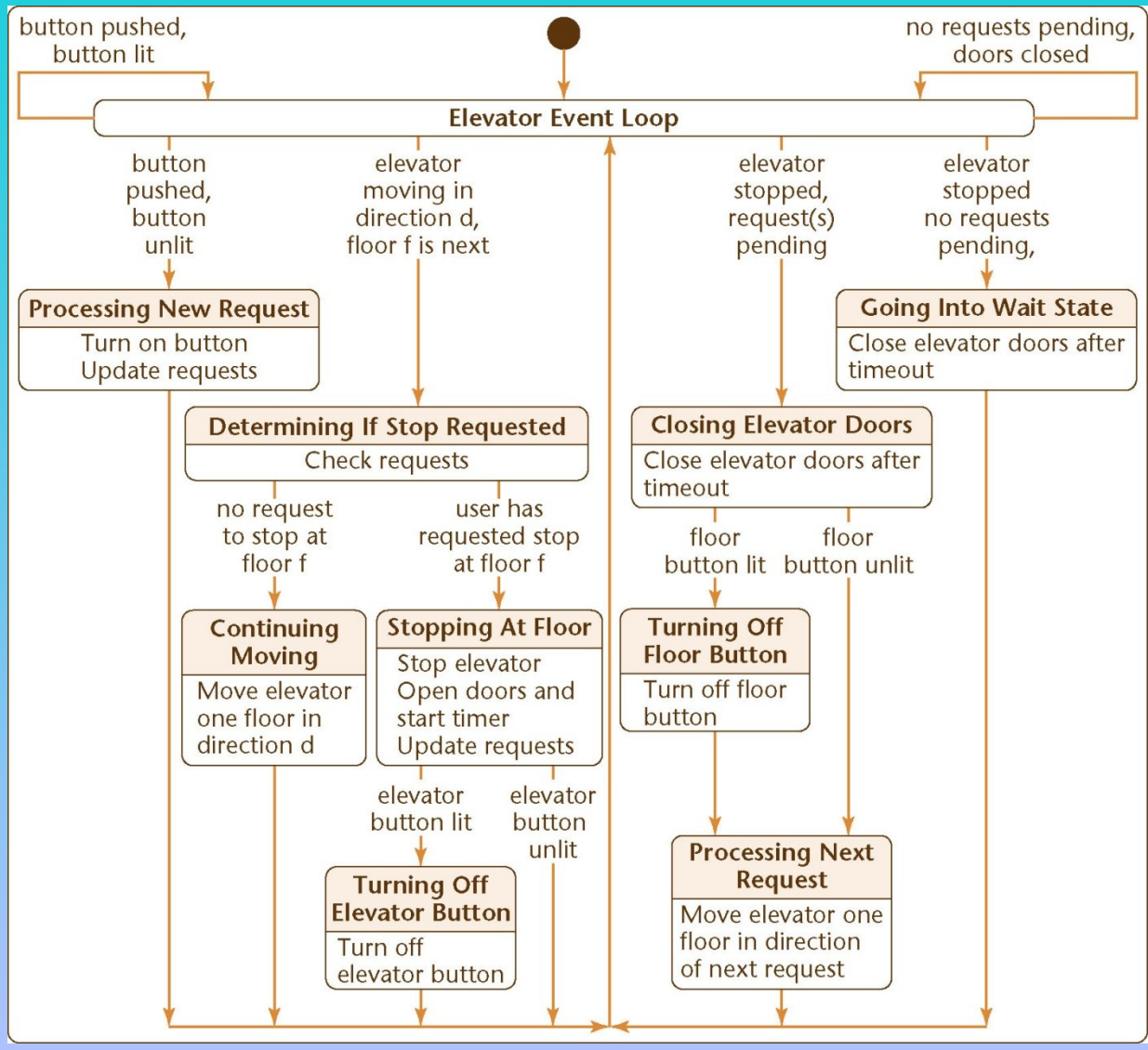
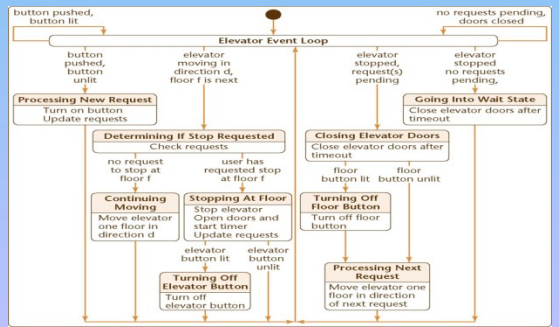


Figure 12.7

How many States?

Detailed Design (pseudocode) Elevator Problem

- Step 2. **OOD – HOW**:
- Complete the class diagram **Detailed Design** (pseudocode) of `elevatorEventLoop` is constructed from the **UML Statechart (OOA - WHAT)**




```
void elevatorEventLoop (void)
{
  while (TRUE)
  {
    if (a button has been pressed)
      if (button is not on)
      {
        updateRequests;
        button::turnOnButton;
      }
    else if (elevator is moving up)
    {
      if (there is no request to stop at floor f)
        elevator::moveUpOneFloor;
      else
      {
        stop elevator by not sending a message to move;
        elevatorDoors::openDoors;
        startTimer;
        if (elevatorButton is on)
          elevatorButton::turnOffButton;
        updateRequests;
      }
    }
    else if (elevator is moving down)
      [similar to up case]
    else if (elevator is stopped and request is pending)
    {
      elevatorDoors:closeDoors;
      determine direction of next request;
      if (appropriate floorButton is on)
        floorButton::turnOffButton;
      elevator:moveUp/DownOneFloor;
    }
    else if (elevator is at rest and not (request is pending))
      elevatorDoors::closeDoors;
    else
      there are no requests, elevator is stopped with elevatorDoors closed, so do nothing;
  }
}
```

OOA -> OO Detailed Design (pseudocode) Elevator Problem

"blue print"

eling: The Elevator Problem Case Study (OOA)

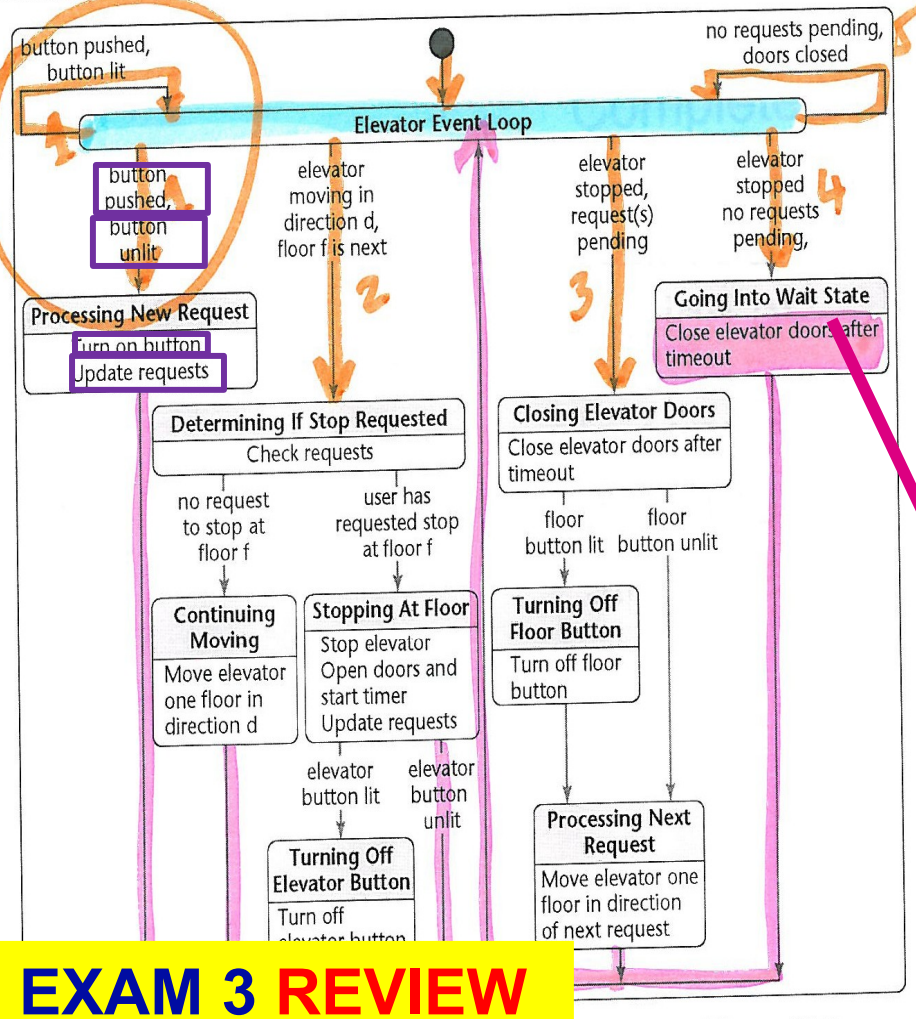


Figure 12.7

Design: Elevator Problem

```
void elevatorEventLoop (void)
{
  while (TRUE)
  {
    if (a button has been pressed)
      if (button is not on)
      {
        updateRequests;
        button::turnOnButton;
      }
    else if (elevator is moving up)
    {
      if (there is no request to stop at floor f)
        elevator::moveUpOneFloor;
      else
      {
        stop elevator by not sending a message to move;
        elevatorDoors::openDoors;
        startTimer;
        if (elevatorButton is on)
          elevatorButton::turnOffButton;
        updateRequests;
      }
    }
    else if (elevator is moving down)
      [similar to up case]
    else if (elevator is stopped and request is pending)
    {
      elevatorDoors::closeDoors;
      determine direction of next request;
      if (appropriate floorButton is on)
        floorButton::turnOffButton;
      elevator::moveUp/DownOneFloor;
    }
    else if (elevator is at rest and not (request is pending))
      elevatorDoors::closeDoors;
    else
      there are no requests, elevator is stopped
  }
```

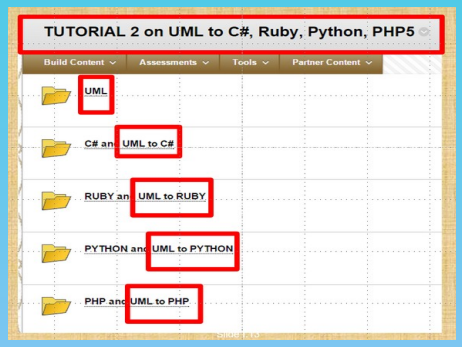
EXAM 3 REVIEW

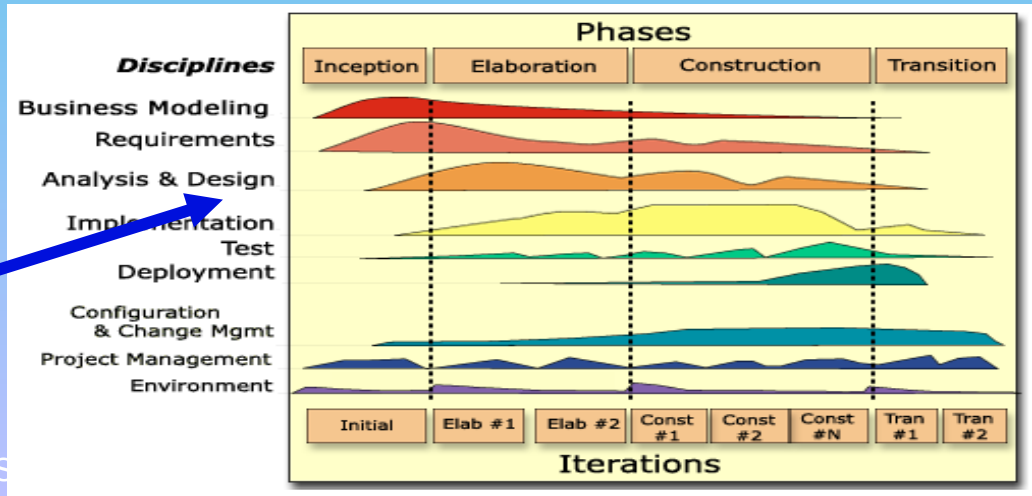
and so on…

The Design Workflow – HOW "blue print"

