
Software Engineering

Lecture 1

Instructor : Conf. dr. Cristina Mîndruță
cristina.mindruta@e-uvt.ro

Sites:

<http://sites.google.com/site/ingswcm>

As a *student* with no other experience than having done *some programming*, it is quite *difficult to understand what more is involved in software engineering*.

Typically, when creating a program in a *course setting*, the *exercise* starts from an *idea* that may have been *explained in a few words*: say, less than one hundred words.

Based on the idea, the student and his classmates *developed a piece of software*, meaning they *wrote code and made sure that it worked*. After the assignment they *didn't need to take care of it*. These assignments were *small* and to perform them they really *did not need much engineering discipline*.

This situation is quite unlike what you have to do in the *industry*, where *code written will stay around for years*, passing through *many hands to improve it*. Here a *sound approach to software engineering* is a must. Otherwise, it would be impossible to *collaborate* and *update* the software with *new features* and *bug fixes*.

Nevertheless, the experience in school is an important and essential beginning.

Topics covered

Issues in *developing* and *supporting* software products

A simple program

Size and complexity of a system

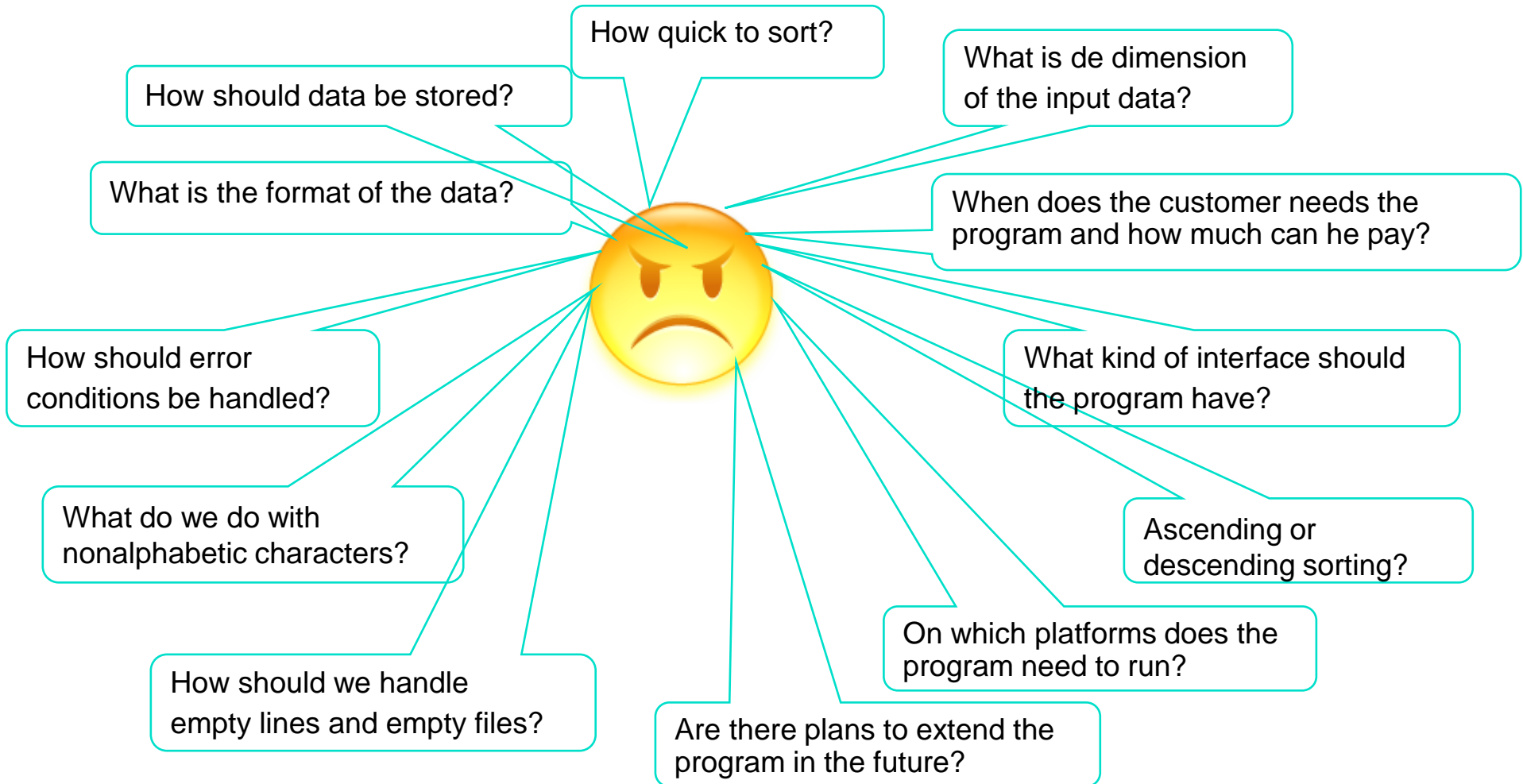
Technical and non-technical issues

Concerns in case of a large application

Coordination efforts for process, product and people

A simple program

“Given a collection of lines of text (strings) stored in a file, sort them in alphabetical order, and write them to another file”.



Requirements

Clarify the REQUIREMENTS.

Program requirements : statements that *define* and *qualify* what the program needs to do.

Functional requirements : What a program needs to do.

Quality attributes (extra-functional requirements) : The manner in which the functional requirements need to be achieved.

Design constraints : Statements that constrain the ways in which the software can be designed and implemented.

How quick to sort?

Are there plans to extend the program in the future?

What kind of interface should the program have?

What is the dimension of the input data?

On which platforms does the program need to run?

Sorting. Ascending or descending sorting?

What do we do with nonalphabetic characters?

How should we handle empty lines and empty files?

Requirements

Functional requirements : What a program needs to do.

Define program functionality.

Quality attributes (extra-functional requirements) : The manner in which the functional requirements need to be achieved.

Typical examples: performance, modifiability, usability, configurability, reliability, availability, security, scalability.

Design decisions and constraints : Statements that constrain the ways in which the software can be designed and implemented.

Choice of:

- Programming language

- Platforms the system runs on

- Interfaces to user and to external systems

- Tools to be used

- etc.

A simple program – the thinking process

Functional requirements : What a program needs to do. Define program functionality.

Input formats:

What is the format of the data? ASCII, UNICODE, hybrid?

How should data be stored: what separates the lines of a file?

