LIX, École Polytechnique

Software Modelling and Architecture: Exercises

Leo Liberti
liberti@lix.polytechnique.fr

Last update: September 17, 2009

Contents

Chapter 1

Introduction

This exercise book is meant to go with the course on Software Modelling given at École Polytechnique by Prof. D. Krob — the current course edition is 1st semester 2009/2010 (code INF556). This book contains a set of exercises in software modelling and architecture.

1.1 Structure of this book

Software modelling and software architecture are concepts needed when planning complex software systems. The book will focus on exercises to be carried out by means of the UML language, some notions of optimization, and a good deal of common sense. One becomes a good software architect by experience.

Chapter 2 motivates to the study and practice of software design and engineering. Chapter 3 focuses on simple UML exercises. It is split in Sections 3.1 (use case diagrams), 3.2 (sequence diagrams), 3.3 (state diagrams) and 3.4 (class diagrams). Chapter 4 groups various modelling exercises, only some of which involve UML. Chapters 6 and 7 are "large scale exercises" that should give meaningful examples on various modelling techniques used in practical settings (these sometimes employ UML-like diagrams, but are not necessarily based on UML). Since some of these exercises use mathematical programming techniques, there is a small collection of exercises on mathematical programming in Chapter 5.

Chapter 2

Software Design and Engineering

Modern software performs complex tasks, and is actually a mixture of several different software modules working together. Each module might have been created by different people at different times, using different programming languages. Mostly, this complexity is hierarchical: the software modules themselves consists of software submodules, and so on for several levels. Thus, the complexity of the whole software system exceeds the sum of the complexities of each part: it is actually this sum plus the complexity given by the inter-relations. If a software has n modules, its has n^2 potential binary relations (sometimes a module interfaces with another instantiation of itself). Sometimes the relations can range subsets of modules, yielding 2^n relations.

Whereas a module is derived from compiling source code, the relations are abstract notions, occurrences of data exchange, temporal orders. Source code can be looked at "statically"; it is more difficult to "look at" relations without actually simulating or executing them.

With hierarchical complexity, each software module might be the result of the work of one or several development teams. This human effort must be coordinated. Development precedences must be identified and enforced. Extensive testing must be performed. Modules must be coded according to precise requirements. A global design must be put in place for everything to fall exactly into place.

2.1 Good practices

In order to ensure that complex software is well-designed and well-behaved, good software engineering practices must be put in place. Software engineering consists of the following set of disciplines.

- **Software requirements**: The elicitation, analysis, specification, and validation of requirements for software.
- **Software design**: The design of software is usually done with Computer-Aided Software Engineering (CASE) tools and use standards for the format, such as the Unified Modeling Language (UML).
- **Software development**: The construction of software through the use of programming languages.
- **Software testing**
- **Software maintenance**: Software systems often have problems and need enhancements for a long time after they are first completed. This subfield deals with those problems.

- **Software configuration management**: Since software systems are very complex, their configuration (such as versioning and source control) have to be managed in a standardized and structured method.
- **Software engineering management**: The management of software systems borrows heavily from project management, but there are nuances encountered in software not seen in other management disciplines.
- **Software development process**
- **Software engineering tools**
- **Software quality**: verification and validation of software.

2.2 A global software design

A global software design is a formal representation (model) of the whole software system, allowing its analysis prior to actual deployment. As all models are a simplification of reality, global designs, only provide the "big picture", neglecting the details local to each software module composing the system. Surely we expect local occurrences to have a local effect, right? Let us see the effect of very local occurrences in the following examples.

2.2.1 Ariane

The Ariane 5 space exploration rocket exploded on June 4, 1996, burning 7 billion dollars along with its fuselage and its expensive load. Ultimately, the explosion was due to a software module data exchange error.

1. Explosion due to self-destruct mechanism 39 seconds after launch;
2. self-destruct triggered as aerodynamic forces were ripping the boosters from the rocket's fuselage
3. the aerodynamic forces were too strong because the rocket swerved off course
4. the rocket swerved off course because its on-board computer ordered it to do so
5. the computer command was issued because it was compensating for a wrong turn, but *no wrong turn had ever taken place*
6. the computer was instructed about the wrong turn by the inertial system
7. the inertial system uses gyroscopes and accelerometers to track the rocket's motion, and reports the numbers (speeds, angles) to the computer
8. the computer was interpreting as *valid numbers* what was in fact an error message warning that the inertial system had shut down
9. the shutdown occurred because the inertial systems's own computer passed one piece of data written over 64 bits of RAM — the sideways rocket velocity — to a software module (let's call it A) only expecting to be passed a 16 bit wide number
10. upon shutdown the inertial system passed control to an identical system put in place for redundancy
11. the identical system had in fact already shut itself down because of the very same error (it was running the same software)

12. module A only expected a 16 bit wide number because the engineers who designed it never thought the rocket could attain velocities whose numerical representation exceeded 16 bit
13. the engineers thought this because module A was actually designed for the Ariane 4 rocket (less performant than Ariane 5)
14. Ariane 5's software designers re-used Ariane 4's code because one of the principles of software design is re-use.