- Summary of the **Design Workflow**:
  - The **Analysis Workflow** artifacts are **iterated** and **incremented** until the Software Engineer can utilize them **"blue print"**

- Decisions to be made include:
  - **Implementation language**
  - **Reuse**
  - **Portability**





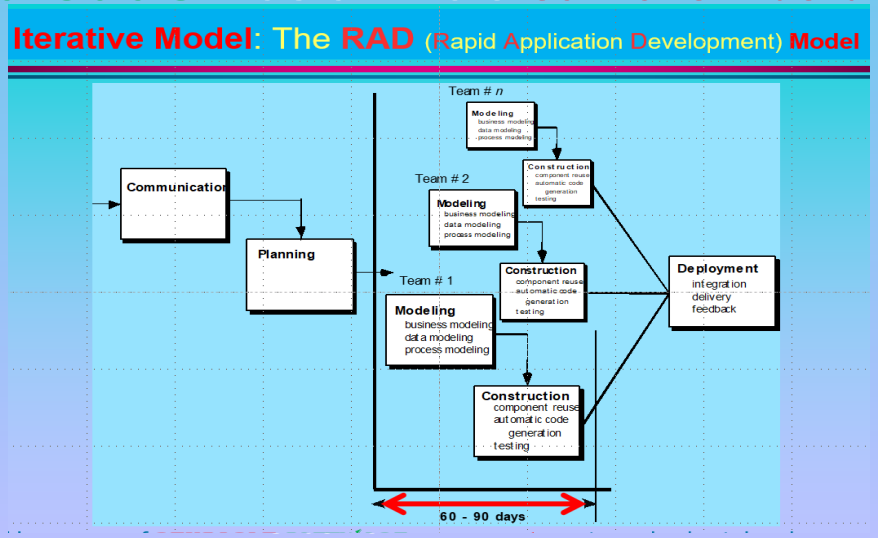
The Design Workflow - HOW "blue print"

- The idea of decomposing a **system** into independent smaller **subsystems** is carried forward to the **Design Workflow**

- The objective is to break up the upcoming **Implementation Workflow** into manageable pieces
  - ***Subsystems***

The Design Workflow - HOW "blue print"

- Why the **product** is broken into **subsystems**:
  - **It is easier** to **I**mplement a number of smaller subsystems than one **large system**
  - If the **subsystems** are independent, they can be **I**mplemented by **Programming Teams** working in **parallel**
    - » The **software product** as a whole **can then be delivered sooner**



The Design Workflow - HOW "blue print"

- The ***Architecture*** of a **software product** includes
  - The various components
  - How they fit together
  - The allocation of components to **subsystems**

- The task of **Designing** the **Architecture** is specialized
  - It is performed by a **Software *Architect***

The Design Workflow - HOW "blue print"

The Architect needs to make *trade-offs*

- Every **software product** must satisfy its **F**unctional **Requirements** (the **UML USE CASES /SRS – F**unctional **Requirements**)

- **It** also must satisfy its non**F**unctional **Requirements** (**SRS –** non**F**unctional **Requirements**), including
  - » Portability, Reliability, Robustness, Maintainability, and Security

- **It** must do all these things **within budget** and **time constraints**

The Architect must assist the Client by laying out the **trade-offs**

The Design Workflow - HOW "blue print"

- It is usually impossible to satisfy all the **Requirements**, **Functional** and non**Functional**, within the cost and time constraints
  - Some sort of compromises have to be made

- The **Client** has to
  - Relax some of the **Requirements**;
  - Increase the budget; and/or
  - Move the delivery deadline

The Design Workflow - HOW "blue print"

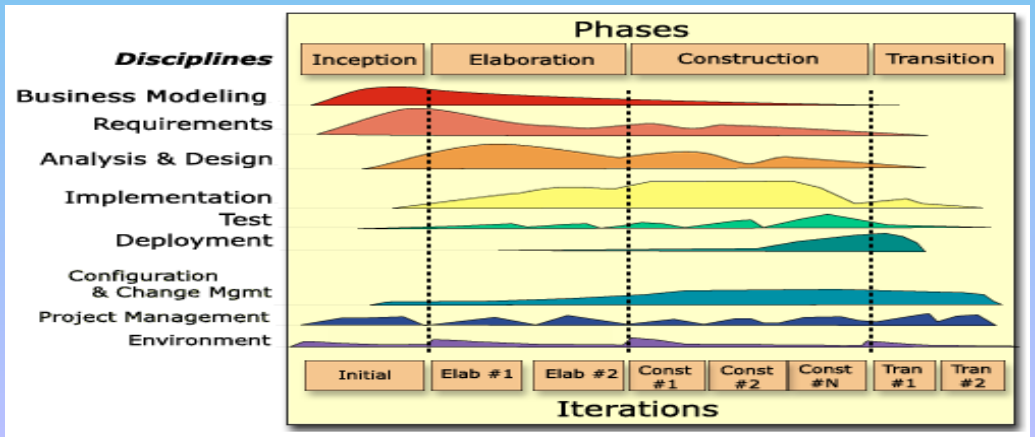
The **Architecture** of **a software product** **is critical**

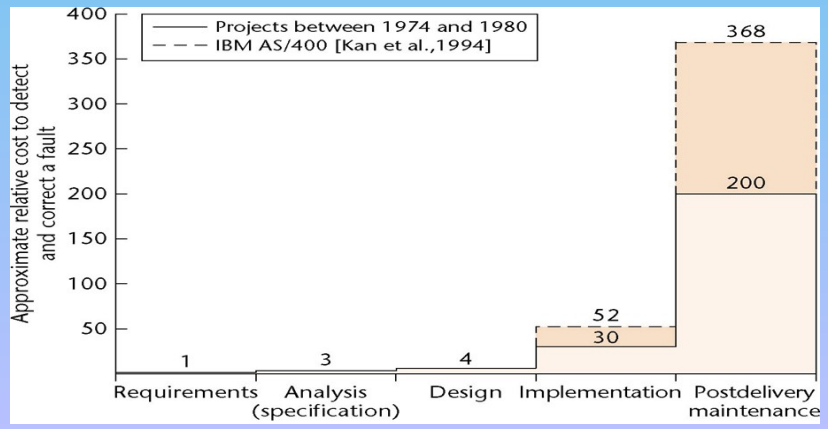
- The **Requirements** Workflow can be fixed during the **Analysis Workflow**
- The **Analysis** Workflow can be fixed during the **Design Workflow**
- The **Design** Workflow can be fixed during the **Implementation Workflow**

(BACKWARD ENGINEERING)

But there is **no way to recover from a suboptimal Architecture**

- The **Architecture** must immediately be **reDesigned**

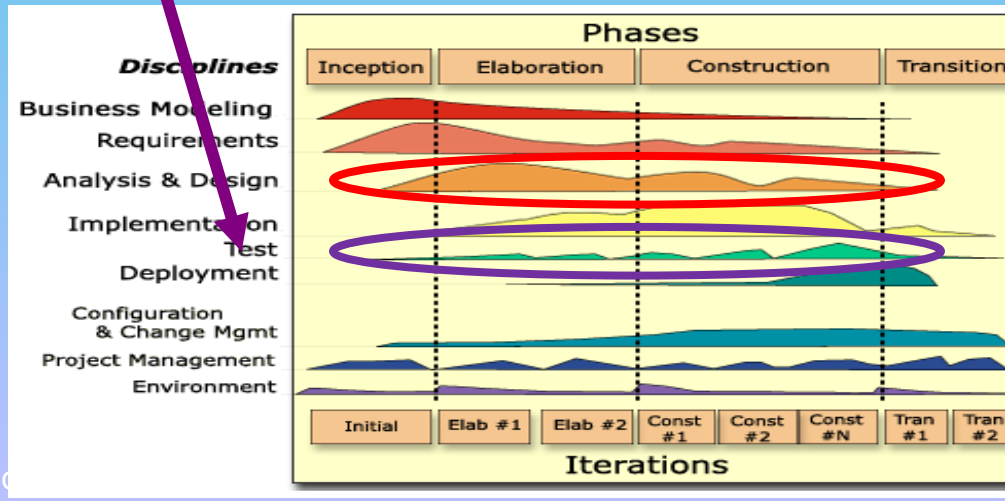




The Test Workflow: Design

Design Reviews must be performed

- The **Design** must correctly reflect the **Specifications (SRS)**
- The **Design** itself must be **correct**

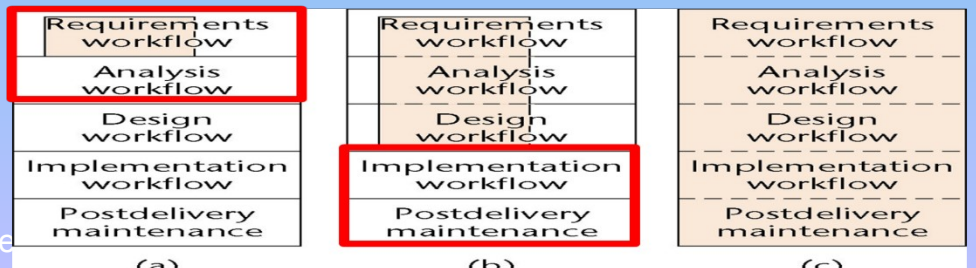


CASE Tools for **Design**

- **UpperCASE Tools**
  - Are built around a **Data Dictionary**
  - They incorporate a **Consistency Checker**, and
  - **Screen and Report Generators**

  - **Management Tools** are sometimes included, for
    - **Estimating** **– COCOMO II**
    - **Planning** **– MS Project**

- **UpperCASE** (front-end tool)
  versus
- **LowerCASE** (back-end tool)

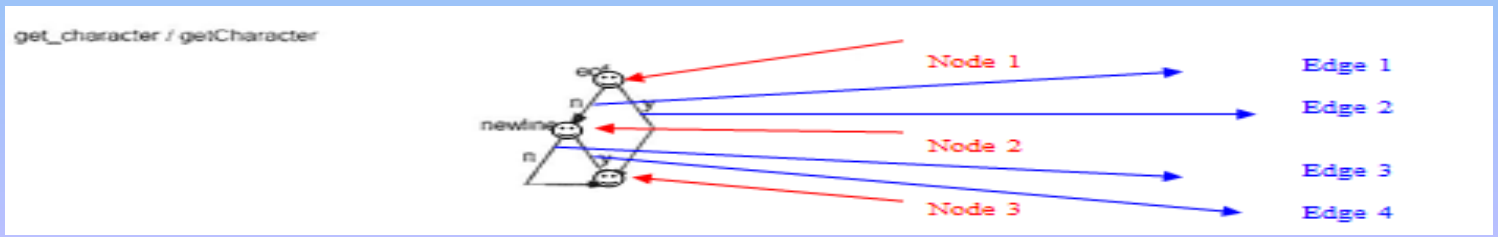


CASE Tools for Design

- Examples of tools for **Object Oriented Design**
  - Commercial Tools
    - » Software through Pictures
    - » IBM Rational Rose
    - » Together
    - » **MagicDraw**
  - Open-Source Tool
    - » **ArgoUML**

Metrics for Design

- **Measures of Design Quality**
  - **Cohesion**
  - **Coupling**
  - **Fault statistics**
  - **Cyclomatic complexity** (**Cyclomatic complexity** is computed using the control flow graph of the **module**: the **nodes** of the graph correspond to indivisible groups of commands of a **module**, and a directed edge connects two **nodes** if the second command might be executed immediately after the first command) is problematic
  - **Data Complexity** is **ignored**
    - » It is not used much with the **Object Oriented Paradigm**



