FILE: CURS1.SE

SOFTWARE ENGINEERING

INTRODUCTION

The term 'software eng
ineering' was first introduced in the late 1960s at a conference held to discuss what was called the software crisis. This software crisis resulted directly from the introduction of third generation computer hardware. These machines were orders of magnitude more powerful than second generation machines and their power made hitherto unrealizable applications . The implementation of these applications required large software systems to be built.

Early experience in building large software systems showed that existing methods of software development were not good enough. Techniques applicable to small systems could not be scaled up. Major projects were sometimes years late, were unreliable, difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling while software costs were rising rapidly.

Now, more than 20 years later, the software crisis has not been resolved. Although there have been real improvements in software engineering methods and techniques, in tools for system development and in the skills of development staff, the demand for software is increasing faster than improvements in software productivity.

There are four key attributes which a well engineered software system should possess:

(1) The software should be maintainable (it should be written and documented so that changes can be reasonable made).

(2) The software should be reliable (it should not fail more than is allowed in its specification).

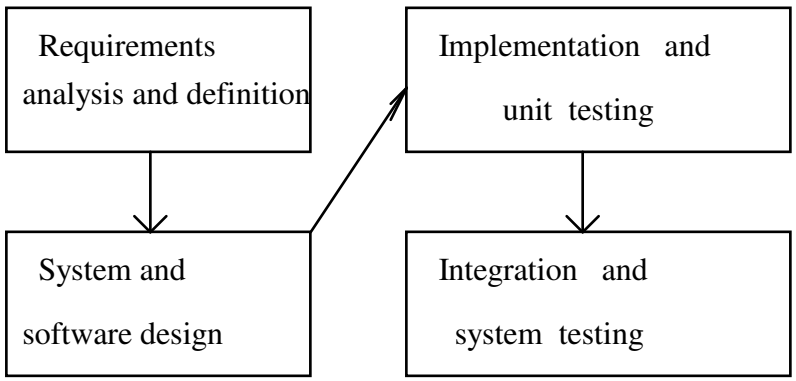
(3) The software should be efficient. This means that a system should not make wasteful use of system resources such as memory and processor cycles.

(4) The software should offer an appropriate user interface.

The costs associated to software engineering systems must be taken into account.

The identification of the software crisis in the late 1960s and the notion that software development is an engineering discipline led to the view that the process of software development is like other engineering processes. Thus a model of the software development process was derived from other engineering activities [Royce70].This was enthusiastically accepted by software engineers, by offering means of making the development process more visible. Because of the passing from one phase to another, this model is known as the waterfall mode (see figure 1). This software model was put into use but soon became clear that it was only appropriate for some classes of software system. Although the managers found the model useful for planning and reporting.

Fig. 1 The waterfall model



Detailed software enginering models are still subject of research but it is now clear that a number of different general models or paradigms of software development can be identified. The original waterfall model is one of these general models. Some of these general development models and paradigms are :

(1) **The waterfall approach**. This views the software process as being made up of a number of stages such as requirements specification, software design, implementation, testing.

(2) **Exploratory programming**. This approach involves developing a working system, as quickly as possible, and then modifying that system until it performs in an adequate way. This approach is usually used in artificial intelligence systems development.

(3) **Prototyping**. This approach is similar to exploratory programming in that the first phase of development involves developing a program for user experiment.This phase is followed by a requirement for implementing the software to produce the system.

(4) **Formal transformation**. This approach involves developing a formal specification of the software system and transforming this specification, by preserving correctness transformations, into a program.

(5) **System assembly from reusable components**. This technique assumes that systems are mostly made up of components which already exist. The system development process becomes one of assembly rather than creation [Gheorghe, Ritchie 93].

The first three of these approaches are all currently used for practical systems development. Some systems have been built using correctness-preserving transformations and the reuse oriented model is still not comercially developed due to the lack of reusable component libraries.

The project referred in this presentation contains an implementation of a set of software engineering tools and a command language interpreter (all produced by P. Ritchie in the University of Central Lancashire) and a language to manage these tools, called SCL, designed by the author of this report and P. Ritchie and an implementation due to the author of a front-end for this language. In order to become operational this implementation, a back-end must be produced starting from the existing CLI and the tools already implemented. This would be implemented in student project co-ordinated by the two partners above mentioned. In this way a

useful software engineering example would be available for teaching reasons in an academic area. This approach can be further developed with other new tools or/and new features added to the language SCL.

1. Software requirements definition

The software engineering problems are often immensely complex. Understanding the nature of the problem can be very difficult, particularly if the system is new and there is no existing system to serve as a model for the software. The result of the analysis of the problem is a requirements specification which constitutes the first formal document produced in the software process.

The software specifications should be produced at several different levels of abstraction with careful correlations between these levels. These levels of specification are:

(1) A requirements definition is a statement, in a natural language, of what the system is expected to do.

(2) A requirements specification is a structured document which sets out the system services in more detail. It should be written so that it is understandable to technical staff from both users and developers. Formal specification techniques may be appropriate for expressing such a specification but this will depend on the background of the user.

(3) A software specification is an abstract description of the software which is a basis for its design and implementation. There should be a clear relationship between this document and the requirements specification but the important distinction is that the readers of this are principally software designers. Formal specification techniques are appropriate in this document.

The requirements document is a reference tool, containing a combination of requirements definition and requirements specification. For example such a document could be organized in the following way:

- introduction, describing the context of the system, the rationale for the software;
- the system model, containing the relations between components and the connections with the environment;
- system evolution, describes anticipated changes due to hardware evolution;
- functional and nonfunctional requirements, contain the details about requirements and restrictions, facilities for the user;
- glossary of technical terms used in the document.

The software requirements document is not a design document, it should contain what the system will do without specifying how it will do.

The specification of the requirements can be given in a structured way using a natural language or in (semi)formal notation using a language of specification. In the first case there are specified the input/output and functions related to the system, the pre- and post-conditions; an Ada-like notation can be used. In the second case a notation due to DeMarco [DeMarco 78], Constantine and Yourdon [Constantine Yourdon 79] and using the concept of data flow diagrams can be investigated.

The structured approach contains a description of the following type:

Function : describes the purpose of the (sub)system
Description : presents the main problems
Inputs : the inputs
Outputs : the outputs
Requires : the required (sub)systems
Destination : the (sub)system using the output information
Pre-conditions : the conditions imposed to input data
Post-conditions: the conditions required for output data

A data flow diagram shows how input data is transformed to output results through a sequence of functional transformations. They are useful and intuitive way of describing a system and they are understandable without a special training. The notation used here has been chosen because it is easy to draw using a personal computer ordinary editor. The symbols used in this natation are as follows:

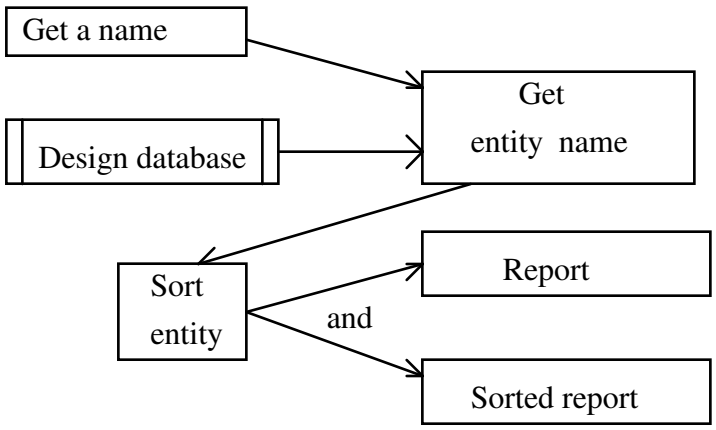
(1) Rectangles: these represent transformations where an input data flow is transformed to an output. The transformation is annotated with a name.

(2) Rectangles with double vertical lines: these represent data store.

(3) Arrows: these show the direction of the data flow. They could contain a name describing the transformation.

(4) The keywords **and** and **or**: these are the usual meaning in the boolean expressions and they are used to link the data flows. See figure 2.

Fig 2. A data flow diagram



1.1. Example: A library system.

The problem. The purpose of this system is to provide the main operations related to the management of a public library: enter a book in the library, loan a book, return a book, list the books with a given property, exit from the system.

The scenario of this application could be described in the following way:
The public library in an anonymous town, received a computer (a compatible IBM PC) and decided, having a young student in computer science as an employee, to design a system in

order to facilitate the main operations in this library. There are some requirements related to this system: the system will contain information about each book in the library (title, author(s), editor, year, date of acquisition, state of the book - in the library, loaned, in another library -, loan date, the domain covered by the book), details about the people using this library (name, affiliation, book loaned). It is to be mentioned that the system will be developed with other features concerning libraries connected with this library, extension of the system with journals, technical reports. There are no restrictions concerning the portability of the system, the software tools used to design. If there are at least two such tools, a case study is required, to analyse the appropriate tools.