Furthermore, that particular software module was only useful during the very early takeoff stages, and could have safely been deactivated right after takeoff. It was instead decided to leave it active for 40 seconds in case takeoff countdown was stopped and resumed briefly.

Software module relations often consist in data exchanges or data passing; mistakes in data passing have the potential for erasing memory containing critical code or data. Every error condition must be carefully catered for.

2.2.2 Patriot missile

On February 25, 1991, an American Patriot Missile battery failed to intercept an incoming Iraqi Scud missile, causing 28 deaths and 100 injuries.

1. Time was measured by the system's internal clock in tenths of seconds
2. this number was divided by 10 to yield the number of seconds
3. the calculation was performed in 24 bit arithmetic
4. the binary representation of 1/10 is nonterminating
5. it was chopped at the 24th bit after the radix point, yielding an error of 0.000000095
6. at crash time, the Patriot had been on for 100 hours; multiplying by 0.000000095 gives 0.34s
7. a Scud flies at 1676 m/s, so it travels more than 500m in 0.34s
8. thus, the Patriot did not manage to track the incoming Scud.

Rounding errors cannot be dispensed with, but must be controlled carefully.

2.3 Architecture complexity

A software architecture is a map of the software modules and its relations. A hierarchical architecture extends into mapping each modules into its submodules and their own relations. A relation on a set of objects can be represented formally by means of a *graph*. A graph G is an ordered pair of sets (V, E) where E is a set of subsets of V of cardinality at most 2. The elements of V are called *vertices* and the elements of E are called *edges*. Vertices represent modules and edges represent binary relations on the modules. If the relation is asymmetric then E is a subset of ordered pairs in $V \times V$ and each element of E is an *arc*. Edges and arcs can be labelled by colours or numbers (also called *weights*). In hierarchical architectures, each $v \in V$ is itself a graph (see Fig. 2.1).

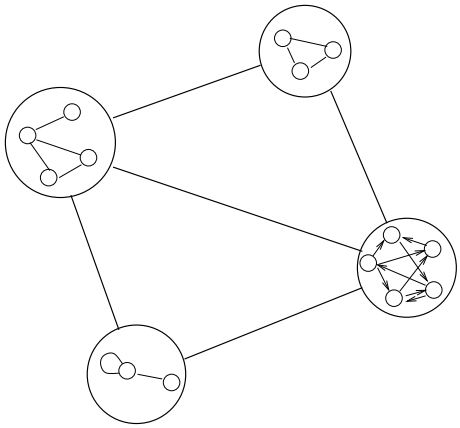


Figure 2.1: A hierarchical software architecture.

Since for n vertices there are n^2 potential edges, and since errors can occur both in software modules and in their inter-relations, one should attempt to reduce the number of relations (perhaps at the expense of increasing the number of software modules) whilst maintaining the same overall functionality.

2.4 Exercises

2.4.1 Graph complexity

Assuming software modules (vertices) and relations (edges) all have unit complexity, simplify the architecture represented by the graph given in Fig. 2.2.

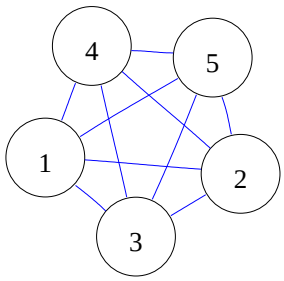


Figure 2.2: Exercise 2.4.1.

2.4.1.1 Architecture complexity

Simplify the architecture given in Fig. 2.3. Black arcs have complexity 1, blue arcs have complexity 3 (two-headed arrows represent two opposite arcs). All vertices (black, yellow, green, blue) have complexity 5.