Challenges of the Design Workflow

The Design Team should not do too much

- The Detailed Design (pseudocode) should not become code

```
public class EstimateFundsForWeek {

  public static void compute() {

    try {
        float expectedWeeklyInvestmentReturn = (float) 0.0; // expected weekly investment return
        float expectedTotalWeeklyNetPayments = (float) 0.0; // expected total mortgage payments less total weekly grant
        float estimatedFunds = (float) 0.0;

        Investment inv = new Investment();  // investment record
        Mortgage mort = new Mortgage();     // mortgage record

        expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment();
        expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments();

                                  1                                  -        2
        estimatedFunds = (expectedWeeklyInvestmentReturn - (MSGApplication.getAnnualOperatingExpenses() / (float) 52.0)
                + expectedTotalWeeklyNetPayments);
                                  3
        MSGApplication.setEstimatedFundsForWeek(estimatedFunds);

    } catch (Exception e) {
        System.out.println("***** Error: EstimatedFundsForWeek.compute () *****");
        System.out.println("\t" + e);
    }

    } // compute
}
```

Challenges of the Design Workflow

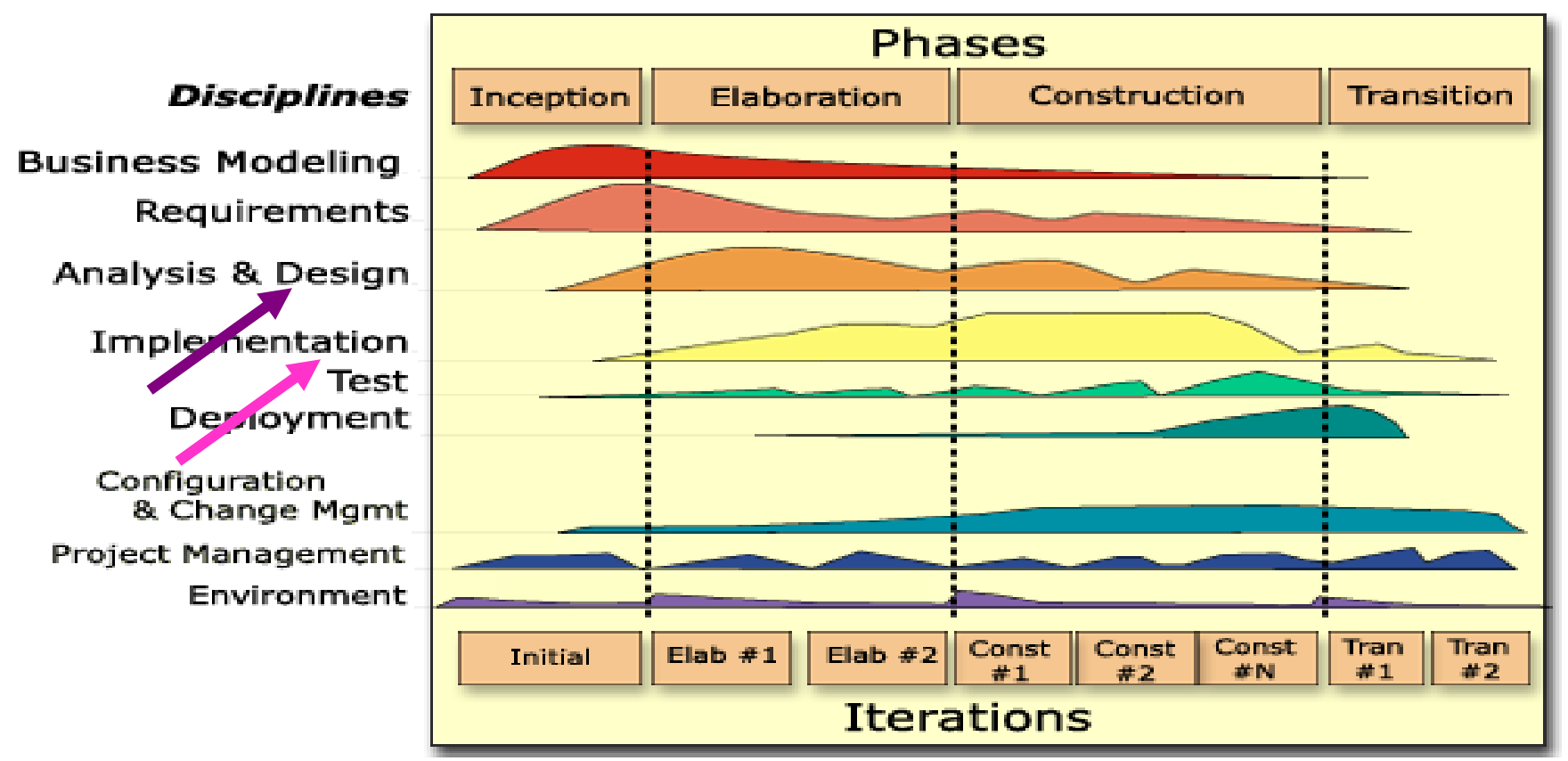
The Design Team should not do too little

- It is essential for the Design Team to produce a complete Detailed Design (pseudocode)

```
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 13.17)
   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
```

End Design – HOW "blue print" Workflow

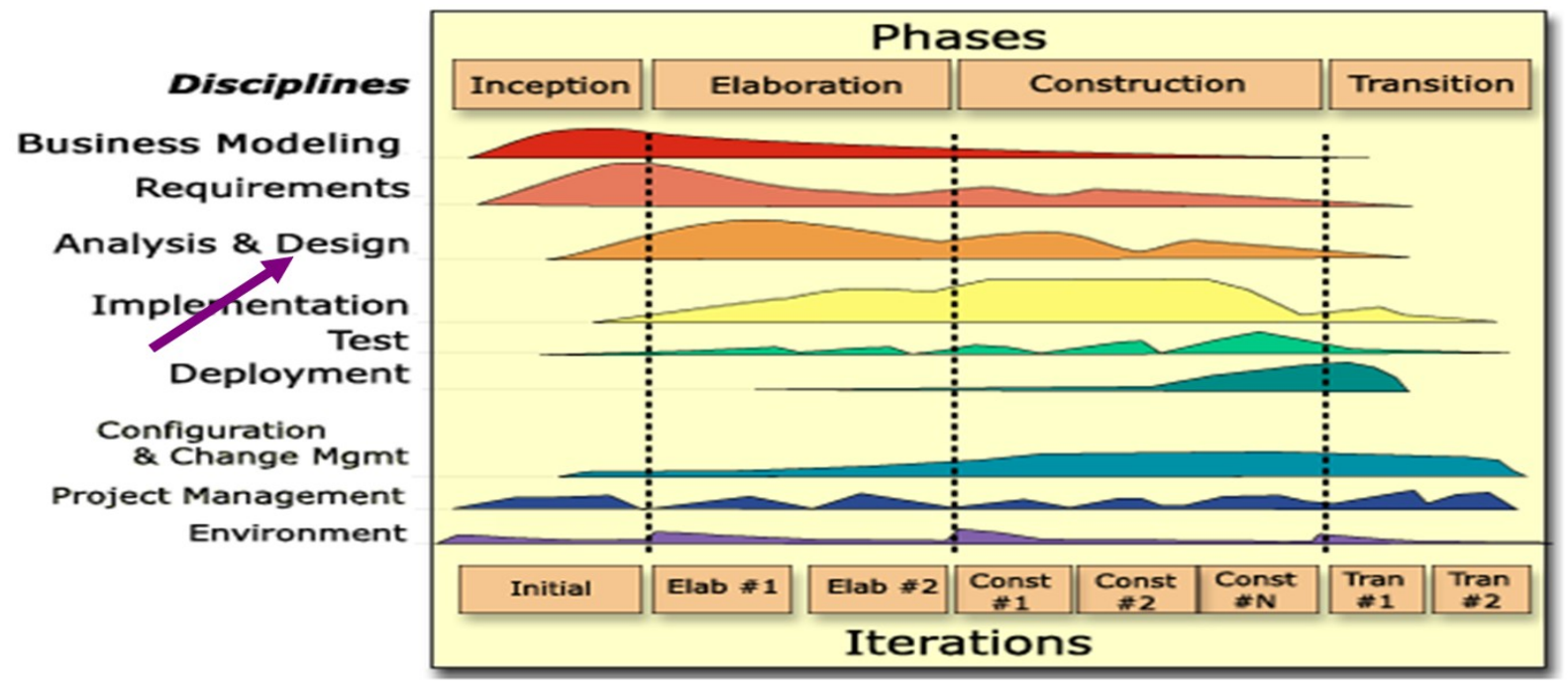
- Next Implementation Workflow



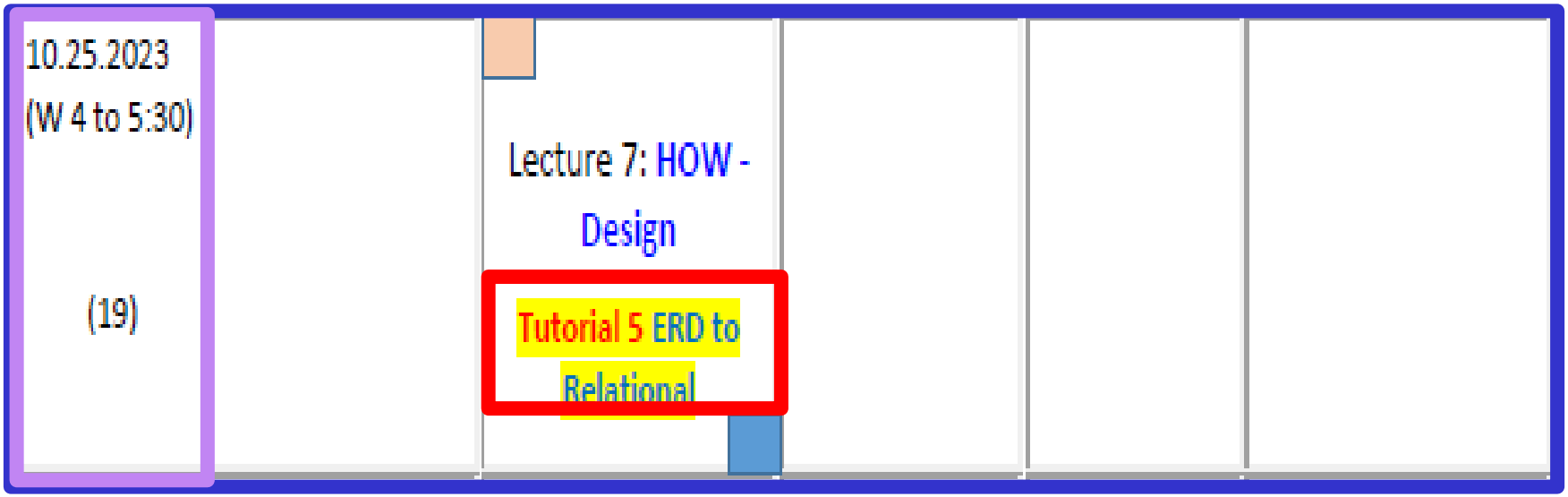
From 4:10 to 4:50 – 40 minutes.

END

Design Workflow



From 4:50 to 5:00 – 10 minutes.