Decision: ASCII; CR+LF

Sorting:

Ascending or descending sorting?

What do we do with nonalphabetic characters? Numbers before or after letters? Lowercase vs. uppercase.

Decision: sorting characters in numerical order, sorting file in ascending order.

Special cases, boundaries, error conditions:

How should we handle empty lines and empty files?

How should error conditions be handled?

Decision: no special treatment for empty line; empty output file created for empty input file; all errors reported to the user; presumption: input file is not corrupted.

A simple program – the thinking process

Quality attributes (extra-functional requirements) : The manner in which the functional requirements need to be achieved.

Typical examples: performance, modifiability, usability, configurability, reliability, availability, security, scalability.

Performance:

Q: How quick to sort?

A: Less than 1 min. to sort a file of 100 lines of 100 characters each.

Decision: an algorithm with a proper response time, selected based on the analysis of existing sorting algorithms.

Modifiability :

Q: Are there plans to extend the program in the future?

A: Changes in sorting order may be requested in the future.

Decision: Prepare the program for future changes in sorting order.

A simple program – the thinking process

Design decisions and constraints : Statements that constrain the ways in which the software can be designed and implemented.

Platforms:

DECISION on architecture, OS, available libraries.

Portability is more or less limited \Rightarrow extra cost implied by supporting a new platform \Rightarrow
TRADEOFF between developing for portability and the predicted need for future portability.

Schedule requirements:

Technical people inform about feasibility and cost for different deadlines.

The client establishes the final deadline.

User interface: CLI, GUI, Web-based ?

Decision:

Avoid upload and download \Rightarrow not Web-based

Allow for automation and REUSE program as a module in future applications

\Rightarrow invokable inside a script

\Rightarrow CLI

A simple program – the thinking process

Design decisions and constraints : Statements that constrain the ways in which the software can be designed and implemented.

Typical and maximum input sizes:

Small \Rightarrow any sorting algorithm \Rightarrow the simplest to implement

Large, fit in RAM \Rightarrow an efficient algorithm

Large, not fit in RAM \Rightarrow specialized algorithm for on-disk sorting

Programming language:

May be a design constraint.

May be a design decision based on type of programming needed, the performance and portability requirements, the technical expertise of the developers.

Algorithms:

May be given as design constraint or as functional requirements.

May be design decisions influenced by:

- the language used and the library available – we may use a standard facility offered by the programming language

- required performance – trade-off with effort required to implement and the expertise of the developers

A simple program – Testing

Testing – dynamic verification of the written code.

Levels of testing

- Unit testing

- Integration testing

- Acceptance testing

Unit testing – process followed by a programmer to test each piece or unit of software.

Programmer:

- Writes code

- Writes tests to check each module, function or method.

XP – programmers write tests before writing the code.

A simple program – Estimating effort

Estimating effort - important aspect of a software project.

Effort estimation \Rightarrow *cost* estimation, *schedule* estimation.

Test – 1 minute:

Estimate how much time do you need to write a program that read lines from one file and write the sorted lines to another file, using your favorite language. You must implement the sort algorithm and to offer a GUI with two text boxes and two buttons in which the user can select the input file and the output file using the File Open dialog. Assume you can work only on this one task, with no interruptions.

Is realistic the assumption to work with no interruptions and only to this task?

A simple program – Estimating effort

Divide the task into subtasks:

Create class `StringSorter` with three public methods: `Read`, `Write`, `Sort`

Implement sorting routine using an algorithm that involves finding the largest element, putting it at the end of array, and then sorting the rest of the array using the same mechanism

⇒ create a method `IndexOfBiggest` that returns the index of the biggest element in an array.

Homework :

Estimate an *ideal time* (asap) and a *calendar time* for each task in the following list:

`IndexOfBiggest`

`Sort`

`Read`

`Write`

`GUI`

`Testing`

Implement the solution and compare estimated with actual time.

A simple program – Estimating effort

Conclusion:

Estimation is more accurate after dividing the work into tasks

Estimation will differ more or less from the actual effort

Estimation benefits from previous experience

Estimation is less accurate at the beginning of the project.

Estimation – important issue in *software project management*.

A simple program – Implementations

Language-independent GUIDING RULES:

Be *consistent* in names, capitalization, programming conventions

Try to follow the established *conventions of the programming language*

(ex. Java: class name-uppercase, variable name-lowercase, separate words with uppercase; C:use lowercase and separate words with underscore)

Chose *descriptive* names; long for elements that have a global scope (ex. classes, public methods), short for local references (ex. local variables, private names).

The *methods/procedures/functions used* in the module you develop must be previously proved to *work* \Rightarrow a possible problem is in your module.

Know and *use* as much as possible the *standard library* provided by the language \Rightarrow improve development time, (re)use of debugged and optimized code.

REVIEW the code, if possible with other people.

A simple program – Implementations

Basic design of the example application to be implemented:

Decisions:

1. Good practice : separate the sorting functionality from the user interface \Rightarrow ? possibility to independently change either UI or sorting functionality.

2. Class: `StringSorter`

Methods:

`Read` strings from a file

`Sort` the collection of strings

`Write` strings to a file

Take input and output file names and combine the previous methods

Exceptions: passed to the UI classes.

3. More UI classes, one for each UI type.

A simple program – Implementations

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;
    ...
}
```

Unit testing of the example application:

Decision: Use of JUnit

Write a class containing all unit tests

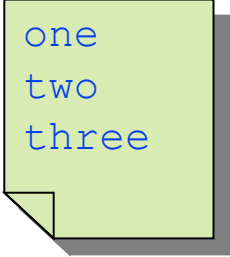
(inherits from `junit.framework.TestCase`, contains `testXXX()` methods, uses `assertEquals()` to compare expected with actual values).

A simple program – Implementations

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;

    public void readFromStream(Reader r) throws IOException{
        BufferedReader br=new BufferedReader(r);
        lines=new ArrayList();
        while(true) {
            String input=br.readLine();
            if(input==null)
                break;
            lines.add(input);
        }
    }
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.



one
two
three