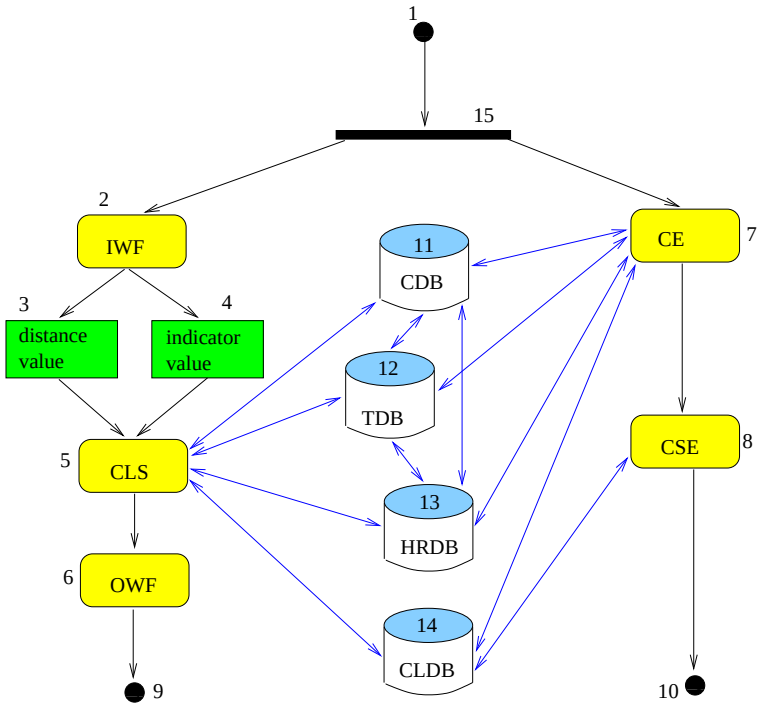


Figure 2.3: Exercise 2.4.1.1.

2.4.2 Complexity comparison

What is the most complex architecture among the ones given in Fig. 2.4? How can they be simplified?

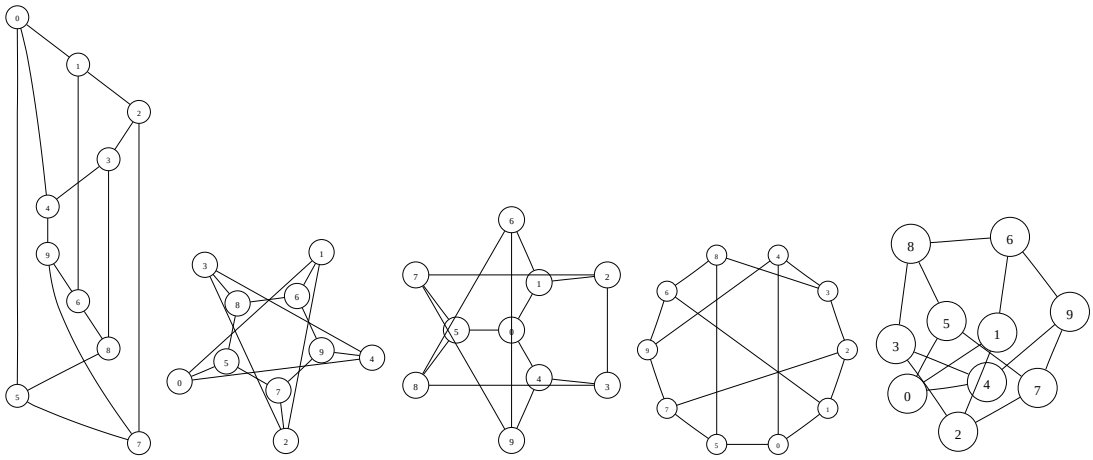


Figure 2.4: Exercise 2.4.2.

Chapter 3

UML exercises

This chapter proposes small to medium scale exercises on UML. Some of them are by the author, whilst others have been taken from books (credits are made explicit in each exercise: where no explicit citation is given, the exercise is to be considered the author's work).

UML is a graphical language consisting of diagrams of many different types, each referring to a *system* (be it a software system or otherwise) seen from a specific point of view. Thus, *use case diagrams* illustrate actors and actions of a system without any reference to logic or the temporal sequence of events; *sequence diagrams* underline the temporal event flow; *state diagrams* encode the logical relations among system components; *class diagrams* are used to represent the system organization into blocks each of which includes data and actions relative to those data. Of these diagrams, the former two (use case and sequence diagrams) are not considered as formal languages but rather as an aid tool to thought and system organization. The latter (state and class) can be considered as formal languages, although, really, compilers exist mostly just for class diagrams (taking a graphical description and outputting corresponding C++ or Java header files). Students should therefore aim at *clarity* for what concerns use case and sequence diagram and at *formal rigourousness* for state and (especially) class diagrams.