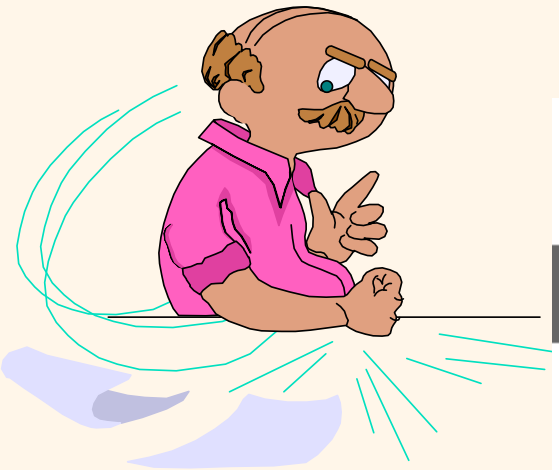


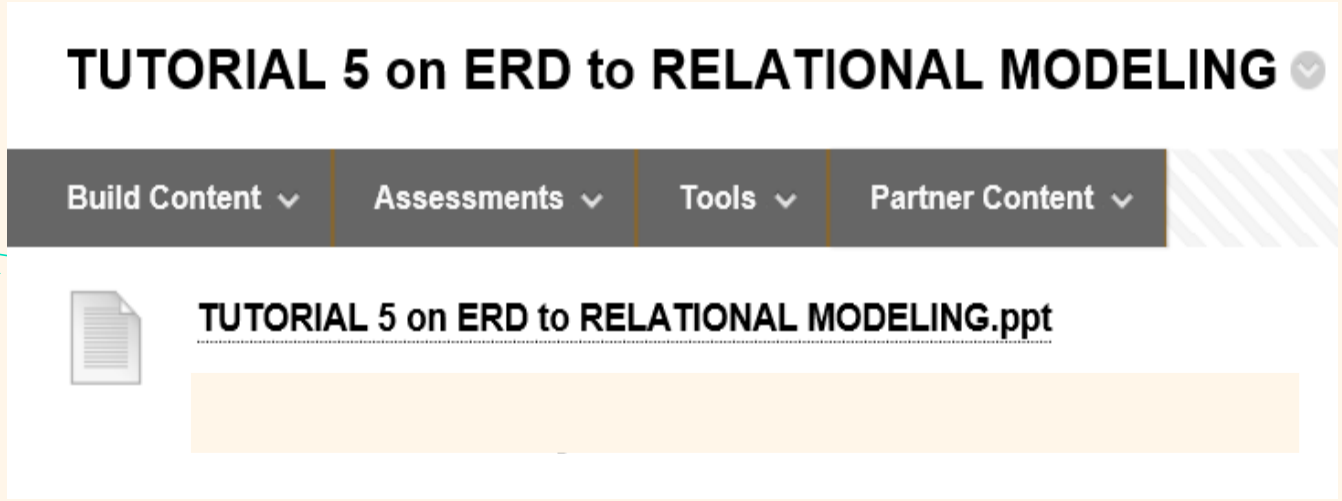
TUTORIAL 5 on ERD to RELATIONAL MODELING

ER *to*

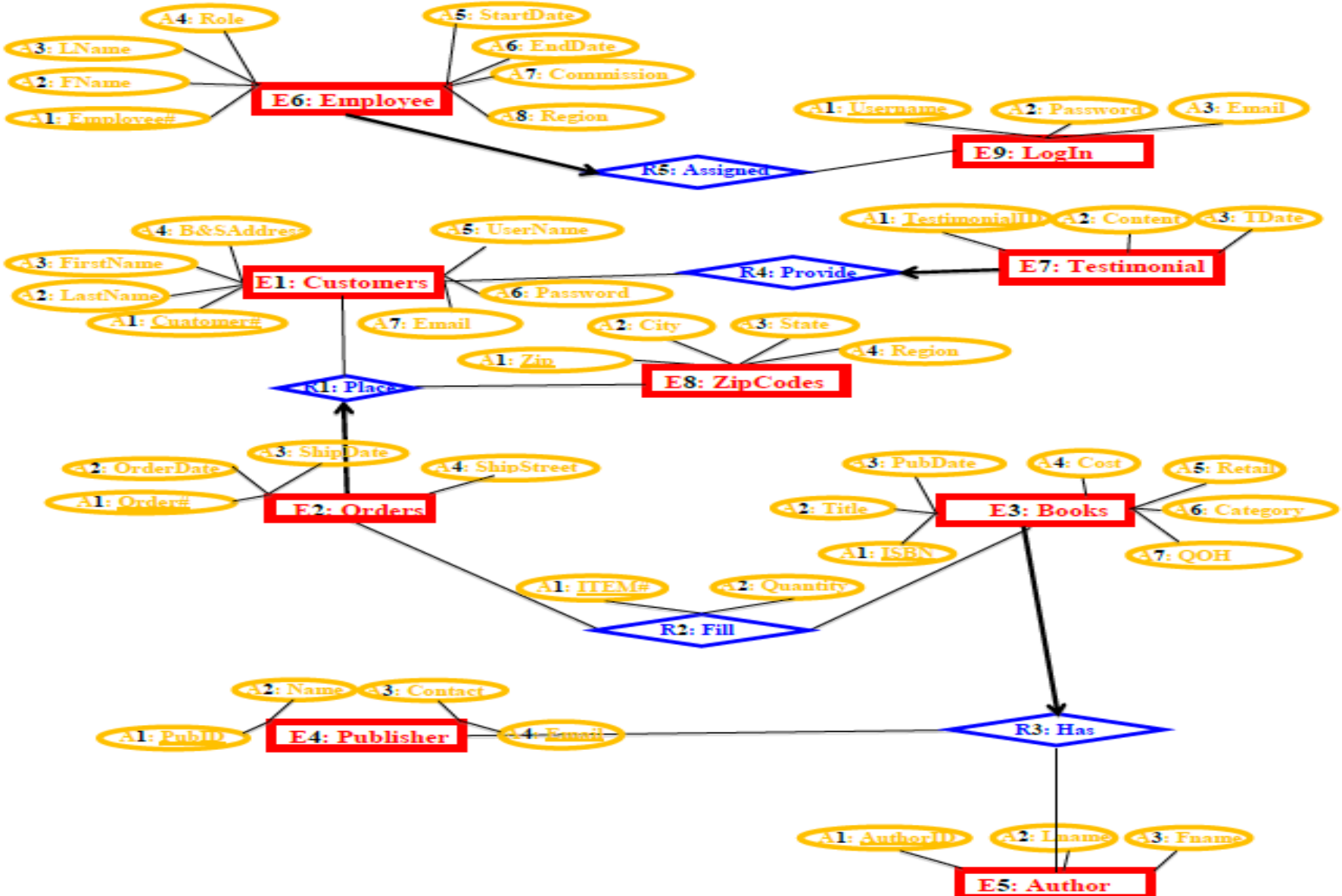
Relational

What *to* ***How***

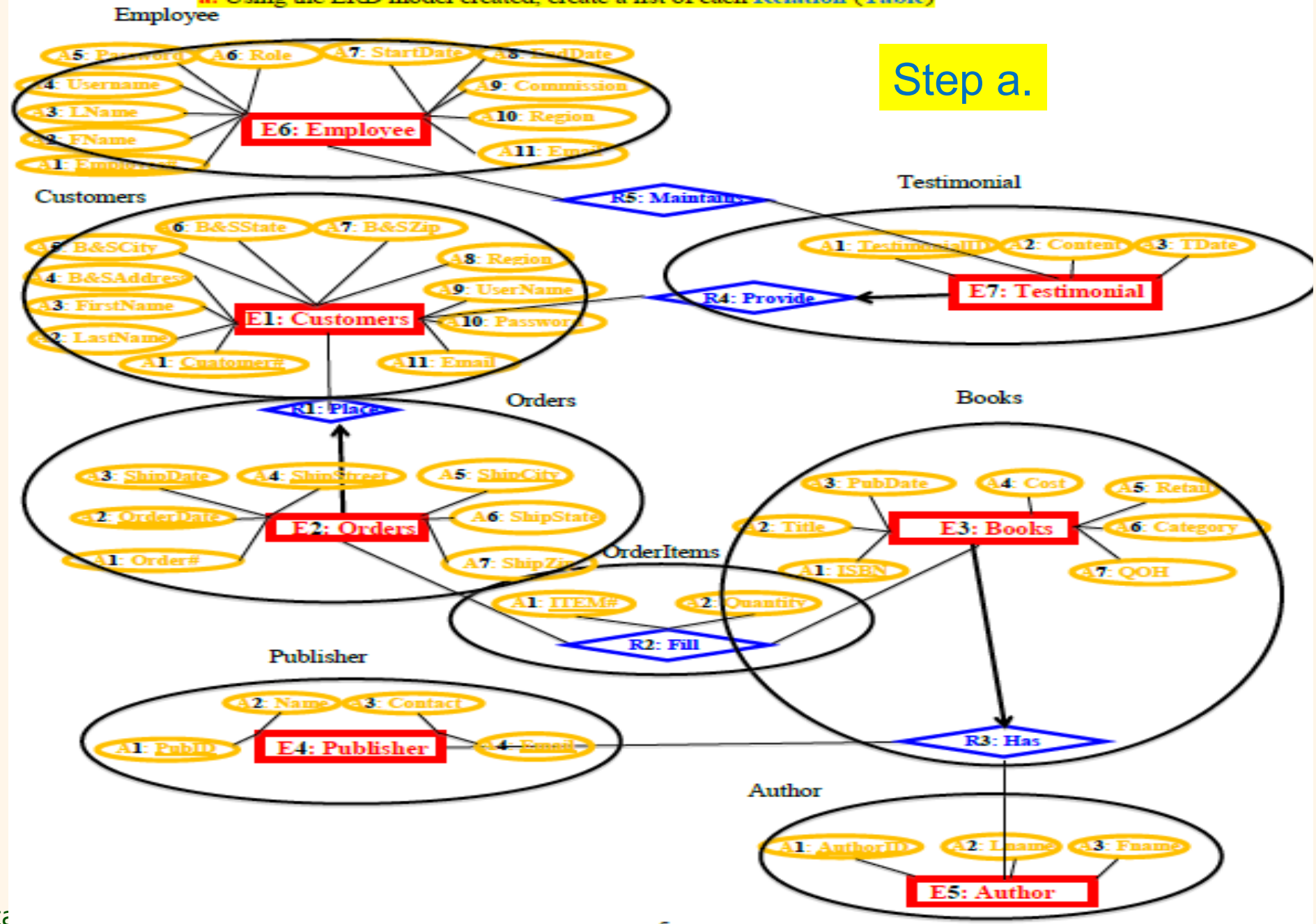




ERD Model (WHAT data)



a. Using the ERD model created, create a list of each Relation (Table)

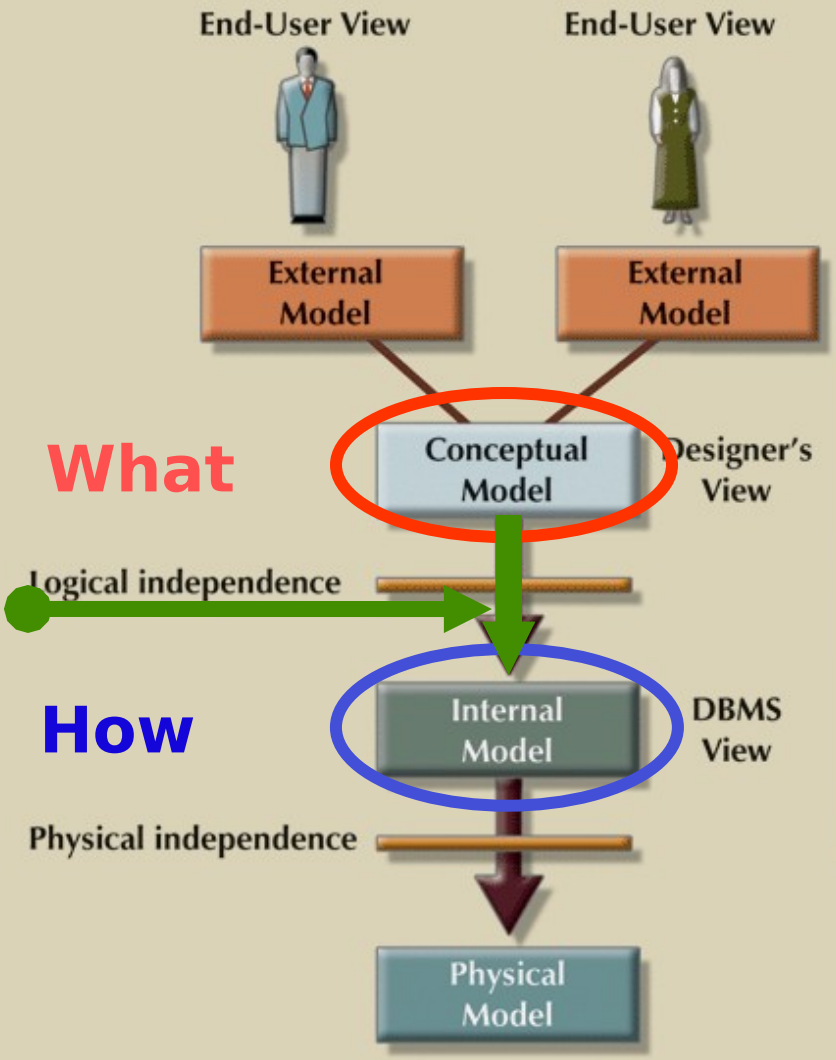


Step b.