```
public class TestStringSorter extends TestCase {
    private ArrayList make123() {
        ArrayList lst = new ArrayList();
        lst.add("one"); lst.add("two"); lst.add("three");
        return lst;
    }
    public void testReadFromStream() throws IOException{
        Reader in=new FileReader("in.txt");
        StringSorter ss=new StringSorter();
        ArrayList lst= make123();
        ss.readFromStream(in);
        assertEquals(lst,ss.lines);
    }
}
```

A simple program – Implementations

The chosen algorithm :

- Find the largest element in the array

- Swap it with the last element

- Repeat with the rest of the array.

Supporting methods:

- Swapping two elements of the array

- Finding the index of the largest element in a given subarray

A simple program – Implementations

```
static void swap(List lst, int i1, int i2) {  
    Object tmp=lst.get(i1);  
    lst.set(i1, lst.get(i2));  
    lst.set(i2, tmp);  
}
```

```
public class TestStringSorter extends TestCase {  
    private ArrayList make123() {...}  
    public void testReadFromStream() throws IOException{...}  
    public void testSwap() {  
        ArrayList lst1=make123();  
        ArrayList lst2=new ArrayList();  
        lst2.add("one"); lst2.add("three"); lst2.add("two");  
        StringSorter.swap(lst1,1,2);  
        assertEquals(lst1,lst2);  
    }  
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.

A simple program – Implementations

```
static void findIdxBiggest(List lst, int from, int to) {
    String biggest=(String)lst.get(from);
    int idxBiggest=from;
    for(int i=from+1; i<=to; ++i) {
        if(biggest.compareTo((String)lst.get(i)<0) {
            biggest=(String)lst.get(i);
            idxBiggest=i;
        }
    }
    return idxBiggest;
}
```

```
public class TestStringSorter extends TestCase {
    private ArrayList make123() {...}
    public void testReadFromStream() throws IOException{...}
    public void testSwap() {...}
    public void testFindIdxBiggest() {
        ArrayList lst = make123();
        int i = StringSorter.findIdxBiggest(lst,0,l.size()-1);
        assertEquals(i,1);
    }
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.

A simple program – Implementations

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;

    public void readFromStream(Reader r) throws IOException{...}
    public void sort() {
        for(int i=lines.size()-1; i>0; --i) {
            int big=findIdxBiggest(lines,0,i);
            swap(lines,i,big);
        }
    }
}
```

```
public class TestStringSorter extends TestCase {
    private ArrayList make123() {...}
    public void testReadFromStream() throws IOException{...}
    public void testSwap() {...}
    public void testFindIdxBiggest() {...}
    public void testSort1() {
        StringSorter ss = new StringSorter();
        ss.lines=make123();
        ArrayList lst2=new ArrayList();
        lst2.add("one"); lst2.add("three"); lst2.add("two");
        ss.sort();
        assertEquals(lst2,ss.lines);
    }
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.

A simple program – Implementations

More efficiency if we know the standard library of the language:
we do not need the static classes `swap` and `findIdxBig`
the code in class `StringSorter` is more simple.

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;

    public void readFromStream(Reader r) throws IOException{...}
    void sort() {
        java.util.Collections.sort(lines);
    }
}
```

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;
```

A simple program – Implementations

```
public void readFromStream(Reader r) throws IOException{...}
public void sort() {...}
public void writeToStream(Writer w) throws IOException {
    PrintWriter pw=new PrintWriter(w);
    Iterator i=lines.iterator(i);
    while(i.hasNext()) {
        pw.println((String) (i.next()));
    }
}
```

```
public class TestStringSorter extends TestCase {
    private ArrayList make123() {...}
    public void testReadFromStream() throws IOException{...}
    public void testSort1() {...}
    public void testWriteToStream() throws IOException {
        StringSorter ss1 = new StringSorter();
        ss1.lines=make123();
        Writer out=new FileWriter("test.out");
        ss1.writeToStream(out);
        out.close();
        Reader in=new FileReader("test.out");
        StringSorter ss2=new StringSorter();
        ss2.readFromStream(in);
        assertEquals(ss1.lines,ss2.lines);
    }
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.

A simple program – Implementations

```
import java.io.*;
import java.util.*;
public class StringSorter {
    ArrayList lines;

    public void readFromStream(Reader r) throws IOException{...}
    public void sort() {...}
    public void writeToStream(Writer w) throws IOException {...}
    public void sort(String inputFileName, String outputFileName) throws IOException{
        Reader in=new FileReader(inputFileName);
        Writer out=new FileWriter(outputFileName);
        readFromStream(in);
        sort();
        writeToStream(out);
        in.close();
        out.close();
    }
}
```

```
public class TestStringSorter extends TestCase {
    private ArrayList make123() {...}
    public void testReadFromStream() throws IOException{...}
    public void testSort1() {...}
    public void testWriteToStream() throws IOException {...}
    public void testSort2() throws IOException {
        StringSorter ss1=new StringSorter();
        ss1.sort("in.txt","test2.out");
        ArrayList lst= new ArrayList();
        lst.add("one"); lst.add("three"); lst.add("two");
        Reader in=new FileReader("test2.out");
        StringSorter s2=new StringSorter();
        ss2.readFromStream(in);
        assertEquals(lst,ss2.lines);
    }
}
```

Apply the test using a *Test Runner* to verify the implementation of the method.

A simple program – User Interfaces

Until now : unit-tested (not extensive) implementation of `StringSorter`.

User Interface – a program that will allow to access the functionality of `StringSorter`.

Variants :

- CLI (command line interface)

- GUI (graphic user interface)

A simple program – User Interfaces

```
import java.io.IOException;
public class StringSorterCommandLine {
    public static void main(String args[]) throws IOException {
        if(args.length!=2) {
            System.out.println("Use: cmd inputfile outputfile");
        }else{
            StringSorter ss=new StringSorter();
            ss.sort(args[0],args[1]);
        }
    }
}
```

The command line to use the `StringSorter` looks like:

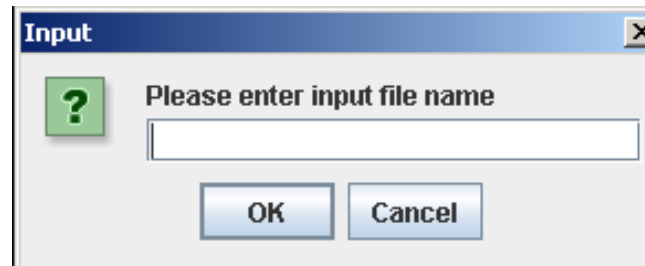
```
java StringSorterCommandLine a.txt a_sorted.txt
```

When is useful such an interface?

What are the advantages of writing such an interface ?