The young student, having an initiation in software engineering, produced, first, a document containing the requirements of this system.

The requirements definition

The purpose of this system is to provide the main operations related to the management of a public library: enter a book in the library, loan a book, return a book, exit from the system.

The requirements specification

This part of the document contains a description of the requirements of the system given in a structured manner, some methods of specification will be also given.

General description

A. - two kind of information are used:
    - book information, containing:
        - title
        - author(s)
        - editor
        - year
        - acquisition date
        - state of the document
        - loan date
        - key words
    - people information, with:
        - name
        - affiliation
        - book name

B. - main operations:
    - books introduction
    - people introduction
    - loan and return a book
    - list of books having a given property

Software specification

In this part it will be presented a structured way of presentation in which the natural language will be used with some spefications concerning the subsystems and a data flow diagram.

(1)

(a) The natural language description;

The main operations of this system can be divided in the following elements:

- introduction of new books in the system
- introduction of people in the system
- find a book and person to assign/return this book
- list the books having a given property.

(b) The structured way of presentation of subsystems:

(i):

Function : Books introduction.
Description : Any new book is introduced in the system.
Input : The input is the keyboard
Output : The information concerning each book are introduced in the 'Library' file
Requires : none
Destination : none
Pre-conditions : none
Post-conditions: Status := in the library

(ii): Person introduction; Exercise.

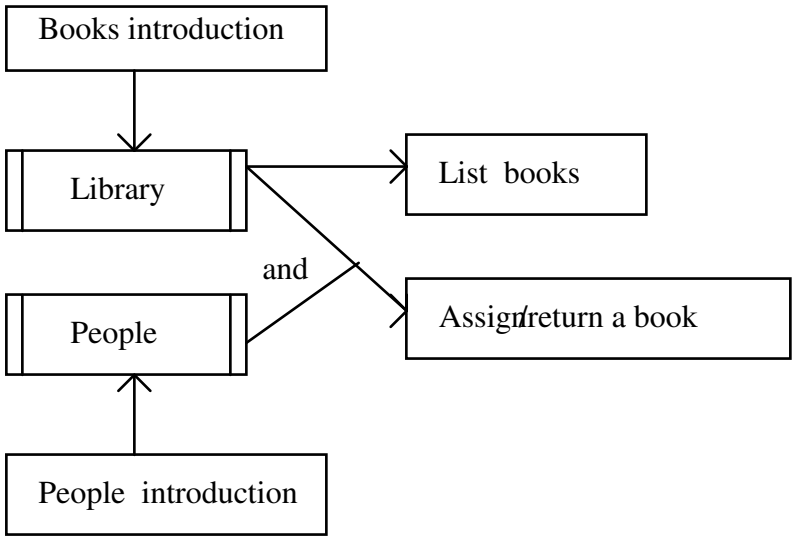
(iii):

Function : Assign/return a book
Description : When a person wants to loan a book, the name of that person and the book are searched. If the person is in the file 'People' and the book in the file 'Library' then the book is assigned or returned.
Input : Keyboard, 'Library', 'People'
Output : Display
Requires : Books introduction & Person introduction
Destination : none
Pre-conditions : the book in 'Library' & the person in 'People' & (for loan, Status = in the library or for return, Status = loaned)
Post-conditions: for loan, Status = loaned; for return Status = in the library

(iv): Books list. Exercise

(2)

A data flow approach will be given for the system and subsystems:



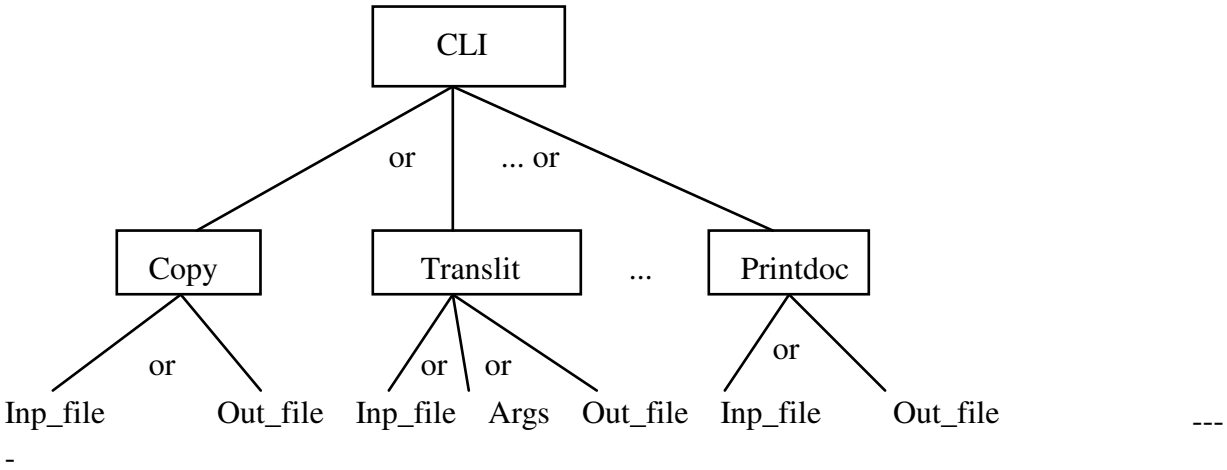
Obs.

1. From these types of specifications it may be observed that these methods are complementary ones.
2. These specifications can be used to define a project with the subject presented above.

Another example is that of a system to manage some software engineering tools for copying, sorting, formating, comparing. These tools have as input/output, text files and could be combined in a UNIX-like manner. The requirements are presented in a document containing the descrition of each function in a natural language.
Structured description for Sort:

Function : Sort a file
Description : Takes the content of an input text file and sorts the lines in ascending order
Input : A text file
Output : A text file
Requires : A tool to create a text file
Destination : Generally, for Unique etc
Pre-condition : An existing text file
Post-conditions: A text file with the lines in ascending order.

A data flow diagram for the problem:

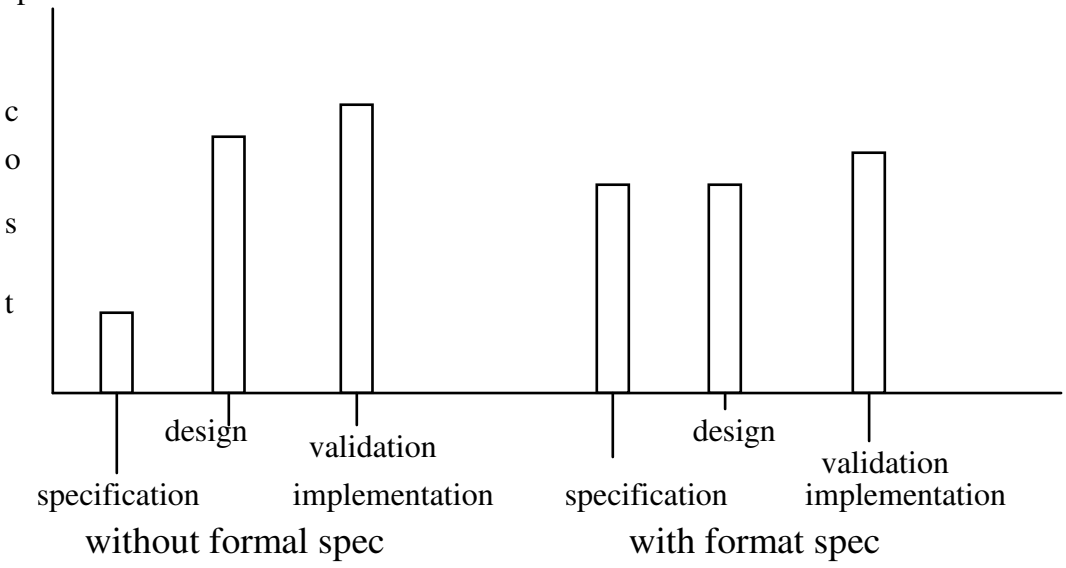


1.2. Formal Specifications

In this part of the report we are dealing with formal specification techniques. This presentation will show formal methods that are used to specify the requirements related to a software project.

There have been different approaches taken to formal specifications and formal methods in Europe and North America. There has been relatively little use of formal methods in North American industrial practice and most US practitioners still consider formal methods as a topic for research. In Europe, however, there have been significant number of industrial trials of formal specifican and the transfer of this technology into industrial practice is accelerating.

The next figure shows how software process costs are affected by the use of formal specification.