UML diagrams can be used as a tool in system design. One usually starts with the simplest type of diagram, i.e. use case diagrams, to make sure the understanding of actors and actions of the system is clear. One then proceeds to sequence diagrams, adding a temporal scale to the system events. Next come state diagrams and finally the closest software representation of the system, the class diagram (there are more than four types of UML diagrams but in this book we shall limit ourselves to these four types). Each step is likely to underline system design errors in previous steps, so that the whole process is a continuous backtrack through use case, sequence, state and class diagrams so that in the end the whole diagram set presents a consistent system picture. Do not eschew backtracking, correcting and re-thinking current diagrams, for this is one of the main system design values added by UML. The student's approach should absolutely *not* be "it took me three hours to put together the use case diagram and several days to compose sequence and state diagrams; and now I'm certainly NOT going to change it just because of one small inconsistency in the class diagram, which is likely to require me to change the whole thing". Usually, one *small* inconsistency in the class diagram is all it takes for the whole system organization to fall apart. The point is that *without* UML the error would have gone unnoticed and likely crept in the system implementation, with catastrophic consequences.

3.1 Use case diagrams

In this section we give some examples of use case diagrams for various situations. In general use case diagrams consists of a system (represented by a large box), actors (represented by small stylized men out

of the box), actions (represented by ellipses within the box) and relations (edges or arcs linking actors with actions, actors with actors, actions with actions).

3.1.1 Simplified ATM machine

Propose a use case diagram for an ATM machine for withdrawing cash. Make the use case simple yet informative; only include the major features.

3.1.2 Vending machine

Propose a use case diagram for a vending machine that sells beverages and snacks. Make use of inclusion and extension associations, mark multiplicities and remember that a vending machine may need technical assistance from time to time.

3.2 Sequence diagrams

In this section we shall present some easy examples of sequence diagrams. These are similar to two-dimensional Euclidean planes, the horizontal axis marking labels for a finite set of *actors* and system *components*, and the (downward) vertical axis representing time. Actors and components are usually derived from a use case diagram. Actions are split into *messages* going back and forth between actors and components. Messages are represented by horizontal arrows. There are two types of messages: *synchronous* (the initiator of the message is blocked until a return value is sent back from the message recipient) and *asynchronous* (the initiator is not blocked and any return value must be carried by a later message where the initiator is the recipient of the asynchronous message).

3.2.1 The norm of a vector

Consider the following algorithm for computing the norm of a vector.

```
Class Array {
  ...
  public:

    // return the size of the array
    int size(void);
    // return the index-th component of the array
    double get(int index);
  ...
};

double norm(const Array& myArray) {
  double theNorm = 0;
  double c = 0;
  for(int index = 0; index < myArray.size() - 1; index++) {
    c = myArray.get(index);
    theNorm = theNorm + c*c;
  }
  theNorm = sqrt(theNorm);
```

```
    return theNorm;
  }
```

Write down a sequence diagram illustrating the behaviour of the `norm()` function.

3.2.2 Displaying graphical objects

Write a sequence diagram for a program that displays Fig. 3.1 on the screen in the order left → right.

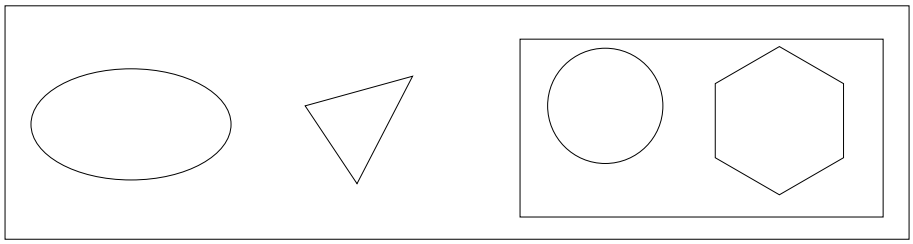


Figure 3.1:

3.2.3 Vending machine

Draw a sequence diagram for the vending machine of Sect. 3.1.2.

3.2.4 Boy/girl interaction

A boy takes a fancy to a girl, and wants her to become his girlfriend. Draw a sequence diagram for their interactions (to avoid any accusation of genderwise discrimination, you can also reverse the roles of boy and girl).

3.3 State diagrams

In this section we present some elementary exercises on state diagrams. State diagrams are similar to state automata, and are used to describe the logic behind any state change within a system or a system component. States are represented by rounded-corner boxes with a label (the state name); a change from state A to state B is represented by an arrow from A to B. Two types of states, "start" and "end", are such that there are no incoming arrows into "start" and no outgoing arrows from "end" states. Arrows are labelled by the name of the activity that caused the state change.