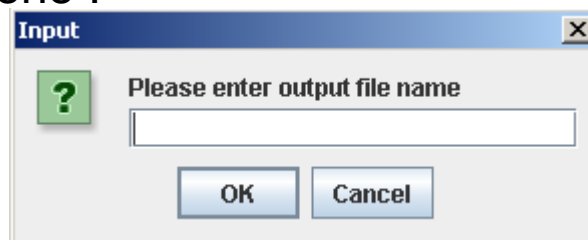
A simple program – User Interfaces

```
import java.io.IOException;
public class StringSorterGUI1 {
    public static void main(String args[]) throws IOException {
        try{
            StringSorter ss=new StringSorter();
            String inputFileName=JOptionPane.showInputDialog("Please enter input file name");
            String outputFileName=JOptionPane.showInputDialog("Please enter output file name");
            ss.sort(inputFileName,outputFileName);
        }finally{
            System.exit(1);
        }
    }
}
```



Do you like this GUI ?

Propose a more friendly one !



Summary; conclusions

Single program

Developed by a one person

Few users

What if ?

Complex system with multiple components.

Issues:

Functional requirements and quality attributes

Design constraints and decisions

Testing

Effort estimation

Implementation details

Activities:

Requirements understanding

Estimate effort and possibly plan the development

Design the solution

Implement the solution

Test for correctness and with the user

A (simple) **PROCESS**.

Minimal **DOCUMENTATION**.

Terminology

Problem - to be solved by a software system.

Problem space – the business domain where the problem is defined.

Specified by user requirements.

Domain model – entities and relationships in the business domain that define the problem to be solved.

Result of the requirements analysis.

Solution space – the software domain in which the solution will be implemented.

Result of design decisions and constraints.

Design – representation of the entities and relationships in the solution space that define the solution software system before it is coded (implemented).

Result of the design process.

Solution – a software system that solves the problem.

Result of the software development process.

Topics covered

Issues in *development* and *support* software products

A simple program

Size and complexity of a system

Technical and non-technical issues

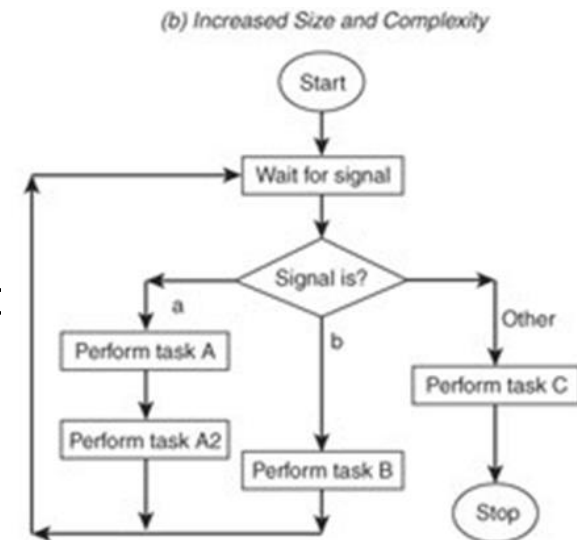
Concerns in case of a large application

Coordination efforts for process, product and people

Size and complexity

Size : number of
major functions
features within each functional area
interfaces to external systems
simultaneous users
types of data and data structures

Complexity :
linkage (ex. sharing data, transfer of control, both)
relationships (ex. hierarchical, sequential, loop, recursive, etc



Topics covered

Issues in *development* and *support* software products

A simple program

Size and complexity of a system

Technical and non-technical issues

Concerns in case of a large application

Coordination efforts for process, product and people

Technical issues

Handling the SIZE and COMPLEXITY:

Problem decomposition

Solution modularization

Separation of concerns

Incremental iterations

Technical issues

Technology and tools

Technical choices:

Programming language:...

Development tools : IDEs, ...

Infrastructure : DBMS, network, middleware,...

Management tools : code version control, ...

Diversity in background and experience of the team members ⇒
agreement for the choices, training plans, etc.

Technical issues

Process and methodology

Software development process (software development lifecycle – SDLC) = The set of ***activities***, the ***sequence*** and ***flow*** of these activities, the ***inputs*** to and the ***outputs*** from the activities, and the ***preconditions*** and ***postconditions*** for each of the activities involved in the production of software.

Contains the activities required for developing and maintaining software.

Used in guiding and in coordinating and managing complex projects involving many people.

Methodology – a particular procedure or a set of procedures.

Technical issues

Process and methodology

Comon activities in software development process.

Is there a methodology for gathering requirements? What if more persons are implied ?

Requirements gathering and specification

Design

Code / unit test

What constitutes user support ?
What kind of problems must be fixed?

User support and problem fix

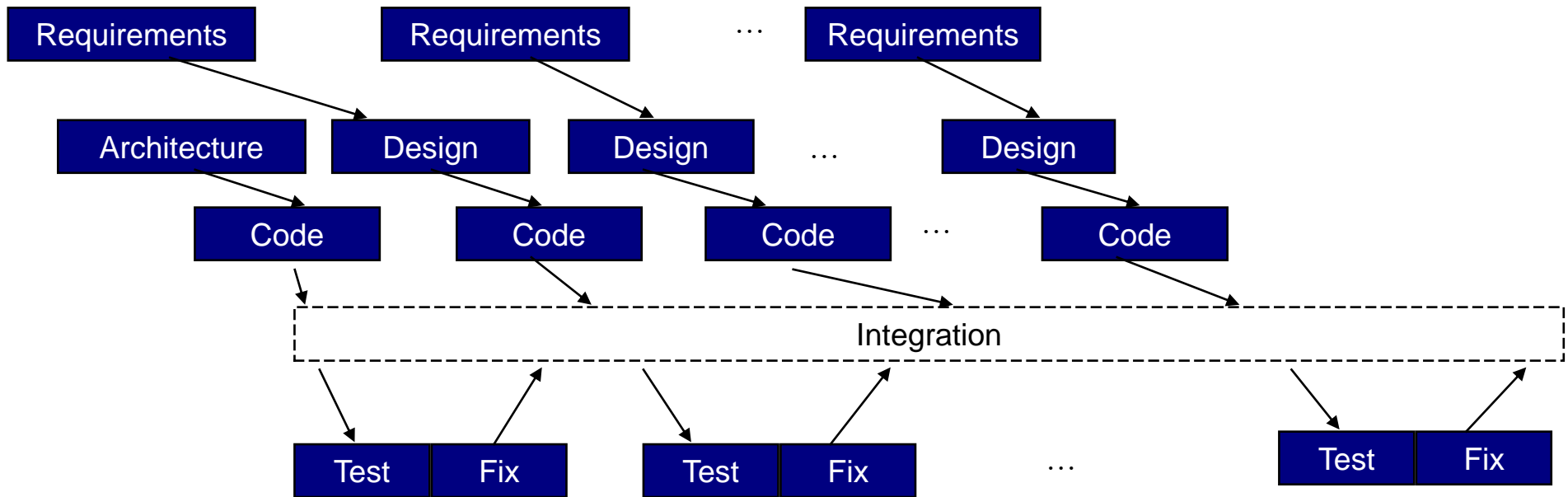
How do these activities relate to each other? (sequences, overlap, starting conditions,...)