1.2.1. Pre- and post-conditions

The simplest form of formal specification is offered by the representation of a system as a set of functions with hidden internal state. Each function is specified using pre- and post-conditions. The pre-condition is a specification of the inputs and the post-condition is a specification for the outputs. The difference between them defines how the function transforms its inputs into the outputs.

For example we give formal description for a search function

function Search(X: Integer_array; Key: Integer): Integer;

Pre-condition: exist I in X'First..X'Last such that X[I]=Key
Post-condition: X"[Search(X,Key)]=Key and X=X"

An error condition may be added:

function Search(X: Integer_array; Key: Integer): Integer;

Pre-condition: exist I in X'First..X'Last such that X[I]=Key
Post-condition: X"[Search(X,Key)]=Key and X=X"
Error: Search(X,Key)=X'Last+1

Other notations like VDM (Jones) or Z (Spivey) involve the use of specialized symbols invented by authors. This leads to precision but involves a significant learning effort before reading or writing specifications.

Z is a specification language developed at the University of Oxford, based on typed set theory whose semantics are formally defined.

The central concept in Z notation is that of schema. A Z schema is composed by a name a signature and a predicate.

Example

```
Container                -- name
---------------------
contents: N              -- signature
capacity: N
---------------------
 contents ≤ capacity   -- predicate
```

Some important concept used in this notation are: functions, operations on sets and functions, sequences.

Other formal notations contain algebraic methods, finite state methods, stream notations. These notations are sequential methods. Parallel methods, CSP (Hoare), CCS (Milner), are also investigated, but not presented here.

For all these methods there are appropriate problems to be formalized.

Examples.

1. Pre- and post-conditions are specified for Sort:

procedure Sort (**in** X: Integer_array; **out** Y: Integer_array);
Pre-conditions: X[i] is a text line, 1<=i<=X'Length;
X[i] is $s_1...s_k$, s_j is a character
and s_k is CR.
Post-conditions:{Y[1], ...,Y[Y'Length]} = {X[1], ...,X[X'Length]}
Y[i]<=Y[i+1], 1<=i<Y'Length.

2. Pre- and post-conditions for assign books to a person:

procedure Assign_books(Books: Books_array; Person: Person_type);
Pre-conditions: Books[i] in Library (i.e. Books[i].Status
:= in_the_library) and Person in People.
Post-conditions:Books[i].Status := Loaned, 1<=i<=Books'Length.

1.2.2. Z notation

The Z notation is a language and a style for expressing formal specifications of computing systems.It is based on a typed set theory. The best way to see how these ideas work out is to look at a small example. For a first example, it is important to choose something simple, a system which records people's birthdays, and is able to issue a reminder when the day comes round.

BirthdayBook
---
known: P(NAME)
birthaday: NAME → DATE
---
known = dom(birthday)

This is called a schema and starting from description in which the domain of the partial function birthday is identified by known, some operations could be constructed. Before this, we mention that known and birthday are the **states**.

(1) Add a new birthday:

AddBirthday

ΔBirthdayBook
name?: NAME
date?: DATE
---
name? ∉ known
birthday' = birthday ∪ {name? ↦ date?}
---

(2) Find the birthday of a person

FindBirthday

ΞBirthdayBook
name?: NAME
date!: DATE
---
name?: ∈ known
date! = birthday(name?)
---

(3) People having a given birthday

Remind

ΞBirthdayBook
today?: DATE
cards!: P(NAME)
---
cards! = { n: known | birthday(n) = today? }
---

(4) If REPORT = { ok, already_known, not_known } then

Success
---
result! : REPORT
---
result! = ok

AlreadyKnown

ΣBirthdayBook
name?: NAME
result!: REPORT
___
name? ∈ known
result! = already_known

We can combine the above descriptions to obtain a new robust version of AddBirthday

RAddBirthday ::= (AddBirthday & Succes) | AlreadyKnown

This definition leads to the following schema

RaddBirthday

ΔBirthdayBook
name?: NAME
date?: DATE
result!: REPORT

(name? ∉ known &
 birthday' = birthday U {name? |→ date?}
 result! = ok) |
(name? ∈ known &
 birthday' = birthday &
 result! = already_known)

1.2.2.1. Background of Z

(0) **Numbers**. The type of integer numbers, Z, is predefined in the language. The following operations are allowed in the language: +, -, *, **div**(integer division), **mod**(modulo). The last operator is defined by
i=(i div j) * j + i mod j, $j \neq 0$
Relation operators are also defined $\neq$, =, <, >, $\leq$, $\geq$.
Z also contains the unary functions - and succ, pred.

(1) **Propositional and predicate calculus**. If p and q are propositions then
$p \wedge q$, $p \vee q$, $\neg p$, $p \Rightarrow q$, $p \Leftrightarrow q$ are also propositions. In predicate calculus the $\forall$ and $\exists$ quantifiers are used.

(2) **Sets**. Enumeration definition: Set_name == {ent1, ent2, ..., enr_n};
A property definition: Set_name == {declaration | formula • term}. Ex
Evens ==$\{n:N | n \neq 0 \wedge n$ **mod** $2 = 0 \bullet n\}$
Operations:
union($\cup$), intersection($\cap$), cartesian product($\times$), difference($-$), symmetric difference(Δ).

(3) **Relations** .
R, a subset of $X \times Y$ has X = **dom** R, Y = **ran** R.Notation: the set of all relations between X and Y is $X \leftrightarrow Y == P(X \times Y)$. In order to specify that R is such a relation we write
$R: X \leftrightarrow Y$.
- the image of a relation R for $U \subseteq X$ is $R((U))=\{y:Y | (x,y) \in R\}$
- relational inversion:if R is a relation then $R^{\sim}$ is the inverse of R.
- domain restriction : if R is a relation and $S \subseteq X$ then $S \lhd R = \{(x,y):R | x \in S\}$
- anti-domain restriction: The complement of the domain restriction

####################
(3) Sequences
seq: finite sequences
seq1: non-empty finite sequences **iseq**: injective sequences
$<a_1, ..., a_n>$ - sequence
< > - empty-sequence
if X is a set then
seq X = { $<a_1,...,a_n> | a_i \_X$, n>=0}
concatenation:
s,t: **seq** X
$s = <a_1,..,a_n>$, $t = <b_1,...,b_k>$
$s \hat{} t = <a_1,..,a_n,b_1,...,b_k>$
head, last, tail, front, rev
$s=<a_1,a_2,...,a_{k-1},a_k>$, a_i referred as s(i)
head $s = a_1$
last $s = a_k$
tail $s = <a_2,...,a_k>$
front $s = <a_1,...,a_{k-1}>$

rev s = <a_k,...,a_1>
composition:
s: **seq** X, f: X ----> Y, then f ° s : **seq** Y
s=<a_1,...,a_k> then f ° s =<f(a_1),...,f(a_k)>

(4) schema
Has the form

Schema_Name

Signature

Predicate
---

input (output) elements are followed by ? (!).
If S is the a schema name then
_S is the schema containing the states s and s'
_S is the schema in which s'=s

1.2.2.2. Examples

(1)
BirthdayBook
---
known: P(NAME)
birthday: NAME ------> DATE
---
known « birthday
---

(2)Remove an entry from BirthdayBook
Remove

_BirthdayBook
name?: NAME
date?: DATE
---
name?: _ known

known' = known - {name?}
birthday' = known' « birthday
---

(3) Search a key in an array

Search
---
key?, i!: Z
X: **seq** Z
---
i!, 1<= i! <= #X, key? = X(i!)
---

(4) Sort a sequence of integers

SeqofInteger
---
a: **seq** Z

SortIntegers

_SeqofInteger
---
a(1)'<=...<=a(#a)'
#{a(j)|a(j)=a(i)} = #{a(j)'|a(j)'=a(i)}, i=1,#a
---

(5) Sort the lines in a text file

TextFile

L: **seq** CHAR
T: **seq seq** CHAR
---
last L = CR
last T = <EOF>

SortTextFile

_TextFile

T(1)'<=...<=T(#T-1)'
#{T(j)|T(j)=T(i)} = #{T(j)'|T(j)'=T(i)}, i=1,#T

---
(6) Add a book

Book = Title x Author x Editor x Year x Acquisition_Date x Keywords x Loan_Date
Person = Person identification

Book_type
---
book: Book

Person_type
---
person: Person

AddaBook

_Book_type
name?: Title
author?: Author
editor?: Editor
loan_date?: Loan_Date
---
book' = book U {(name?,author?,editor?,...,loan_date?)}
status' = in_library
---

(7) Stack operations

Let X be a type

Stack
---
s: **seq** X
limit: Z

(7.1) Initial state

Init

Stack
---
s = < >

(7.2) Push an element
Push

_Stack
x?: X

#s < limit
s' = <x?>^s

(7.3) Pop an element

Pop

_Stack
x!: X
---
s = < >
x! = **head** s
s' = **tail** s

1.2.2.3. Schema Operations

(1) Inclusion
Let S be a schema and T will be defined by

T

S
-- other declarations

-- predicate

There are two conditions:
a. the signature and predicate of S are inserted into T
b. S and T are not contradictory

(2) Conjunction, disjunction, implication, equivalence

Let S, T be two schemas and **op** one of the above mentioned operations, then

S_op_T

-- declarations of S, T
---
predicate of S **op** predicate of T
---

(3) Unnamed schema

[signature | predicate]

Exercises

1. For the library system a structured description for person introduction and books list is required.
2. For the Books list subsystem describe a data flow approach.
3. Provide for Charcount a structured description and define the pre- and post-conditions associated to it.
4. A structured description and a formal description using the pre- and post-conditions are required for Unique.
5. Give a formal description using Z for Charcount.