TABLE format

```
TABLE Customer (
                    CustomerID          CHAR(20),
                    Cname               CHAR(20),
                    Address             CHAR(20),
                    State               CHAR(20),
                    City                CHAR(20),
                    PostalCode          CHAR(15),
                    Email               CHAR(20),
                    UserName            CHAR(20),
                    Password            CHAR(20),
                    PRIMARY KEY         (CustomerID))

TABLE ProductLine (LineID           CHAR(30),
                   Lname            CHAR(30),
                   PRIMARY KEY(LineID))


TABLE Order (
                OrderID             CHAR (20),
                orderDate           DATE,
                   CustomerID                CHAR(20),
                PRIMARY KEY     (OrderID),
                FOREIGN KEY     (CustomerID) REFRENCES Customer)

TABLE Product (
                  ProductID             CHAR(20),
                  Pname                 CHAR(20),
                  Finish                CHAR(20),
                  StandartPrice.        FLOAT,
                  Photo                 IMAGE,
                  LineID                CHAR(20),
                  PRIMARY KEY           (ProductID),
                  FOREIGN KEY (LineID) REFRENCES ProductLine)

TABLE OrderLine (
                    Quantity                INTEGER,
                    SalePrice.              FLOAT,
                    ProductID                   CHAR(20),
                    OrderID                 CHAR(20),
                    PRIMARY KEY                 (ProductID, OrderID),
                    FOREIGN KEY (ProductID) REFERENCES Product,
                    FOREIGN KEY (ProductLineID) REFRENCES Order)
```

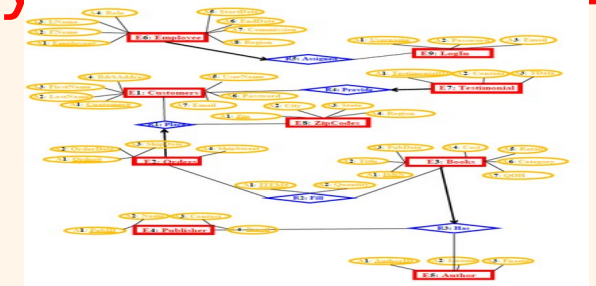
Data abstraction levels

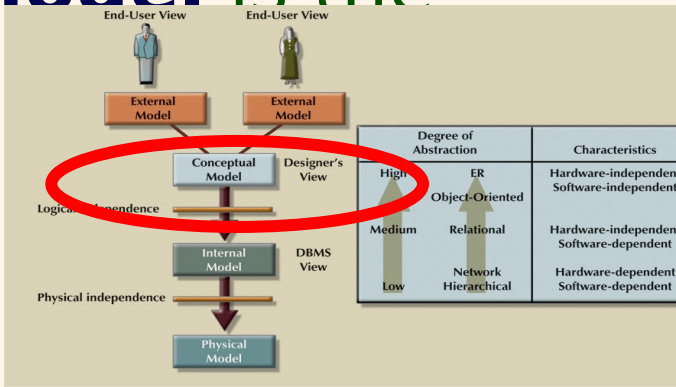


Degree of Abstraction		Characteristics
High	ER	Hardware-independent Software-independent
	Object Oriented	
Medium	Relational	Hardware-independent Software-dependent
Low	Network Hierarchical	Hardware-dependent Software-dependent

The Conceptual Model - What

- Represents global view of the entire **database**
- Representation of **data** as viewed by the entire organization
- Basis for identification and high-level description of main **data objects**, avoiding details
- Most widely used conceptual **Model** is the Entity-Relationship (ER) **Model**





The Internal Model - How

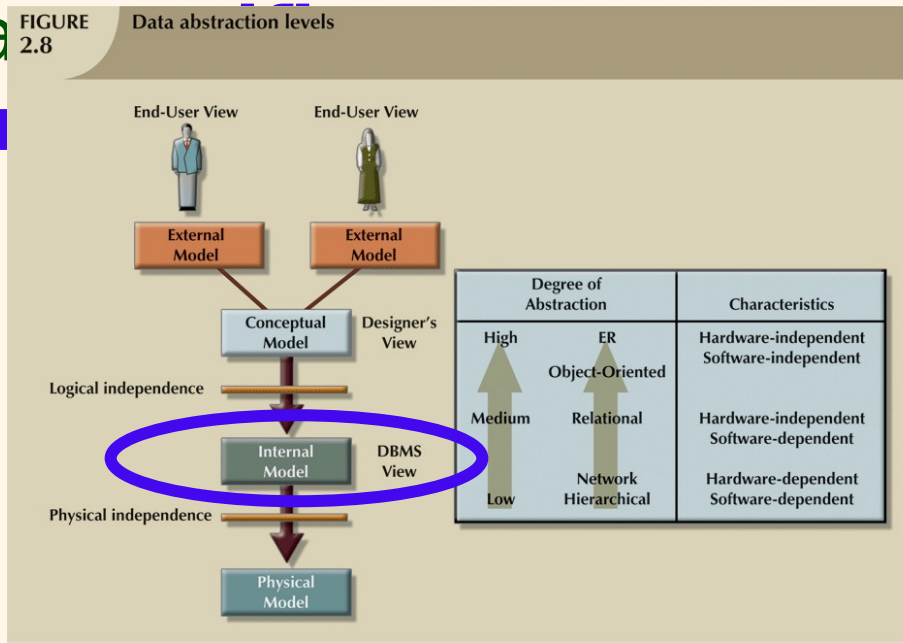
Representation of the database as “seen” by the **DBMS**

Maps the conceptual **Model** **ERD** to the **DBMS** **Relations**

Internal schema depicts a

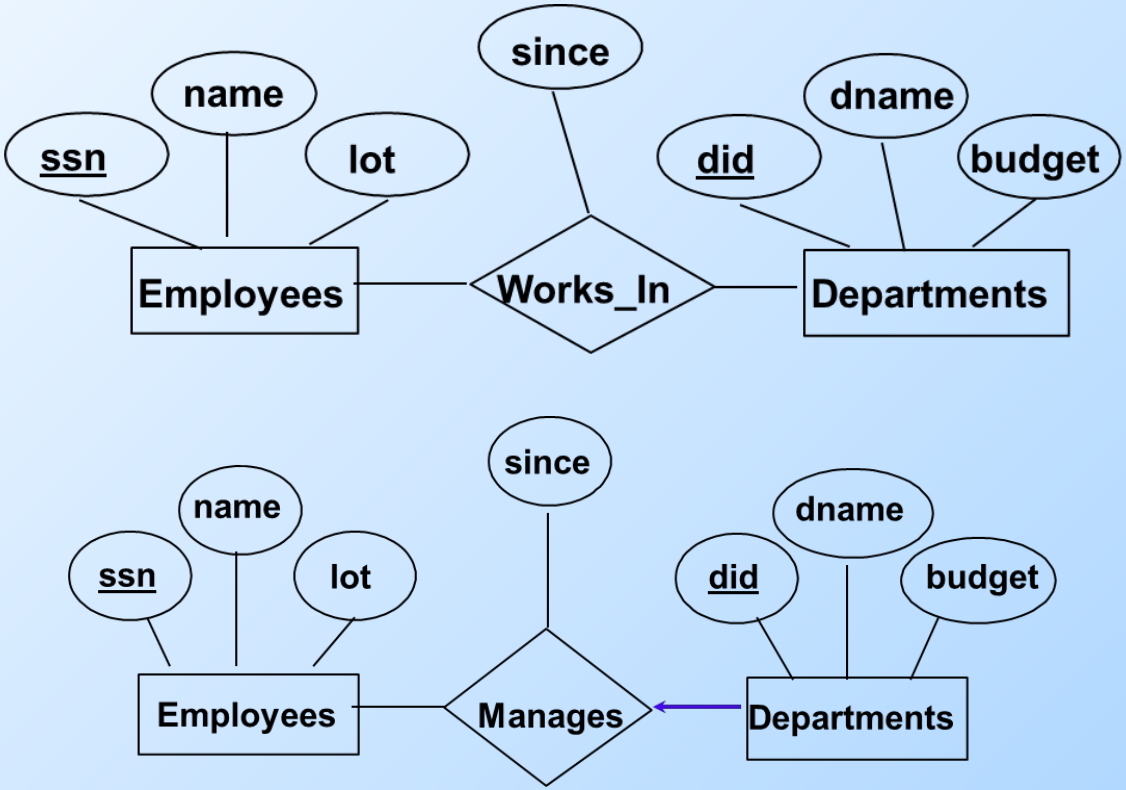
re ter

```
TABLE format

TABLE Customer (
                CustomerID          CHAR(20),
                Cname               CHAR(20),
                Address             CHAR(20),
                State               CHAR(20),
                City                CHAR(20),
                PostalCode          CHAR(15),
                Email               CHAR(20),
                UserName            CHAR(20),
                Password            CHAR(20),
                PRIMARY KEY          (CustomerID))

TABLE ProductLine (LineID           CHAR(30),
                   Lname            CHAR(30),
                   PRIMARY KEY(LineID))

TABLE Order (
              OrderID               CHAR (20),
              orderDate             DATE,
                CustomerID          CHAR(20),
              PRIMARY KEY    (OrderID),
              FOREIGN KEY    (CustomerID) REFRENCES Customer)

TABLE Product (
                ProductID           CHAR(20),
                Pname               CHAR(20),
                Finish              CHAR(20),
                StandartPrice.      FLOAT,
                Photo               IMAGE,
                LineID              CHAR(20),
                PRIMARY KEY         (ProductID),
                FOREIGN KEY (LineID) REFRENCES ProductLine)

TABLE OrderLine (
                  Quantity          INTEGER,
                  SalePrice.        FLOAT,
                  ProductID         CHAR(20),
                  OrderID           CHAR(20),
                  PRIMARY KEY       (ProductID, OrderID),
                  FOREIGN KEY (ProductID) REFERENCES Product,
                  FOREIGN KEY (ProductLineID) REFRENCES Order)
```



WHY modeling

Using this ERD Language



HOW Modeling

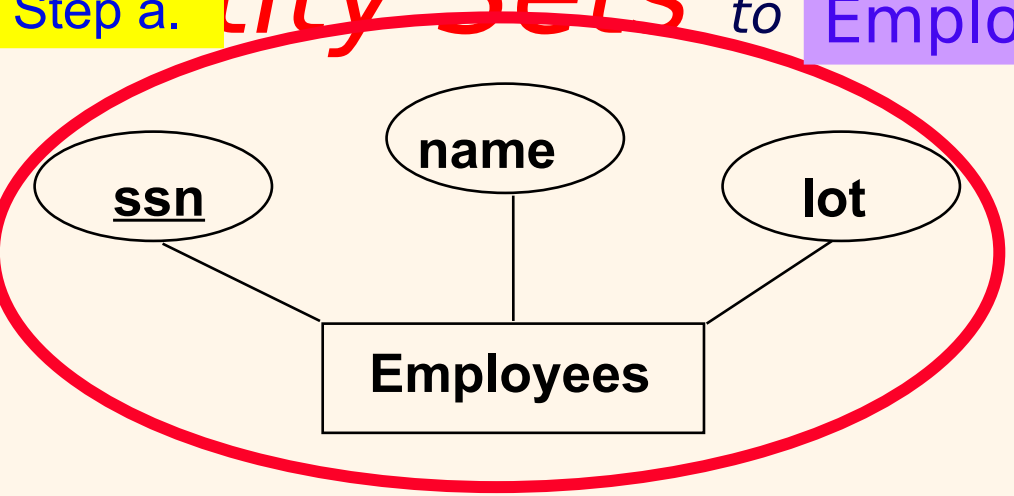
Using this **Relational** Language

```
TABLE Employees (ssn CHAR(11),
                 name CHAR(30),
                 lot INTEGER,
                 PRIMARY KEY (ssn))

TABLE Departments (did INTEGER,
                   dname CHAR(20),
                   budget REAL,
                   PRIMARY KEY (did))
```

Logical DB Design: ER to **Relational**

Entity Sets to Relations:

Step a.



ssn	name	lot
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

Figure 3.9 An Instance of the Employees Entity Set

```
CREATE TABLE Employees ( ssn      CHAR(11),
                         name     CHAR(30),
                         lot      INTEGER,
                         PRIMARY KEY (ssn) )
```

Step b.