Integrate and test

Technical issues

Process and methodology – a possible process approach:



Incremental development (problem and solution divided into increments) and continuous integration.

nonTechnical issues

Effort estimation and schedule

Issues:

Difficult before understanding the requirements.

Needs knowledge about individual productivities of the team members.

Difficulty in direct relation with the size and complexity of the problem.

More accurate as the project goes on.

More accurate if experience with similar projects exist.

Assignments and communications

Assignment of people to tasks based on *skills* and *availability*

Objective : assign the most effective and properly skilled people to the right tasks.

Communications

The number of direct communication paths grows as C_n^2 with the number of people involved in software development.

Probability of error in communication grows with the number of people implied

⇒ need for communication *structures* and *protocols* and for *standard meaning* of messages.

Topics covered

Issues in *development* and *support* software products

A simple program

Size and complexity of a system

Technical and non-technical issues

Concerns in case of a large application

Coordination efforts for process, product and people

A more complex problem: Hypothetical Payroll System

Payroll : (1) the sum of all financial records of salaries for an employee, wages, bonuses and deductions. (2) amount payed to employees for services they provided during a period of time.

Hypothetical Payroll System Requirements

Functional requirements: (examples)

What the “associated information” is?
Whom shall I ask? User, client, project manager?
Should I document the answers?

- Add, modify, delete names and associated personal information of all employees
- Add, modify, delete all the benefits associated with all employees

What are “all benefits”?
What is the implication of having a benefit on an employee’s payroll?
Is there a list of all possible benefits? Will it be modified in the future?

Hypothetical Payroll System Requirements

Functional requirements: (examples)

- Add, modify, delete all the tax and other deductions associated with all employees
- Add, modify, delete all the gross income associated with all employees
- Add, modify, delete all the algorithms related to computing the net pay for each employee

Hypothetical Payroll System Requirements

Functional requirements: (examples)

- Generate a paper check or an electronic direct bank deposit for each employee

Which is the payroll cycle?

Which is the deadline for inputs to the cycle (ex. salary increase)

In order to properly handle the functional requirements you need to understand the *application domain-specific knowledge* related to the problem.

Hypothetical Payroll System Requirements

Quality attributes (nonFunctional requirements): (examples)

- Performance

Which is the volume of payroll transactions?

Which is the speed of processing each payroll transaction?

In order to properly handle some quality attributes you need to have *knowledge of technical system*.

Hypothetical Payroll System Requirements

Quality attributes (nonFunctional requirements): (examples)

Usability

What is the experience of the user with GUI?
What are the user profiles?
How to undo and reprocess a paycheck based on a bad record?

In order to properly handle other quality attributes you need to have *interface information*.

Hypothetical Payroll System Design

Should “add, update, delete” functional requirements be grouped into a single component called “payroll administrative functions”?

YES !

Group related functions into components



Hypothetical Payroll System Design

Should I group processing functions (calculations of all deductions and of net pay amount) into a single component called “payroll processing”?



YES !

Group related functions into components

Hypothetical Payroll System Design

Shall I place all interface functions with external systems into a component called “payroll interfaces”.



YES !

Group related functions into components

Hypothetical Payroll System Design

I must be prepared to handle errors and exceptions.
Shall I aggregate handlers into an exception-processing
component?

YES !

Group related functions into components



Hypothetical Payroll System Design

Advantages of *grouping related functions into components*:

- Provides some design *cohesiveness* within the component
- Matches the business flow and the payroll processing environment
- Provides a potential assignment of work by components
- Allows easier packaging of software by components.

Hypothetical Payroll System Design

Designers must deal with both *cohesion* and *coupling* characteristics of a software design

looking for

high cohesion of each component

and

low coupling between components.

Hypothetical Payroll System Design

User interface prototype :

- layout and style
- content
- navigation relationships

Obs. In this example is not a prime design concern because the application is heavily batch-oriented and less interactive.

Database design :

- tables
- keys

Obs. In this example is an important design concern because the application is data intensive.

Hypothetical Payroll System Design

Applied techniques:

- Functional decomposition
- Synthesis
- Defining *intercomponent* interactions
- Defining *intracomponent* interactions

Difference from designing a single programming module:

- Greater discipline
- Additional guiding principles
- More team members

Hypothetical Payroll System

Code and Unit Testing

For each functional unit in a given component **build** one or more **programming units (modules)**

In case of *more programming units* establish standards for:

- naming conventions for unique identification of the module
- comments
- error messages

At module level:

Test each programming unit (module) :

- Set the conditions of the module
- Choose the appropriate input data
- Run the module and observe its behaviour through checking the output

Fix the discovered error

Re-test the module

Hypothetical Payroll System

Integration and functionality testing

Functional unit level:

Integrate (compile and link together) all *tested modules* into the corresponding *functional unit*.

Apply *functional test cases*¹ on the *functional unit*.

Fix errors in the specific *modules*.

Re-test the *functional unit*.

After functional testing have been passed, **lock** the modules in the functional unit for further changes.

Use a *configuration management mechanism*. It may be automated using configuration management tools (ex. CVS, Subversion,...)

¹ *Test cases* (scenarios) are designed based on the *use cases* (scenarios) which are derived from the *functional requirements* of the system.

Hypothetical Payroll System Release

Integrate all the *components* and **test** to assure that the *complete system* works as a whole and in the user business environment.

Detect *system interface* problems.

Detect *component interaction* problems.

Fix discovered problems.

Re-test each *modified module, related functional unit* and the *system as a whole*

When no more problem is found **protect** this *version of the system* (release) from further changes.