References

[Ritchie91] Peter Ritchie: Software Engineering Tools. User Manual, University of Central Lancashire, 1991.
[Sommerville93] Ian Sommerville: Software Engineering, Addison- Wesley,1992.
[Spivey89] J. M. Spivey: Understanding Z. A specification language and its formal semantics, Cambridge University Press, 1989.
[Lightfoot90] D. Lightfoot: Formal Specification using Z, Macmillan Computer Science Series, 1990.

FILE: CURS2.SE

2.Software design

A general model of software design process is a directed graph. Nodes represent entities in the design, such as processes, functions or types and links represent relations between these design entities.

Software design involves the following stages:

(1) Study and understand the problem.

(2) Identify the main features of at least one possible solution. It is useful to identify a number of solutions and to evaluate each of these.

(3) Describe each abstraction used in the solution. Before creating formal documentation, the designer will construct an informal design description and debug this one by developing it in more detail. Activities in the design of large software systems:

(1) Architectural design

(2) Abstract specification

(3) Interface design

(4) Component design

(5) Data structure design

(6) Algorithms design

This process is iterated for each sub-system until the components identified can be mapped directly into programming language constructs such as packages, procedures, functions.

A top-down approach to design is widely recommended in conjunction with a functional decomposition.

2.1. Design methods

In many organizations an ad-hoc software design is produced. A more methodical approach to software design is proposed by 'structured design':

- Constantine & Yourdon
- structured systems analysis
- RAPID/USE
- Jackson system development (JSD - 1983)
- MASCOTT: for real time systems

Some CASE tools to support particular methods were developped. Many of them use a diagrammatic approach: data flow diagrams, structure charts. Three types of notations: graphical notation, informal text and program description languages (PDLs).

2.2. Structure charts

Structure charts are a graphical means of showing the hierarchical component structure of a system. Structure charts show how diagrams can be realized as a hierarcy of program units.

Functions are represented as rectangles and data stores as round-edged rectangles and user inputs as circles.

Example. Let T be a text file. Write in a text file C the number of characters in file T and in S write the lines of T but in ascending order.

A data flow diagram solution is:

A structure chart for this problem is:

A CLI solution is:

Some steps in converting a data floww diagram into a structure chart are:
- identification of the highest level input/output units
- identification of the main transformations.

2.3. Data dictionaries

Data dictionaries are an appropriate way to link descriptive and diagrammatic design description.

Example

```
**************************************************************
Entity name | Type     | Description
**************************************************************
Design name | Date     | The name of the design to be processed
--------------------------------------------------------------
Get design  | Transfor | Communicates with the user
name        | mation   |
--------------------------------------------------------------
Sorted enti | Data     |  The names of design entities in ty names   |          |  ascending order
--------------------------------------------------------------
```

2.4. Program Description Language

In the requirements specification phase a language for description of the problem may be used. A PDL is a language derived from a programming language like Ada but it may contain additional, more abstract constructs to increase its expressive power. The advantages of using a PDL is that it may be checked syntactically and semantically by software tools.

FILE: CURS3.SE

3. Case study for requirements specification and software design for Library system

3.1. Requirements specification

Introduction

This document describes the requirements of a software engineering project dealing with information related to books contained in a public library and persons from a given institution having access to this library. The main data and transformations applied to these ones are specified.

3.1.1. The information repositry

Books information:
- title
- author(s)
- editor
- publication year
- code
- status (in_library, loaned, restricted loan)
- identification
  - field (maths, comp. sci., others)
  - type (introductory text, textbook, advanced)
  - purpose (students, researchers, pupils)
- loaning date.

Persons information:
- name - profession
- age

It is assumed that each book is assigned for a specified period of three weeks.

3.1.2. System Functions

The following functions are identified::
a. Loan a book
b. Return a book
c. List the books having the same
  - author or
  - field or
  - purpose or

- status.

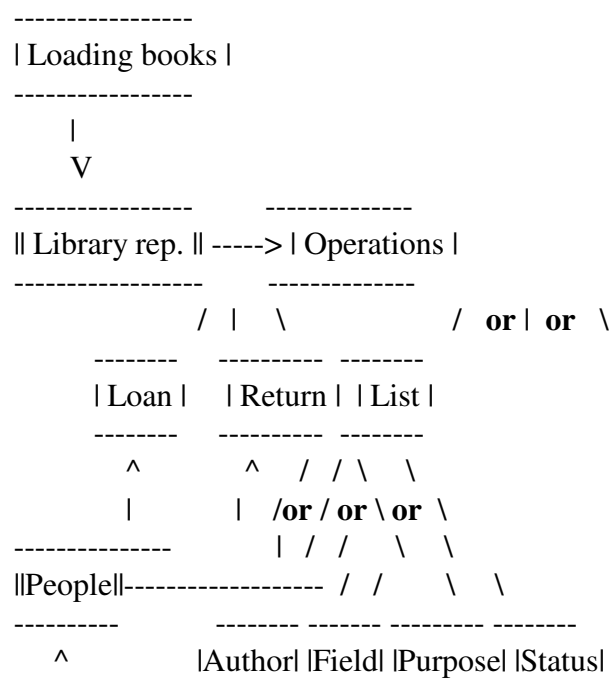
3.1.3. User interface

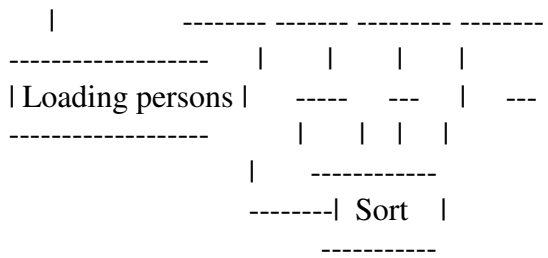
A menu window is available with the options:
1 Loan
2 Return
3 lI**I**st
choose the option:
after **1** or **L**, **2** or **R** the following text issues:
code:...
date:...
and after**3** or **I** the following submeniu appears:
1 Author
2 Field
3 Purpose
4 Status
choose the option:
A list of all the books having one of the mentioned alternatives is produced. The elements of this list are sorted after author in the cases 2, 3, 4 and after title in the first case.

3.1.4. Other operations

Loading the information into the system. The user interface will be specified.

3.1.5. Data flow diagram for this project





3.1.6. Structured description for Field

Function Search for all the books in Library repository having the same field.
Description All the books in Library having the same given field are identified to be listed with the related information.
Input Library repository
Output none
Requires List function and Loading books
Destination Sort function
Pre-condition There exists the specified Field
Post-condition All the books with the specified Field

3.1.7. Formal specification for List using Z

Let

Books - the set of all possible books. A book has the form
(title, author, editor, ..., purpose,...)
Authors - the set of all authors
Purpose_type = [students, researchers, pupils]
Field_type = [maths, comp.sci., others]
Status_type = [in_library, loaned, restricted_loan]

Books_description
---
books: P(Books)
books_set!: P(Books)

Purpose

Books_description
purpose?: Purpose_type purpose_books: Books -------> Purpose_type
---
dom books_purpose = books
books_set! = purpose_books~(purpose?)

Author

Books_description
author?: Authors
author_books: Books --------> Authors
---
dom author_books = books
books_set! = author_books~(author?)

Field

Books_description
field?: Field_type
field_books: Books ---------> Field_type
---
dom field_books = books
books_set! = field_books~(field?)

Status

Books_description
status?: Status_type
status_books: Books --------> Status_type
---
dom status_books = books
books_set! = status_books~(status?)