How many **Relations** we will have?

1

Relational Relationship sets to Relations:

- In translating a Relationship set to a **Relation**, attributes of the **Relation** must include:
  - Keys for each participating *Entity set* (as foreign keys).
    - This set of attributes forms a *key* for the **Relation**.
  - All descriptive attributes (attributes of the Relationship set!).

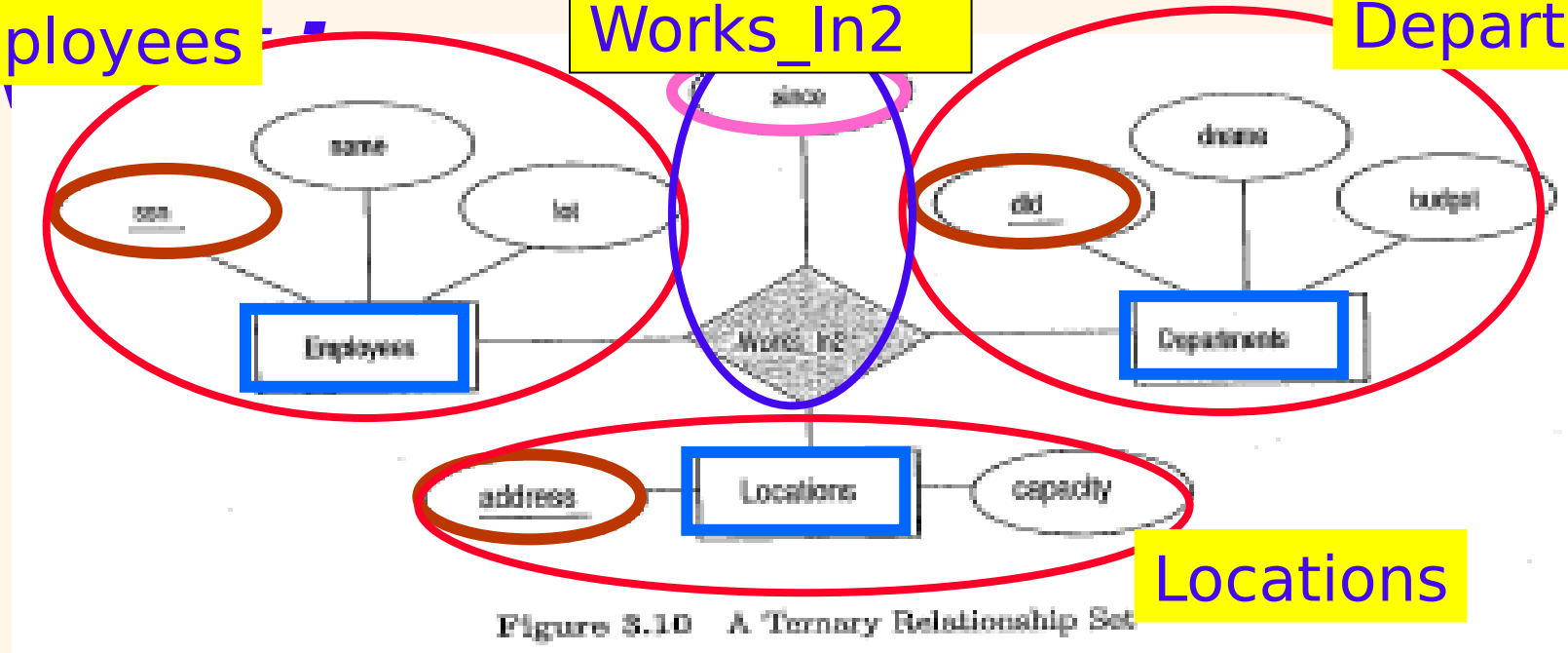
Relational

Relationship sets to

Employees

Works_In2

Departments



Locations

Figure 3.10 A Ternary Relationship Set

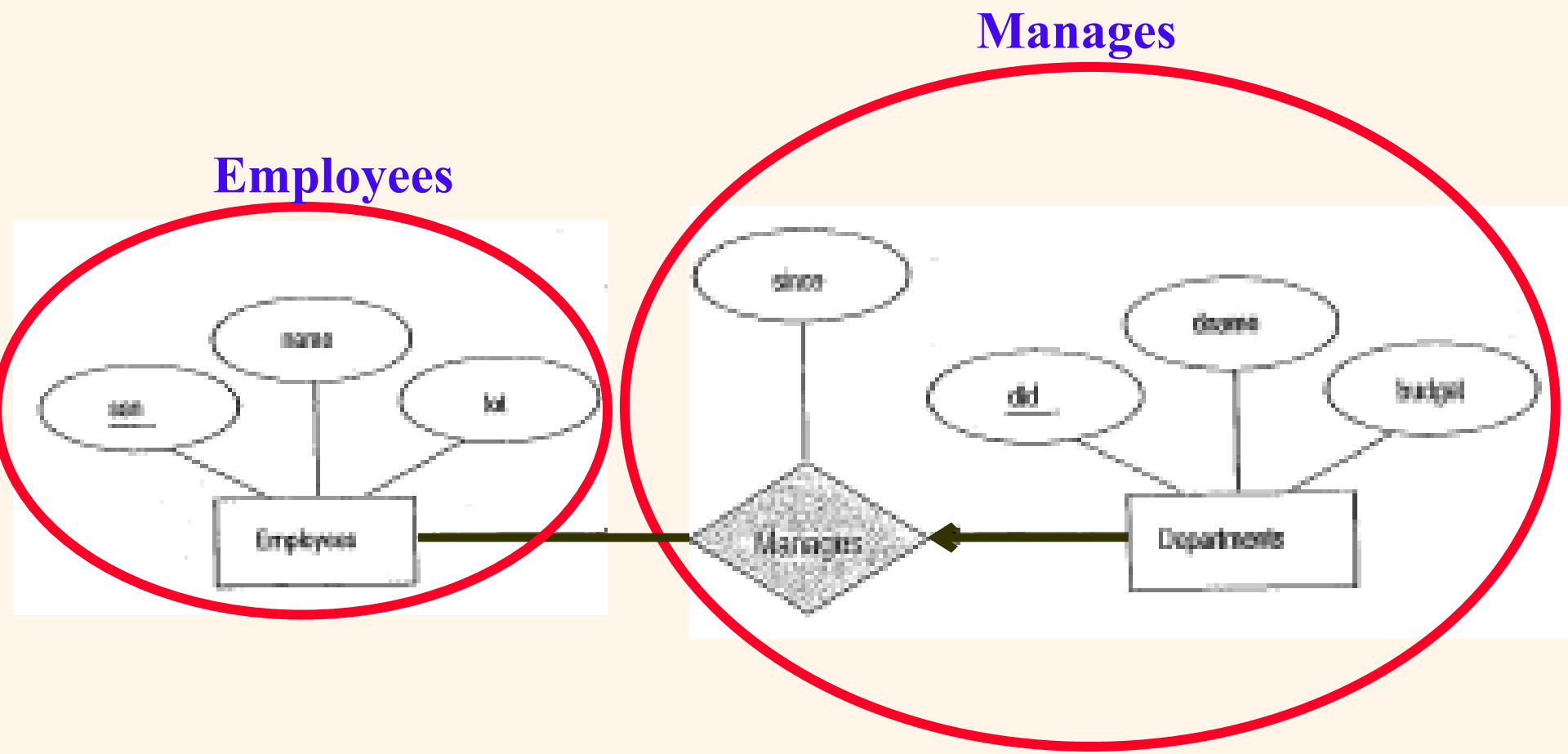
```
TABLE Works_In2 ( ssn       CHAR(11),
                  did       INTEGER,
                  address   CHAR(20),
                  since     DATE,
                  PRIMARY KEY (                    ),
                  FOREIGN KEY (    ) REFERENCES Employees,
                  FOREIGN KEY (      ) REFERENCES Locations,
                  FOREIGN KEY (    ) REFERENCES Departments )
```

- In translating a Relationship set to a **Relation**, attributes of the **Relation** must include:
  - Keys for each participating *Entity set* (as foreign keys).
    - This set of attributes forms a *key* for the **Relation**.
  - All descriptive attributes (attributes of the Relationship set!).

How many Relations? 4

Translating ER Diagrams with Key Constraints

- Since *each* ***Department*** *has a unique Manager*, we must instead combine **Manages** and **Departments**.