Hypothetical Payroll System Release

Educate the users in the usage of the system:

- Prepare the training material
- Train the users

Prepare user support personnel – trained in:

- Payroll system
- User environments
- Tools needed to support customers

RELEASE the payroll system.

Hypothetical Payroll System Support and Maintenance

Simple application : is not a major concern.



Complex application :
may be a very complex set of tasks.



Consider at least two sets of support personnel:

- A group to answer and handle system usage and simple problem work-arounds

Skills: communication, payroll system usage knowledge

- A group to fix difficult problems and implement future enhances

Skills: design, coding



Obs. Support organization is comparable in size and complexity to the original development team.

Topics covered

Issues in *development* and *support* software products

A simple program

Size and complexity of a system

Technical and non-technical issues

Concerns in case of a large application

Coordination efforts for process, product and people

Coordination Efforts

Process:

1990s - Improved (extended, complicated) by:

- More reviews, inspections, testing, meetings
- Expensive quality assurance and measurement efforts

to prevent, detect and correct problems, improve the quality of software and increase the productivity of software developers.

2000s – simplified to face new market challenges of speed and cost.

No single process fits all cases!!!

Coordination Efforts

Product:

Software product *content* coordination:

- Executable code
- Documentation: requirements specification, design, functional test scenarios, user manual
- User education (optional)
- Product support (optional)

Coordination of *changes* because:

- The initial high functional cohesiveness and low coupling erode after changes are applied
- Complexity increases \Rightarrow testing effort increases

How to design for changes ?

How to coordinate changes ?

Coordination Efforts

People:

Human resource is crucial in the development and support of software!!!

Software industry is still “labor intensive” (! High qualified intellectual labor !)

How to coordinate people’s activities?

How to manage people ?

Q&A

What ...? Why...? How...?
Who...? When...?

SOFTWARE ENGINEERING

“A typical software development endeavor involves more than one person working on a complex problem over a period of time to meet some objectives.

As a new student, understanding what software engineering is about is not easy, because there is no way we can bring its realities and complexities into the student’s world.

Nevertheless, it is a student’s responsibility to embark on this journey of learning and discovery into the world of software engineering.”

Computing

Computing - any goal-oriented *activity requiring, benefiting from, or creating* computers.

Computing disciplines:

- Computer Engineering
- Computer Science
- Information Systems
- Information Technology
- **Software Engineering**

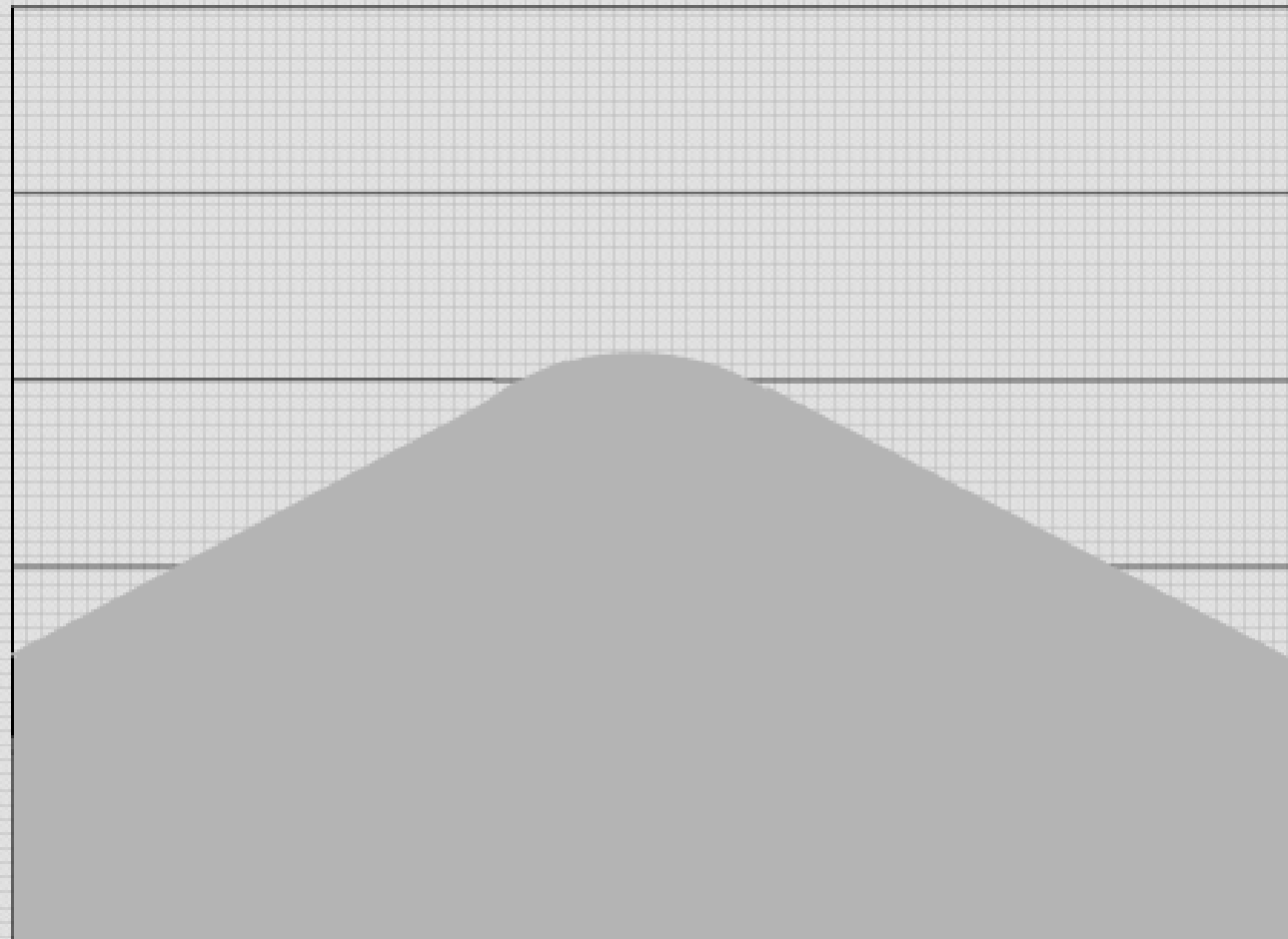
Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