Using the above defined schemas, the notation for List will be

List ::= Author **or** Field **or** Purpose **or** Status

3.2. Software design

The software product will be a menu driven system. A menu having the description given in 1.3 will appear at any time after loading the program.

3.2.1. Main operations

3.2.1.1. Loan/Return a book

A person takes (returns) a book. In this case the status of that book is changed to 'loaned' ('in_library') and to the involved person is assigned (released) the book. Restriction: maximum 10 books are assigned to each person. The book is identified by code and the person by name (see 1.1).

3.2.1.2. List of the books with the same property

All the books with the same property are selected from the database Library and sorted in ascending order. All the characteristics of each book are listed (see 1.1 and 1.3).

3.2.2. Databases

3.2.2.1. Library

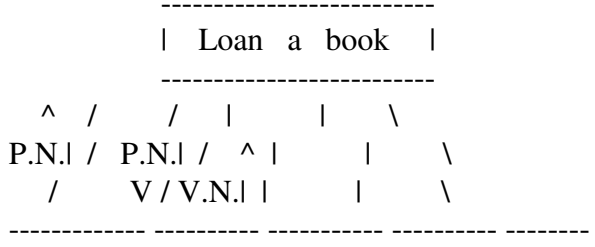
This DB contains all the information about books. All codes are used for each book.

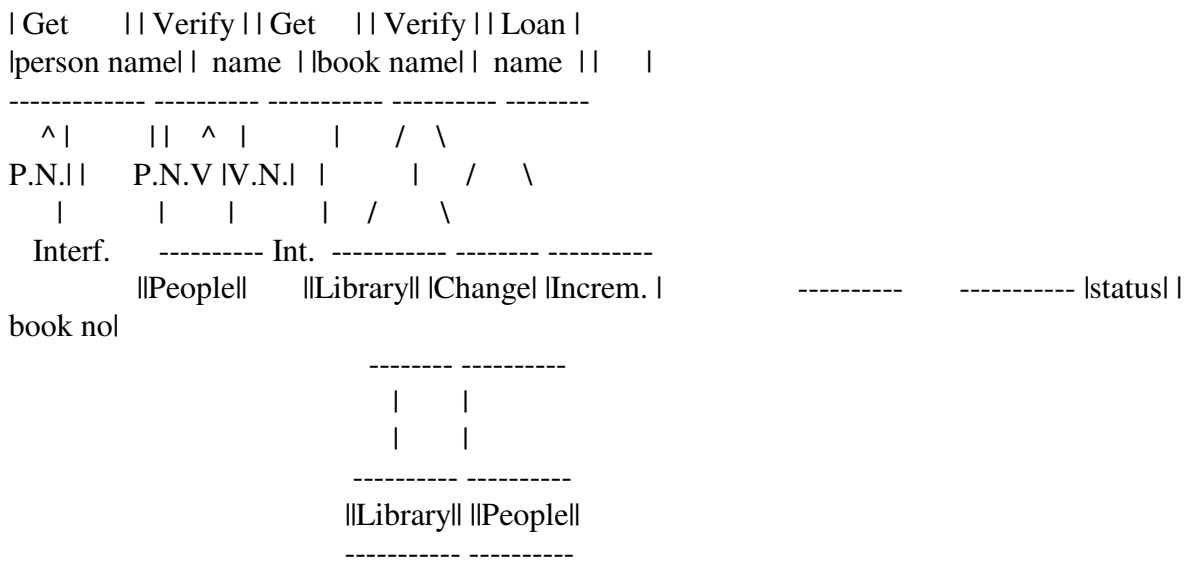
3.2.2.2. People

People DB contains the information related to person, described in 1.1.

3.2.3. Structure charts presentation

3.2.3.1. Loan a book





3.2.4. PDL description for Return

```
package Return_books is
    Read_person_name(out name: string);
    Verify_name(in name: string; out Ok: Boolean);
    -- verify if name is in People => Ok := True
    loop
        exit when no books;
        Read_book_name(in code: code_type);
        Identify_book(in code: code_type; out Ok: Boolean);
        -- verify if code is in Library => Ok := True
        Change_status;  -- status will be in_library
        Decrement_no(in name: string);
        -- decrement the number of assigned books
    end loop;
end Return_books;
```

Exercises

1. Give other structured descriptions and formal notations, taking into account functions presented in 1.2.
2. Introduce error conditions in Purpose.
3. Give structure charts for Return a Book and List the books.
4. Give a PDL description for Loan and List.

FILE: CURS4.SE

4. User interface design

4.1. Design principles

(1) The interface should use terms, objects and concepts which are familiar to the anticipated class of users.
(2) The interface should be consistent.
(3) The user should not be surprised by the system.
(4) The interface should include some mechanism which allows users to recover from their errors.
(5) The interface should include some form of oser guidance.

4.2. Graphical user interface

This type of interface is characterized by:
(1) Multiple windows allowing different information to be displayed simultaneously on user's screen.
(2) Iconic information representation. On some systems, icons represent files, processes.
(3) Command selection via menus rather than a command language.
(4) A pointing device such a mouse, is used.
(5) Support for graphical as well as textual information display.

4.2.1. Direct manipulation

A direct manipulation interface presents users with a model of their information space and they modify their information by direct action.
Examples: a)insert text to the cursor position in a text processor; b)a graphical user interface in which a list of filenames is presented.

4.2.2. Interface models

The control panel model presents the following entities:
(1) Buttons. Picking a button a single action is initiated.
(2) Switches. A switch has many states.
(3) Menus. Collection of buttons and switches which causes a pull-down menu to appear.
(4) Lights. are activatedwhen some actions are taking place.
(5) Displays. Areas of textual or graphical information.
(6) Sliders. INput devices used to set a specified value.

4.2.3. Menu systems

In a menu interface, users select one of a number of possibilities.
Solutions:
(1) scrolling menus; impractical

(2) wallking menus; when a menu is selected a further menu will be displayed.

(3) hierarchical menus.

4.2.4. Information display

(1) Text display:

Jan	Feb	March
123	246	369

(2) Histogram display

(3) Pie chart; thermometer; horizontal bar

4.2.5. Colour display

Colour gives the user interface designer an extra dimension which can be exploited in the display of complex information structures. A number of guidelines for the effective use of colours:
(1) No more than 4-5 separate colours in the same window and no more than 7 in the same system
(2) Design first for monochrome and then add colours to the system.
(3) Use colour codind in a consistent way. The same colour for similar or identical actions.
(4) Sometimes using better graphics is better than using colours.

4.3. Command interface

There are systems (ex: operating systems) having languages for commands: batch files (MS-DOS), shell language(UNIX).
Is recommended to use combinations of command languages and graphical user interface.

4.4. User guidance

The on-line helps are the most used guidance information systems.

References

I. Sommerville: Software Engineering

FILE: CURS5.SE

5.Programming techniques and environments

5.1. Programming for reliability

The programming techniques described can be used in the development of software with high reliability requirements. Fault avoidance and fault tolerance features are investigated. Fault avoidance means utilizing development techniqueswhich reduce the probability of introducing faults into a program; fault tolerance is concerned with writing the program so that it will continue to operate in the presence of software faults.

5.1.1. Fault avoidance

Fault-free software in this context means software which conforms to its specification. Developing fault-free software is very expensive and, as faults are removed from a program, the cost of finding and removing remaining faults tends to rise exponentially.

Fault avoidance and the development of fault-free software rely on:
(1) The production of a precise (preferably formal) system specification.
(2) The adoption of an approach to software design which is based on information hiding and encapsulation.
(3) The extensive use of reviews in the development process which validate the software system. (4) The adoption of an organizational quality philosopy where quality is the driver of the software process.
(5) The careful planning of system testing to expose faults which are not discovered during the review process and to assess system reliability.

It is essential that a high-level programming language with strict typing be used for system development.

5.1.1.1. Structured programming

This term was introduced in the late 1960s to mean programming without using goto statements. Only while loops and if statements are used as control structures. Structured programming is important because its disciplined use of control structures forces programmers to think carefully about their program. However, avoiding unsafe control statements is only the first step in programming for reliability. There are several other constructs in programming languages which are also error-prone.
(1) Floating-point numbers are inherently imprecise and present a particular problem when they are compared because representation imprecision may lead to invalid comparisons.

(2) Pointers are low-level constructs which refer directly to areas of the machine memory. They are dangerous because they allow 'aliasing'. This makes programs harder to understand. It is often impractical to avoid the use of pointers but their use should normally be confined to abstract data type implementation.
(3) Parallelism is inherently dangerous becuase of the difficulties of predicting the subtle effects of timing interactions between parallel processes. Parallelism may be unavoidable but its use must be carefully controlled to minimize inter-process dependencies.
(4) Recursion can lead to very concise programs but it can be difficult to follow the logic of recursive programs. Errors in using recursion may result in the allocation of all the system's memory as temporary stack variables are created.
(5) Interrupts are a means of forcing control to transfer to a section of code irrespective of the code currently executing. their use must be unavoidable but must be minimized.