How many **Relations** we will have? 2

Translating ER Diagrams with Key Constraints

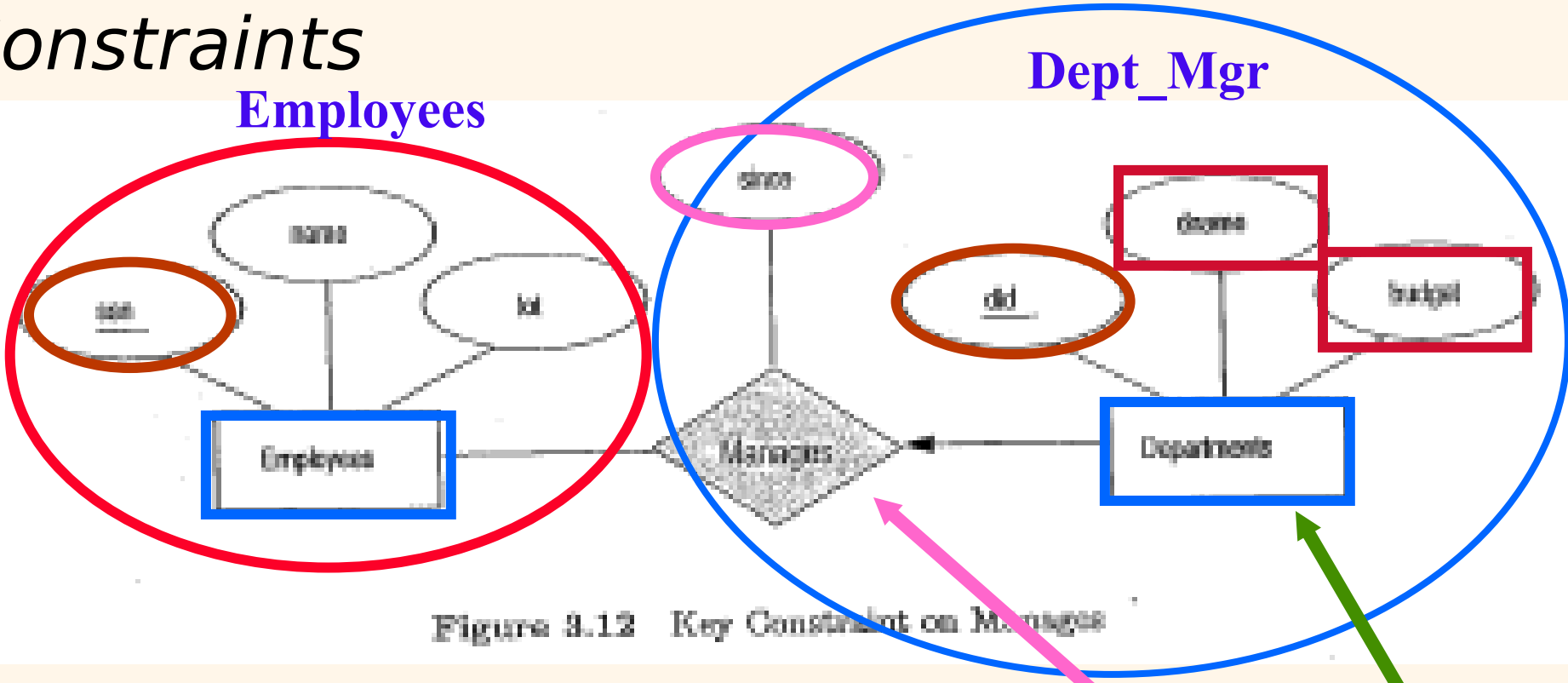


Figure 3.12 Key Constraint on Manages

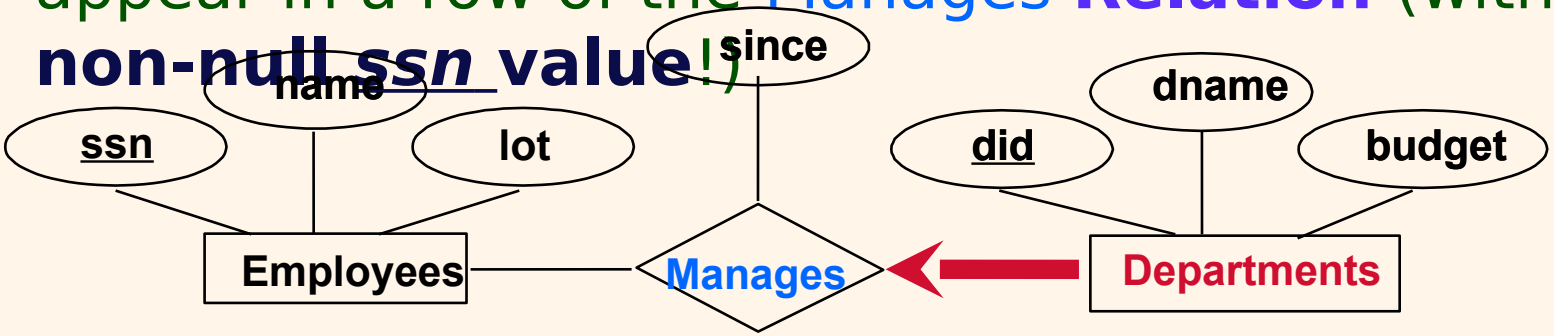
```
TABLE Dept_Mgr ( did       INTEGER,
                 dname     CHAR(20),
                 budget    REAL,
                 ssn       CHAR(11),
                 since     DATE,
                 PRIMARY KEY (did),
                 FOREIGN KEY (ssn) REFERENCES Employees )
```

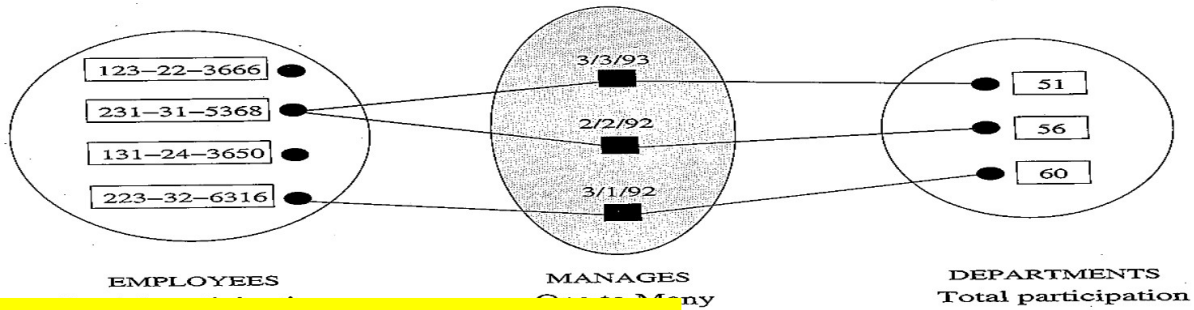
The idea is to include information about the Relationship set in the **Relation** corresponding to the Entity set with the key, taking advantage of the key constraint!

How many **Relations** we will have? 2

Review: Participation Constraints

- Does every Department have a Manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be ***total*** (vs. *partial*).
    - Every *did* value in Departments **Relation** must appear in a row of the Manages **Relation** (with a **non-null *ssn* value!**)

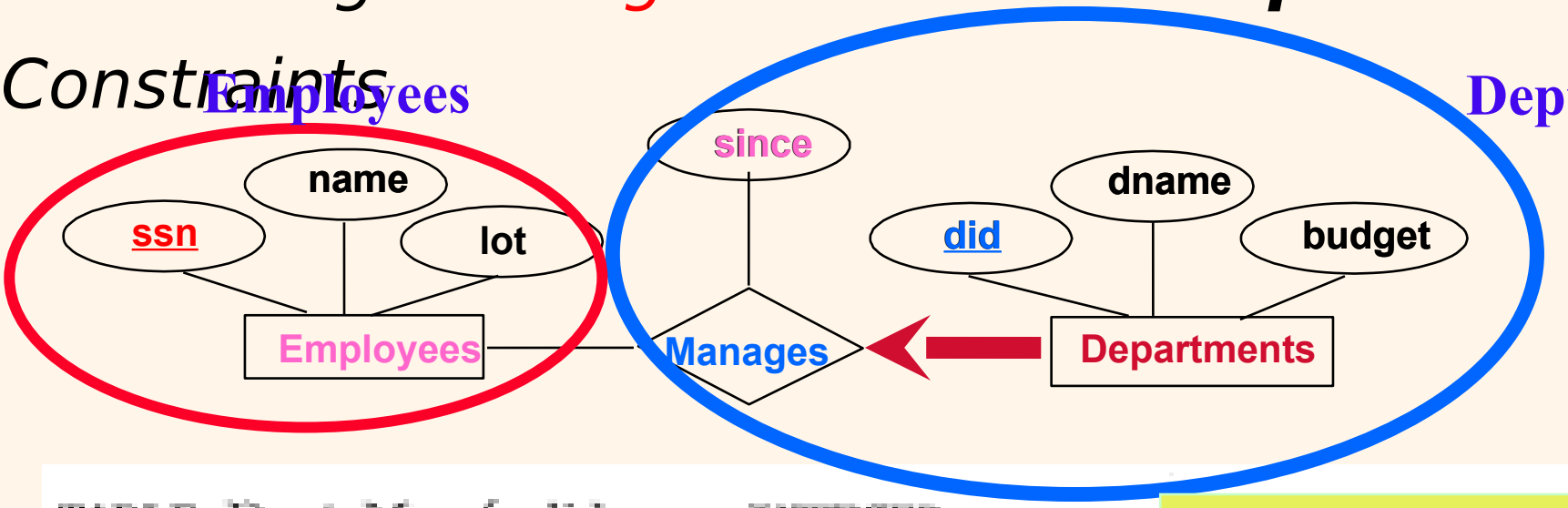




Manages Relationship Set

How many **Relations** we will have? 2

Constraints




```
TABLE Dept_Mgr ( did       INTEGER,
                 dname     CHAR(20),
                 budget    REAL,
                 ssn       CHAR(11) NOT NULL,
                 since     DATE,
                 PRIMARY KEY (did),
                 FOREIGN KEY (ssn) REFERENCES Employees
                       ON DELETE NO ACTION )
```

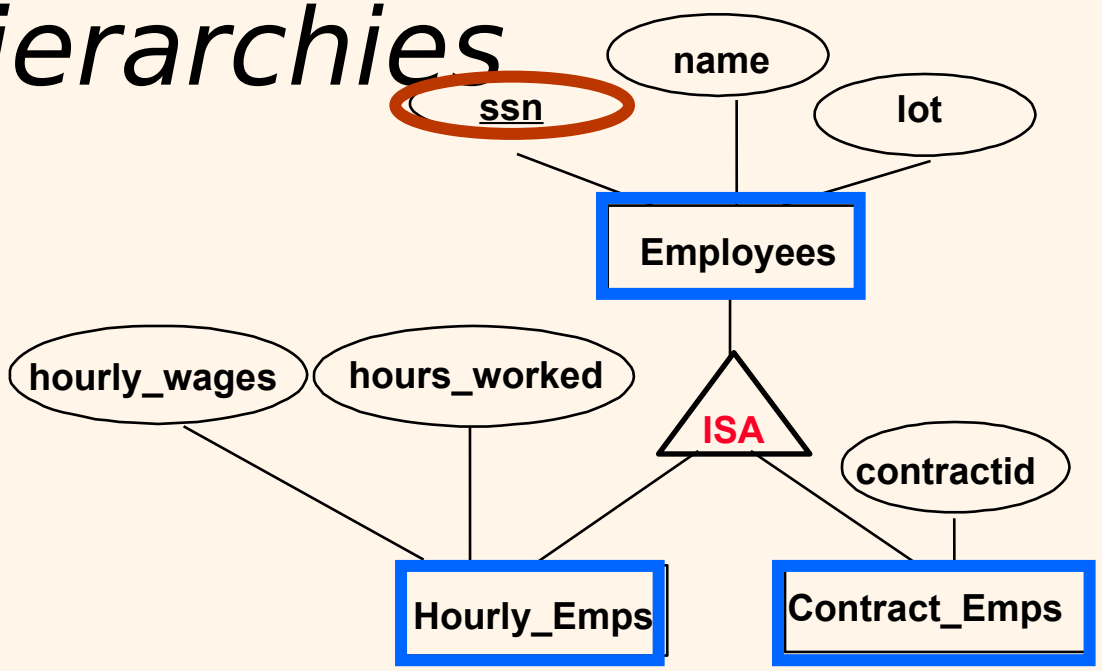
Can ssn be NULL?

Every Department must have a Manager!

Because ssn cannot take NULL values, each tuple of **Dept_Mgr** identifies a tuple in **Employees** (who is the manager).

The ON DELETE NO ACTION ensures that an **Employee** tuple cannot be deleted while it is pointed to by a **Dept_Mgr** tuple.

Review: ISA Hierarchies



- When A **ISA** B, every A entity is also considered to be a B entity.
  - *Overlap constraints*: Can Joe be an **Hourly_Emps** as well as a **Contract_Emps** entity? (*Allowed/disallowed*)
  - *Covering constraints*: Does every **Employees** entity also have to be an **Hourly_Emps** or a **Contract_Emps** entity? *(Yes/no)*

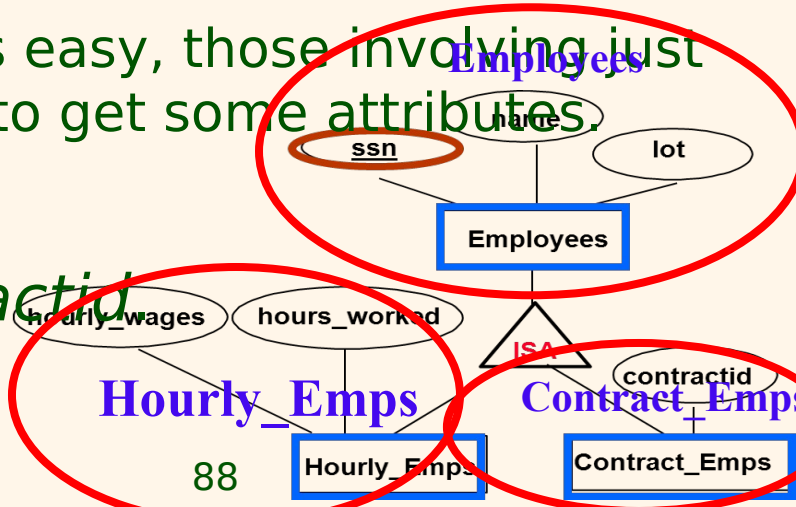
How many **Relations** we will have?