Computer Hardware
and Architecture



Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration

← More Theoretical

More Applied →

CE

Computing Curricula 2005 – The Overview Report
A cooperative project of ACM, AIS, IEEE-CS

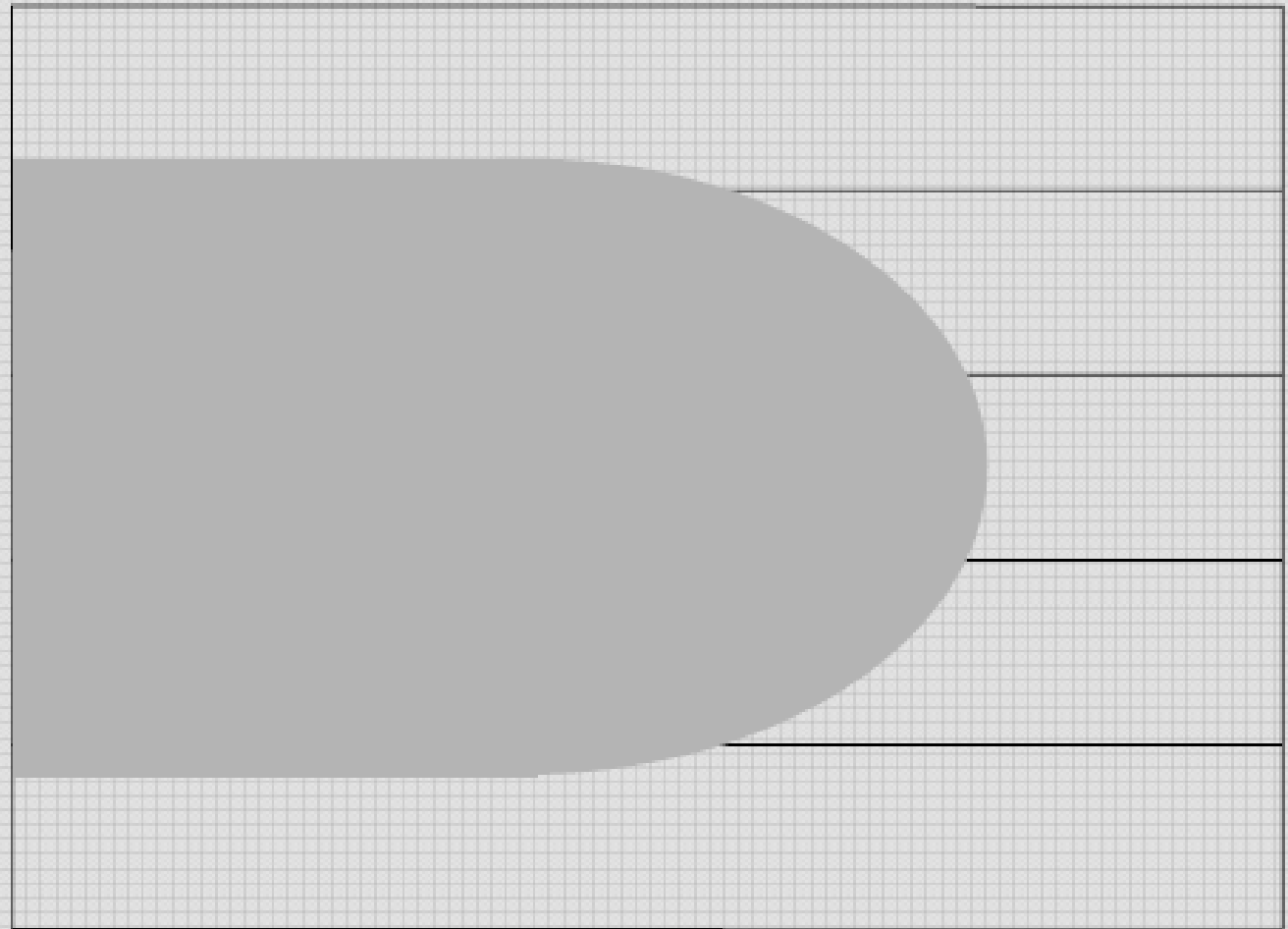
Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

Computer Hardware
and Architecture



Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration

More Theoretical

More Applied

CS

Computing Curricula 2005 – The Overview Report
A cooperative project of ACM, AIS, IEEE-CS

Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

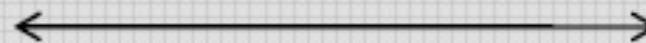
Computer Hardware
and Architecture



Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration



More Theoretical

More Applied

IT

Computing Curricula 2005 – The Overview Report
A cooperative project of ACM, AIS, IEEE-CS

Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

Computer Hardware
and Architecture

Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration

More Theoretical

More Applied

IS

Computing Curricula 2005 – The Overview Report
A cooperative project of ACM, AIS, IEEE-CS