5.1.1.2. Data typing

Each program component should only be allowed access to data which it needs to implement its function.

Data types are used in languages like Ada, Modula-2, C++ (also in Turbo Pascal) for information hiding or to enhance program readability:

- enumaration types

```
type Traffic_Light_Color is (red, yellow, green);
Color, Next_Color: Traffic_Light_Color;
```

- range types

```
type Positive is Integer range 1..MAXINT;
x, y, z: Positive;
```

- derived types

```
type Oil_Status is new BOOLEAN;
type Door_Status is new BOOLEAN;
Oil_button: Oil_Status;
Door_Button: Door_Status;
```

The operations associated with a type should be packaged with the type declaration to create an abstract data type. The abstract data type hides information about these operations and about the type representation.

An abstract data type is made up of an **interface specification** and an **implementation part**. The implementation part is hidden to the user.

In Ada abstract data types are implemented using packages. In this case the specification is contained in the specification part of the package and the implementation in the body part othis one.

Example. A package specification for an integer queue:

```
package Queue is
    type Q_Range is range 0..100;
    type Q_Vec is array(Q_RAnge) of Integer;
    type T is record
        The_Queue:   Q_Vec;
        front, back: Q_Range := 0;
        end record;
    procedure Put(IQ: in out T; X: Integer);
    procedure Remove(IQ: in out T; X: out Integer);
    function Is_empty(IQ: T) return Boolean;
end Queue;
```

A package body is

```
package body QUEUE is
    Size:  Integer := 101;
    Level: Q_Range := 0;

    procedure Put(IQ: in out T; X: Integer) is
    begin
        if Level  >= Size then Error;
        end if;
        IQ.The_Queue(IQ.back) := X;
        IQ.back := (IQ.back + 1) mod Size
        Level := Level + 1;
    end Put;

    procedure Remove(IQ: in out T; X: out Integer);
    begin
        if Level <= 0 the Error;
        end if;          X := IQ.The_Queue(IQ.front);
        IQ.front := (IQ.front + 1) mod Size;
        Level := Level -1;
    end Remove;

    function Is_empty ... end Is_empty;
end Queue;
```

Comments. The variables Size and Level are declared in the implementation part of Queue and are hidden. The types, variables, procedures and function defined in the specification part of Queue are visible.

The above mentioned package can be used in the following way

```
package  Sample is

with Queue;
Int_Queue: Queue.T;
```

```
begin
    Queue.Put(Int_Queue, 0);
    Queue.Put(Int_Queue, 1);
end Sample;
```

The same problem can be implemented in Turbo Pascal:

```
unit queue;

interface
type Q_Range = 0..100;
    Q_Vec = array [Q_Range] of Integer;
    T = record
      The_Queue: Q_Vec;
      front, back: Q_Range
      end { T };
procedure Put( var IQ: T; X: Integer);
procedure Remove( var IQ: T; var X: Integer);
function Is_empty( IQ: T): Boolean;
procedure Init_Queue( var IQ: T);

implementation
const Size: Integer = 101;
     Level: Integer = 0;
procedure Error;
begin
    writeln('Error in using an Integer Stack');
    halt;
end;
procedure Put;
begin
    if Level >= Size then Error;
    IQ.The_Queue[IQ.back] := X;
    IQ.back := (IQ.back + 1) mod Size;
    Level := Level + 1;
end; procedure Remove;
begin
    if Level <= 0 then Error;
    X := IQ.The_Queue[IQ.front];
    IQ.front := (IQ.front + 1) mod Size;
    Level := Level - 1;
end;
function Is_empty;
begin
end;
```

```
procedure Init_Queue;
begin
    IQ.back := 0;
    IQ.front:= 0;
end;
end {Queue}.
```

5.1.2. Fault tolerance

There are four activities which must be carried out if a system is to be fault tolerant:
(1) Failure detection
(2) Damage assesment
(3) Fault recovery
(4) Fault repair.

5.2. Software reuse

The approach to reuse is component-oriented. Several types:
(1) Application systems.
(2) Sub-systems
(3) Modules or objects
(4) Functions.

5.2.1. Standards

(1) Programming language standards. Ada, COBOL, C, Fortran, Pascal.
(2) Operating system standards. UNIX and MS-Dos for IBM- compatible.
(3) Window system standards. X-Window system.

FILE: CURS6.SE

6. Testing process

For a more complex system the testing should proceed in stages:
(1) Unit testing treats each component as a stand alone entity.
(2) Module testing. A module contains many procedures, objects, units.
(3) Sub-system testing consists in testing collection of modules. Testing interfaces of sub-systems.
(4) System testing consists in finding errors which result from interractions between sub-systems.
(5) Accetance testing is supplied with tests of system procurer.

Obs
(1) & (2) are component testing
(3) & (4) are integration testing
(5) is user testing.

6.1. Testing strategies

Testing strategies include:
- top-down testing
- bottom-up testing
- thread testing
- stress testing

6.1.1. Top-down testing

Top-down testing involves starting at the sub-system level with modules represented by stubs. The stubs are simple components which have the same interface as the module. After sub-system testing is complete, each module is tested in the same way. The functions are represented by stubs and so on.

Top-down testing is used with top-down program development so that a module is tested as soon as it is coded. If a project is developed in a team, then it is recommended togive the responsability of testing a module to another team member than the developer.

In some cases it is difficult to apply this method. For example, a function which relies on the conversion of an array of objects into a linked list is difficult to be sinulated by a relevant stub.

6.1.2. Bottom-up testing

This method is the converse of top-down testing. It involves testing the modules of the lower levels. It is recommended to combine top-down with bottom-up testing.

6.1.3. Thread testing

This type of testing is used in real time and concurrent processes. To test each path in the program data test are supplied:

6.1.4. Stress testing

Some systems are designed to support a specified load. For example an operating system will manage 200 terminals. In this case it is recommended to study the system under this load because many-times the systems in such cases exhibit severe degradations.

6.2. Black-box vs. white-box testing

6.2.1. Functional testing

In this case the component under testing is a black-box whose behaviour is determined by studying its inputs and the related outputs. Equivalence partitioning is a technique for determining which classes of input data have common properties. These partitions may be identified from the specifications; also correct and incorrect inputs are specified.

Let us consider the following specification for a binary search procedure:

```
procedure Binary_search(Key: ELEM; T: ELEM_ARRAY;
                        Found: in out Boolean;
                        L: in out ELEM_INDEX);
Pre-condition:
  T'LAST - T'FIRST > 0 and Ordered(T)

Post-condition:
  (Found and T(L) = Key) or
  (not Found and not (exists i, T'FIRST<=i<=T'LAST, T(i) = Key))
```

We can distinguish several partitions of tests after the different inputs:

(1) inputs which conforms to the pre-conditions:
1 3 5 7 9
(2) inputs with false pre-conditions:
empty input
1 3 2 5
(3) inputs with the Key element in the array
(4) inputs without the Key element in the array

In the third case will be considered inputs T having one and many element(s), also the key will be the first, last and an arbitrary element.

We refer now to some formal specifications using Z notation to express the requirements of a function called "Inserare_tari". From these specifications we shall produce the associated classes of test partitions. The example is taken from the project "Explozia demografica si implicatiile asupra mediului" by M. Rasuceanu and D.-M. Roharik.

Let us consider the following sets:

Anistatistici = set of all years with statistics
Tari = { x | x=(an_statist, nume_tara, populatie, pop_urbana,
densitate, mortalitate, natalitate)}

and the following schema:

Descriere_tari
---
ans: P(Anistatistici)
tari: P(Tari)
Antara: {(s(1), s(2))| s _ tari}
populatie?: Integer
pop_urbana?: Integer
densitatea?: Integer
mortalitatea?: Integer
natalitatea?: Integer

Inserare_tari

_Descriere_tari
an?: Anistatistici
tara?: Tari

(an? _/ ans &
ans' = ans U {an?} &
tari' = tari U {(an?, tara?, ..., natalitate?)}
)
|
(an? _ ans & ans' = ans &

```
 ( (an?,tara?) _ Antara & tari' = tari)
 |
 ( (an?,tara?) _/ Antara &
          tari' = tari U {(an?, tara?, ... natalitate?)}) )
-----------------------------------------------------
```

From this formal specification , considering that a function will be implemented according to the requirements, it can be deduced the following partitioning of data test:
(1) input with "an" not contained in the existing years recorded in "ans"; in this case it is necessary to specify data for "tara", ..., "natalitate" (as integer values). Data for "ans" and "tari" are added.
(2) input with value for "an" contained in "ans" and values for "an" and "tara" in "tari". In this case no any new information is added.
(3) input with value for "an" contained in "ans" but values for "an" and "tara" not contained in "tari". In this case are specified data for "tara" , ..., "natalitate", but only the information for "tari" are added.