2 **or** 3

Translating ISA Hierarchies to Relations

General approach:

- 3 **Relations**: ***Employees***, ***Hourly_Emps***, and ***Contract_Emps***.
  - ***Employees*** : ***ssn***, *name, lot*.
  - ***Hourly_Emps***: ***ssn***, *hourly_wages*, *hours_worked*.
    - extra info recorded in **Hourly_Emps**.
    - must delete **Hourly_Emps** tuple if referenced **Employees** tuple is deleted.
    - queries involving all employees easy, those involving just **Hourly_Emps** **require a join** to get some attributes.
  - ***Contract_Emps***: ***ssn***, *contractid*



Translating ISA Hierarchies to Relations

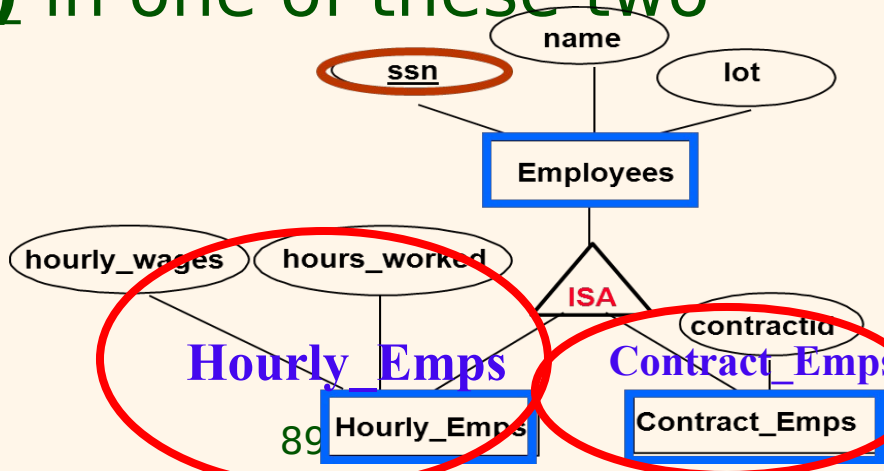
Alternative:

- 2 **Relations**: ***Hourly_Emps*** and ***Contract_Emps***.

  - ***Hourly_Emps***: ***ssn***, ***name***, ***lot***, *hourly_wages, hours_worked.*

  - ***Contract_Emps***: ***ssn***, ***name***, ***lot***, *contractid.*

Each employee **must be(!)** in one of these two subclasses.



a. Using the ERD model created, create a list of each Relation (Table)

Step a.

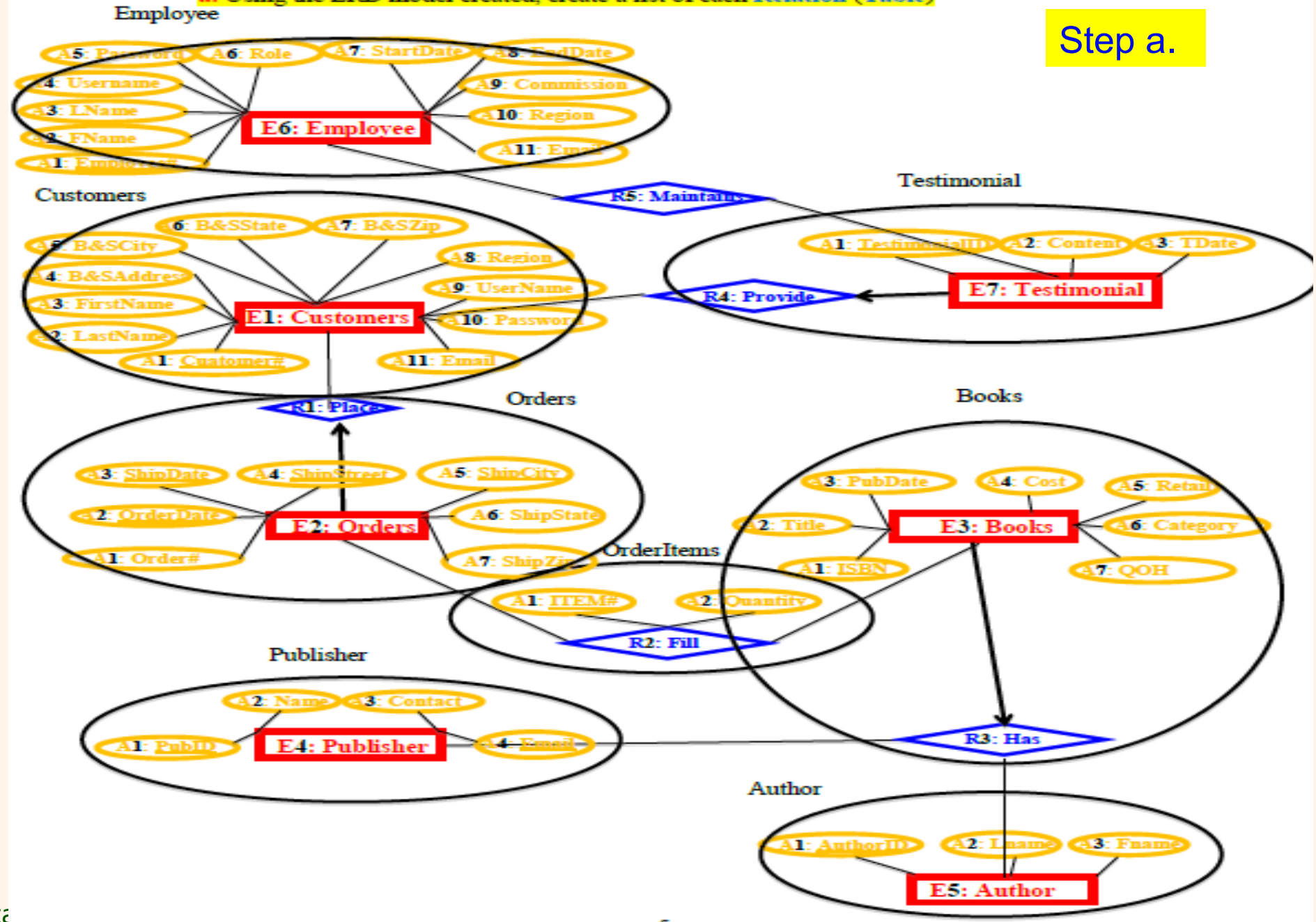


TABLE format

Step b.

```
TABLE Customer (
                    CustomerID          CHAR(20),
                    Cname               CHAR(20),
                    Address             CHAR(20),
                    State               CHAR(20),
                    City                CHAR(20),
                    PostalCode          CHAR(15),
                    Email               CHAR(20),
                    UserName            CHAR(20),
                    Password            CHAR(20),
                    PRIMARY KEY         (CustomerID))

TABLE ProductLine (LineID           CHAR(30),
                   Lname            CHAR(30),
                   PRIMARY KEY(LineID))


TABLE Order (
                OrderID             CHAR (20),
                orderDate           DATE,
                  CustomerID              CHAR(20),
                PRIMARY KEY     (OrderID),
                FOREIGN KEY     (CustomerID) REFRENCES Customer)

TABLE Product (
                  ProductID             CHAR(20),
                  Pname                 CHAR(20),
                  Finish                CHAR(20),
                  StandartPrice.        FLOAT,
                  Photo                 IMAGE,
                  LineID                CHAR(20),
                  PRIMARY KEY           (ProductID),
                  FOREIGN KEY (LineID) REFRENCES ProductLine)

TABLE OrderLine (
                    Quantity                INTEGER,
                    SalePrice.              FLOAT,
                    ProductID                   CHAR(20),
                    OrderID                 CHAR(20),
                    PRIMARY KEY                 (ProductID, OrderID),
                    FOREIGN KEY (ProductID) REFERENCES Product,
                    FOREIGN KEY (ProductLineID) REFRENCES Order)
```

Sent: Sunday, February 03, 2013 2:15 PM
To: Hilford, Victoria

Hi Dr. H,

I had to ask my brother about the details. I remembered telling you about it, but didn't remember the scenario completely.

The contract was to sell a particular data set to the FBI. The contract value was $70,000 per month, $840k per year. This type of contract can be renewed or extended for years and years.

The data model was to be constructed per the specifications of the bureau and joined with one or more other data sets. Care was not taken to review the data model, so it did not reflect the process the bureau was modeling. When I told you about it, my brother thought they were going to be able to correct it and eventually make the sale, with four to six months of delay from the original starting date. They were not successful in their effort. The contract was cancelled. The economic damage was a minimum of $840,000 for a very small firm.

From 4:50 to 5:00 – 10 minutes.

THE END

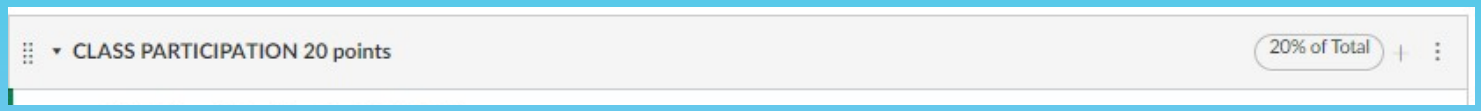
VH, next EXAM 3 REVIEW

VH, next Q & A

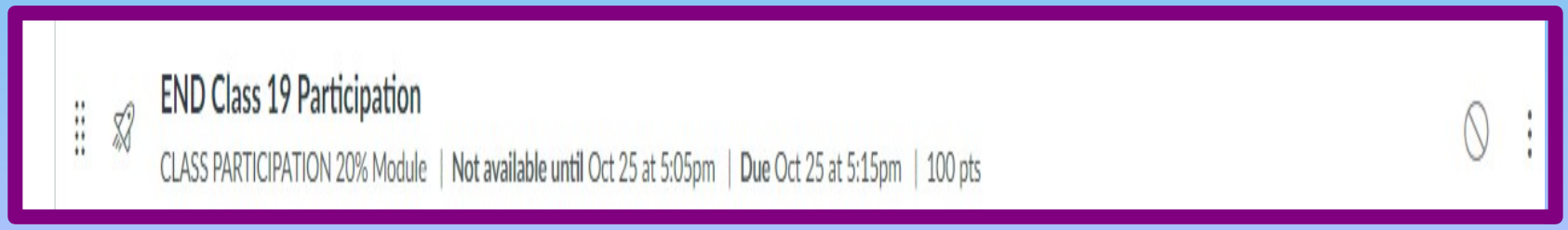
10.30.2023 (M 4 to 5:30) (20)		EXAM 3 REVIEW (CANVAS)	Download ZyBook: Sections 10-11		
11.01.2023 (W 4 to 5:30) Optional (21)					Q & A Set 3 topics.
11.06.2023 (M 4 to 5:30) (22)					EXAM 3 (CANVAS)

From 5:05 to 5:15 – 10 minutes.

10.25.2023 (W 4 to 5:30) (19)		Lecture 7: HOW - Design Tutorial 5 ERD to Relational			



PASSWORD: I AM IN TEAMS



VH– Publish.

At 5:15 PM.

End Class 19

VH, Download Attendance Report
Rename it:
10.25.2023 Attendance Report FINAL

VH, upload Class 19 to CANVAS