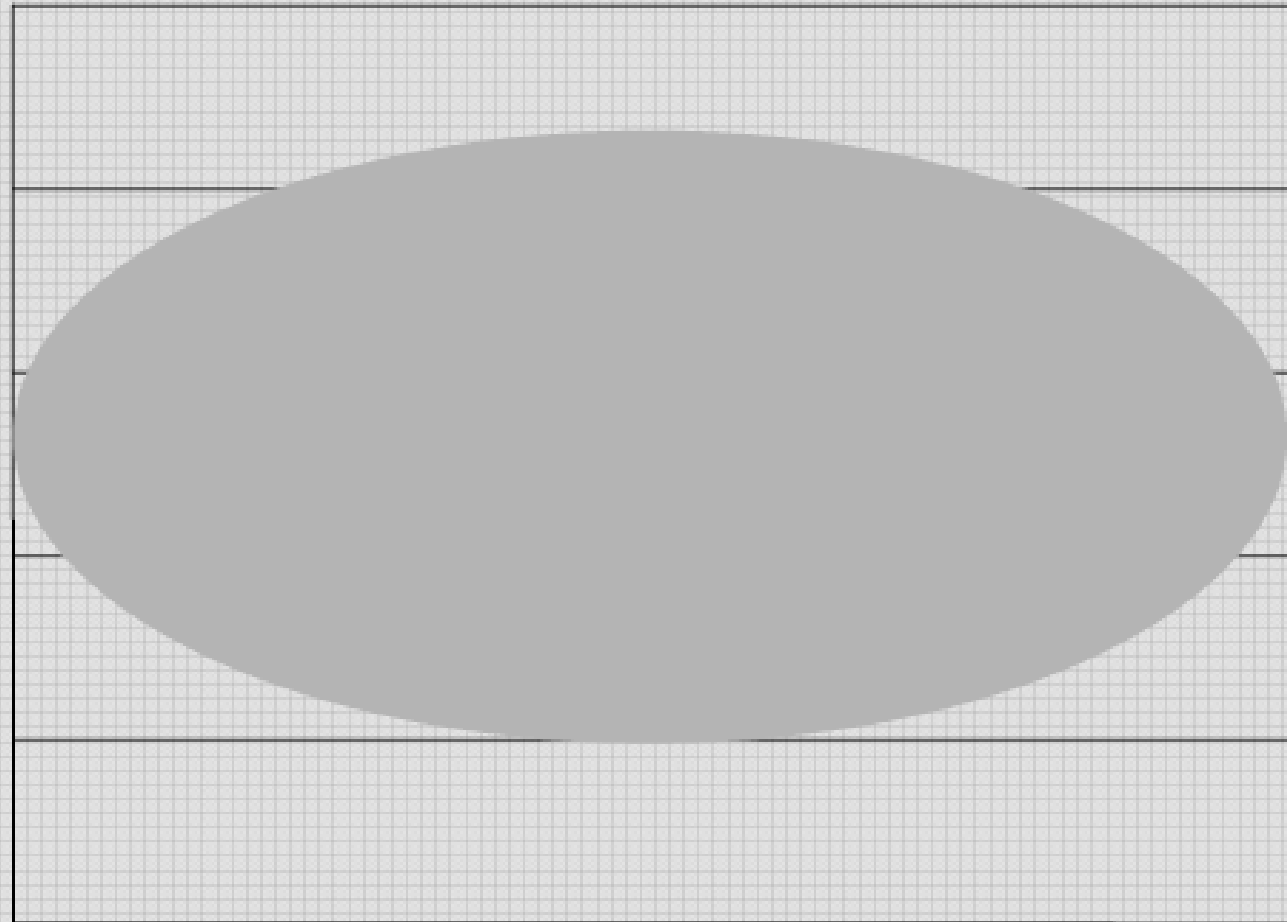
Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

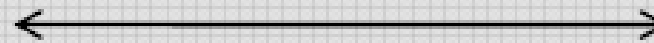
Computer Hardware
and Architecture



Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration



More Theoretical

More Applied

SE

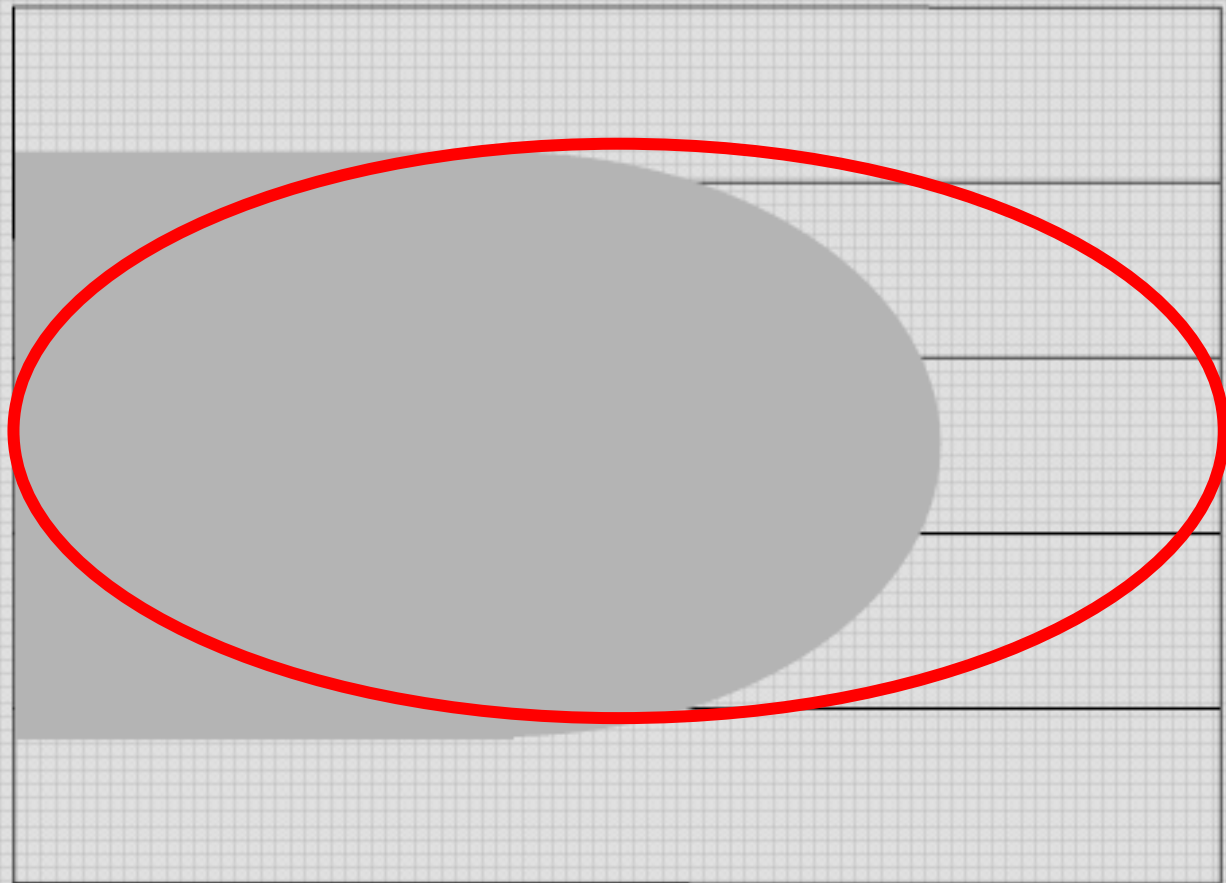
Organizational Issues
& Information Systems

Application
Technologies

Software Methods
and Technologies

Systems
Infrastructure

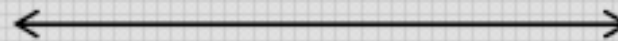
Computer Hardware
and Architecture



Theory
Principles
Innovation

DEVELOPMENT

Application
Deployment
Configuration



More Theoretical

More Applied

CS

Computing Curricula 2005 – The Overview Report, A cooperative project of ACM, AIS, IEEE-CS