Obs. When new information are added (cases (1) & (3)), the following cases are considered:
- empty file
- the information to be added, will be assigned at
  - the beginning
  - the end
  - an arbitrary position

6.2.2. Structural testing

This is a complementary approach to testing, sometimes called 'white-box' or 'glass-box'. In this case the tester can analyse the code and use knowledge about it and the structure of a component to derive the test data. The binary search routine is used again as an example (in Ada notation).

```
procedure Binary_search(Key: ELEM; T: ELEM_ARRAY;
                        Found: in out Boolean;
                        L: in out ELEM_INDEX) is
  Bott: ELEM_INDEX := T'FIRST;
  Top:  ELEM_INDEX := T'LAST;
  Mid:  ELEM_INDEX;
begin
  L     := (Bott+Top) / 2;              -- 1
  Found := T(L) = Key;
  while Bott <= Top and not Found  loop   -- 2
    Mid := (Top+Bott) / 2;
    if T(Mid) = Key then                -- 3
      Found := TRUE;                    -- 4
      L     := Mid;
    elsif T(Mid) < Key then             -- 5
```

```
    Bott := Mid + 1;                -- 6
  else
    Top := Mid - 1;                 -- 7
  end if;
end loop; end Binary_search;
```

A white box testing strategy is path testing, which exercises every independent execution path through the component. In this case a graph will be constructed using the above notations in which the main control structures are considered (if, case, while loop).

The graph associated to procedure:

The tests resulting from this graph are suggested by the paths:
(1) 1 2 3 4 9
(2) 1 2 3 5 6 8
(3) 1 2 3 5 7 8
(4) 1 2 9

For the sequence
26 28 31 32 40
taking Key = 31, it is obtained the path (4)
taking Key = 28, it is obtained
path (2)
part 2 3 5 7 8 from (3)
and part 2 3 4 9 from (1).

FILE: CURS7.SE

7. Static verification

Static verification techniques do not require the program to be executed. They involve examining the source code of a program and detecting the errors before execution. It results that 60% of the errors in a program can be detected using systematic program inspections. It is suggested that more formal static validation using mathematical verification can detect more than 90% of the errors in a program.

The costs of informal verification is about 100 source lines per hour.

Effective combination of static validation and testing is necessary to achieve high quality software.

7.1. Program inspections

This process is carried out by a small team of at least 4 persons: author, reader (reads the text for inspection), tester (reviews the code from a testing point of view), chairman or moderator.

From 90 to 125 statements per hour can be inspected.

The problems checked during the inspection process:

- all the variables should be initialized before use
- for each conditional statement, the condition should be correct
- loop termination
- check the lower/upper bound of the array
- the correct allocation of dynamic space
- the formal and actual parameters match
- compound statements are correct bracketed
- error conditions are taken into account

7.2. Mathematically based verification

Formal program verification involves proving using mathematical arguments. This research is based on the work of McCarthy, Floyd, Hoare, Dijkstra.

In the axiomatic approach the following correctness formulae is used : {f} P {g}.

If f (the pre-condition) is correct, then after execution of P, g (the post-condition) will be correct.

A general technique is to introduce in a program P in some points P_1, P_2, ..., P_n the assertions A_1, A_2, ..., A_n (logical conditions). To prove that the program is correct between points P_i and P_(i+1), it is necessary to show that the statements cause assertion A_i to be transformed into A_(i+1). If it is shown that A_1 leads to A_2, A_2 to A_3 and so on until all statements have been considered, then A_1 leads to A_n and the program is said to be partially correct.

Let us consider again the binary search procedure with the following assertions:

```
procedure Binary_search(Key..) is
-- Pre-condition: T'LAST-T'FIRST > 0 & Ordered(T) & T'FIRST >= 0
begin
  L := (T'FIRST + T'LAST) / 2;
  Found := T(L) = Key;
  -- A_1: Found & T(L) = Key or
  --      not Found & not Key in T(T'FIRST..Bottom-1) or in
  --                          T(T'Top+1..T'LAST)    while Bottom <= Top and not Found loop
    Mid := (Top + Bottom) / 2;
    if T(Mid) = Key then
      Found := True; L := Mid;
      -- A_2: Key := Mid & Found
    elsif T(Mid) < Key then
      -- A_3: not Key in T(T'FIRST..Mid)
      Bottom := Mid + 1;
      -- A_4: not Key in T(T'FIRST..Bottom-1)
    else
      -- A_5: not Key in T(Mid..T'LAST)
      Top := Mid - 1;
      -- A_6: not Key in T(Top+1..T'LAST)
    end if;
    end loop;
  -- Post-condition: Found & T(L) = Key or
  --        (not Found and not exists i, T'FIRST <= i <= T'LAST,
  --                                T(i) = Key)
end Binary_search;
```

Termination argument

(1) The program contains one loop which terminates when Found = True or Bottom > Top;

(2) if there exists an element with the value = Key, then Found = True;

(3) the for termination condition, Top-Bottom < 0, is guaranteed: if an element matching Key is not found then either Bottom := Mid + 1 or Top := Mid - 1. The effect is to reduce Top-Bottom;

(4) the loop must terminate and consequently the program.

Correctness argument

To prove that this design is correct, it must be shown that the final assertion follows from the initial assertion and the program code.

(0) Pre-condition: sequence of non-empty and ordered numbers

(1) A_1: the loop invariant specifies:

- the value of the mid-point matches Key or
- the value matching Key does not lie in the portion examined;

(2) A_2: follows from successful test Key = Mid then Found = True

(3) A_3: T is ordered and T(Mid) < Key

(4) A_4: follows by substituting Bottom-1 for Mid
(5) A_5: similar to A_3
(6) A_6: follows by A_5 - substituting Top+1 for Mid
(7) A_7: follows from loop invariant
(8) the Binary_search procedure is correct

FILE: CURS8.SE

8 Software management

8.1. Project Planning and Scheduling

When formulating a software project plan, the project manager is usually presented with a set of goals which must be achieved - **essential goals** - and **goals** which are **desirable** but not essential. Some of the goals may be mutually opposing. For example: minimization of the project costs vs maximization of system reliability.

Project specific goals might be:

- high maintainability
- low cost
- high reliability

Example:

Option	Cost mill	Schedule months	Reliability coeff	Reuse %	Portability %	Efficiency
A	1.2	33	5	40	90	0.35
B	0.8	30	9	40	75	0.75
C	1.75	36	13	30	30	1

Polar graph

This method derives from techniques used in computer performance evaluation - Boehm (1981).

The option which offers the best overall payoff is the one which encloses the greatest area.

8.1.1. Project scheduling

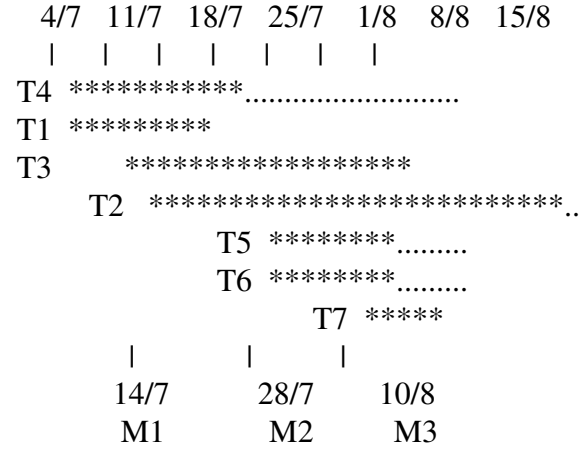
Project scheduling involves separating the total work contained in a project into separate tasks and assessing when these tasks will be completed.

Representations:

- bar chart
- network of activity
- staff allocation

Bar chart

A bar chart is represented in the following way:



- each rectangle (*****) specifies the duration of the task
- the possible delay without critical effects is shown by the rectangle (.....)
- some tasks could be developed in parallel others in sequence

Network of activity

Staff allocation

Tast	Programmer	Checker
T1	Name_1	Name_2
T2	Name_3	Name_4
T3	Name_5	Name_2
....		

In staff allocation it is necessary to consider that not all staff will be occupied all the time on the project. During this period it is necessary to take into account: holiday time, working on other projects, attending training courses, engaged in other activities.

8.2. Software cost estimation

The software estimation cost activity is carried out in tandem with project scheduling. The main components of project costs are: - hardware costs (if necessary)
- travel and training costs
- software engineers costs

Softwaare cost estimation is a continuous activity starting at the proposal stage and continues throughout the lifetime of a project. Projects normally have a budget, and continual cost estimation is necessary to ensure that spending is in line with the budget.

Boehm (1981) discusses seven different techniques of software cost estimation:
(1) algorithmic cost modelling
(2) expert judgement
(3) estimation by analogy
(4) Parkinson's law
(5) pricing to win
(6) top-down estimation
(7) bottom-up estimation

(1) a model is developed using related information
(2) some experts in software development techniques are consulted
(3) this technique is applied when other projects in the same application domain have been completed
(4) expressed in the well-known man/month report
(5) the software cost is estimated to be whatever the customer has available to spent on the project (!)
(6) the costs are examined by considering the functionality of the product
(7) the cost of each component is estimated; all these costs are added to produce a final cost estimate.

We will concentrate on technique (1) which is objective but is not necesarily more accurate than approaches to cost estimation.

8.2.1. Algorithmic cost modelling

A mathematical formula or formulae can be established by linking costs with one or more metrics such as
- project size
- number of programmers

There are a number of algorithmic models, but there are a wide range of discrepancies between models. Examples:
- cost estimation: $362,000 - $2,766,667
- cost effort: 230 person-months - 3857 person-months.

The most commonly used metric for cost estimation is the number of source code.

8.2.2. The COCOMO model

The model exists in **basic** (a) and **intermediate** (b) forms. The model assumes that
- software requirements are relatively stable
- the project will be well managed by both the customer and the software developer.

a. The **basic COCOMO model** gives an order of magnitude for software estimation costs. It uses
- the estimated size of software project
- the type of software being developed

Three versions of the estimation formula depending on the class of software project:
(1) organic mode projects: small team, a well known environment, well understood applications;
(2) semi-detached mode projects: team with experienced & inexperienced staff; members may be unfamiliar with some aspects of the system being developed;
(3) embedded mode projects: system concerned with strongly coupled complex hardware, software, operational procedures; requirements specifications will not be modified; unusual for project team members to have experience in the application being developped.

The effort required for software development is given by:

$$\text{Effort} = A*(\text{KDSI})^b$$

where KDSI is the number of thousands (Kilo) of Delivered Source Instructions; A and b are constants which depend on the type of project

model	A	b
(1)	2.4	1.05
(2)	3	1.12

|(3) | 3.6 | 1.20|

The effort is given in person-months:

In this model a person month (PM) consists in 152 hours of working time.

Time required to complete the project given when sufficient personnel resources are available, is

$TDEV = C*(PM)^{d}$

---
model	C	d
(1)	2.5	0.38
(2)	2.5	0.35
(3)	2.5	0.32

Example Taking a large embedded mode software project consisting of about 128000 DSI, we have
KDSI = 128

$PM = 3.6*(128)^{1.20} = 1216$ person-months

$TDEV = 2.5*(1216)^{0.32} = 24$

Persons_number = 50 (1216/24).

This model uses an implicit productivity:
- organic mode: 352DSI/person-month ==> 16DSI/day
- embedded systems: 105DSI/p-month ==> 4 DSI/day

Observations

1. The function giving TDEV depends on PM and not of the number of software engineers.
2. Adding more people to a project which is behind schedule it is unlikely to help schedule to be regained.

Exercise

Let us take a medium project of type (1) of about 2,000 source lines of Pascal or C. Then the following outcomes are obtained:

PM = $2.4*2^{1.05}$ = 5 person-months

TDEV = $2.5*5^{0.38}$ = 4-5 months,

it follows that 1 people is necessary to carry on this project.

b. **Intermediate COCOMO model** takes into account other factors than project size and type of the project. At this level the two parameters computed by basic COCOMO model are used as starting points and new values are provided by applying a number of multipliers. Fifteen factors (Boehm - 1981) are considered; they are divided into four classes:
- product attributes (1)
- computer attributes (2)
- personnel attributes (3)
- project attributes (4)

In the first class are contained factors like:
- reliability - data base size (low value for size <= 10*DSI; nominal value for size <= 100*DSI and high value > 100*DSI)
- product complexity(low value if it uses simple I/O, simple data structures and straight line code; nominal value if I/O processings, multi-file, library routines, intermodule communication are used; high value is for using re-entrant or recursive code, complex file handling, parallel processing, complex data structures)

In the second class constraints imposed by hardware for software are encountered:
- execution time constraints(nominal <= 50%, high >= 95%)
- storage constraints (nominal <= 50%, high >= 95%)

The third class contains
- analyst capability
- application experience
- programmer capability and programming language experience

The last one describes:
- use of software tools
- project development schedule
- use of modern programming practice (top-down design, code reviews, structured programming, program support libraries).

Some values scaled from low to high are:
- reliability: very high=1.4, very low=0.75
- complexity: very high=1.3, very low=0.7
- memory limitation: high=1.2, none=1
- tool use: high=0.9, low=1.1

Example

Let us suppose that the basic COCOMO gives PM=5
For the intermediate level we take into account: reliability = 1.22; execution time = 1.1; storage contraints = 1.1; tools use = 0.85. It results PM = 6.

References

1. Boehm B. W.: Software Engineering Economics, Englewood Cliffs NJ, Prentice-Hall, 1981.
2. Sommerville I.: Software Engineering, Addison-Wesley, 1993.

FILE: CURS9.SE

9. Software maintenance

Maintenance means the modification of a program after it has been delimited and is in use. Software maintenance falls into three categories: a) perfective maintenance: improve the system
b) adaptive maintenance: required for changes in the environment of the program
c) corrective maintenance: correction of system errors.

Lientz, Swanson (1980) specify that:
- 65% of maintenance is of type a
- 18% of type b
- 17% of type c

Maintenance has a poor image among software engineers. It is seen as a less skilled process than program development and in many organizations, maintenance is allocated to inexperienced staff.

9.1. Maintenance costs

From the existing system it results that maintenance costs are the greatest cost in developing a system.
Factors affecting the maintenance costs are:
(a) a non-technical factors:
- the application being supported
- staff stability
- lifetime of the program
- dependence of the program on its external environment
- hardware stability
(b) technical factors
- module independence
- programming language
- programming style
- program validation and testing
- quality of program documentation

9.2. Maintenance cost estimation

Using data gathered from 63 projects in a number of application areas, Boehm (1981) established a formula for estimating maintenance costs, as a part of the COCOMO software cost estimation model.
ACT - Annual Change Traffic
SDT - Software Development Time
AME - Annual Maintenance Effort

Example: A software project requires 236 PM and the estimated ACT is 15% of the code would be modified, then
AME = 0.15 * 236 = 35.4 PM

This is a basic model. A further refined method is given by considering some other factors. These factors are used as multipliers.
In the above example it is considered that the maintenance cost is affected by: - reliability, very high = 1.10
- staff with experience in programming languages, = 0.91
- staff with experience in applications = 0.95
- use of modern programming languages = 0.72

AME' = AME * 1.10 * 0.91 * 0.95 * 0.72 = 24.2 PM

10. Documentation

Management should pay as much attention to documentation and its associated costs as tp the development of the software itself.

10.1. Document classification

(1) Process documentation - record the process of development and maintenance (plans, schedules, process quality documents);
(2) Product documentation - describes the product which is being developed; describes the product from the point of view of the engineers developing and maintaining the system and user point of view.
Process documentation contains some materials (plans, schedules, reports, standards, working papers etc) which are used during the project development and normally there is no need to preserve them the system has been delivered. Some of these are of interest after development. For example some working papers. Other materials are used by software historians.
Product documentation is generally dealing with user documentation and software engineers documentation, called user documentation and system documentation.
User documentation:

- introductory manual
- installation document
- functional description
- reference manual

Other easy-to-use documentation might be provided:

- a quick reference of system available facilities
- on-line help system

System documentation contains:
(1) the requirements document and the associated rationale (if necessary)
(2) a document describing the system architecture
(3) for each program in the system, a description of the architecture of this program
(4) a description of each component
(5) program source listings; these should be commented appropriately where the comments should explain complex sections;
(6) Validation documents describing how each program is validated and how the validation information relates to the requirements
(7) a maintenance document describing which parts of the system are hardware and software dependent and how evolution of the system has been taken into account.

REFERENCES

Boehm B. W. Software Engineering Economics, Englewood Cliffs NY; Prentice-Hall,1981
Lientz B. P., Swanson E. B. Software Maintenance Management, Reading MA, Addison-Wesley,1980
Sommerville I. Software engineering, Addison_Wesley, 1993.