Another type of state diagram, called *activity diagram*, emphasizes activities rather than states: activity names label the round-cornered boxes, and the state names label the arrows. Apart from "start" and "end" nodes, activity diagrams also have special "test" nodes represented by small rhombi.

3.3.1 Vending machine

Draw state and activity diagrams for the vending machine described in Sections 3.1.2 and 3.2.3.

3.4 Class diagrams

In this section we present some elementary exercises on class diagrams.

3.4.1 Complex number class

Draw a class diagram for the single class `Complex`. A `Complex` object has a private real and an imaginary part (of type `double`), and can perform addition, subtraction, multiplication and division by another complex number.

3.4.2 Singly linked list

Draw a class diagram representing a singly linked list.

3.4.3 Doubly linked list

Draw a class diagram representing a doubly linked list.

3.4.4 Binary tree

Draw a class diagram representing a binary tree.

3.4.5 n-ary tree

Draw a class diagram representing an n-ary tree (a tree with a variable number of children nodes).

3.4.6 Vending machine

Draw a class diagram for the vending machine described in Sect. 3.1.2 and 3.2.3.

Chapter 4

Modelling

This chapter groups some modelling exercises, only some of which involve UML.

4.1 The vending machine revisited

Consider the vending machine described in Sect. 3.1.2, 3.2.3 and 3.4.6. The proposed use case diagram, sequence diagram and class diagram make up for a very poor system modelling indeed. The vending machine is always thought of as a monolitic entity: this makes the external relationships clear but says nothing about how to plan and build one. In particular, the monolitic view is incompatible with the fact that a vending machine is composed of different parts. Given the following list of parts:

1. main controller
2. mechanical robot
3. coin acceptor
4. remote messaging system
5. door

and the fact that 2,3,4,5 can only be interfaced with 1, draw a use case diagram and a sequence diagram to provide an initial blueprint for the inner workings of a vending machine.

4.2 Mistakes in modelling a tree

Fig. 4.1 describes the class diagram of a tree node, which can be used recursively to build an expression tree.

Generate the header file and implementation code using Umbrello, then add the implementation of the only non-obvious functions (`getNumberOfChildren` and `getChildType`) as follows:

```
int TreeNode::getNumberOfChildren ( ) {
  // number of children
  int nc = 0;
  switch(m_operatorLabel) {
```

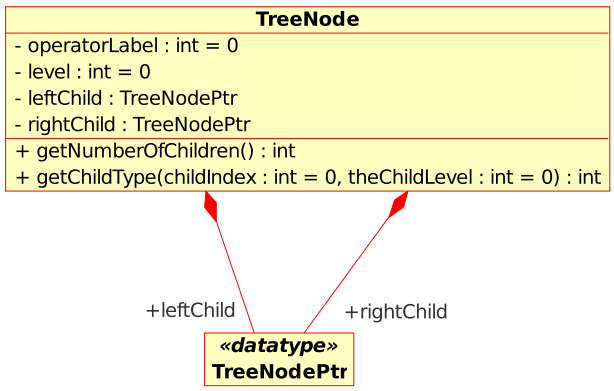


Figure 4.1: The UML class diagram for the `TreeNode` class.

```
  case 0: // sum
    nc = 2;
    break;
  case 1: // difference
    nc = 2;
    break;
  case 2: // multiplication
    nc = 2;
    break;
  case 3: // division
    nc = 2;
    break;
  case 4: // square
    nc = 1;
    break;
  case 5: // cube
    nc = 1;
    break;
  case 6: // sqrt
    nc = 1;
    break;
  case 10: // number
    nc = 0;
    break;
  default:
    break;
  }
  return nc;
}

int TreeNode::getChildType (int childIndex, int theChildLevel) {
  int ret = -1;
  // increase the level by one unit
  theChildLevel++;
  if (childIndex == 0) {
    // left child
    ret = m_leftChild->getOperatorLabel();
  } else if (childIndex == 1) {
    // right child
    ret = m_rightChild->getOperatorLabel();
  }
```

```
    return ret;
  }
```

Now consider the following `main` function in the file `TreeNode_main.cxx`:

```
// TreeNode_main.cxx

#include <iostream>
#include "TreeNode.h"

int main(int argc, char** argv) {

  int ret = 0;

  // expression tree t: number + number^2
  TreeNode t;
  t.setOperatorLabel(0);
  t.setLevel(0);
  t.setLeftChild(new TreeNode);
  t.setRightChild(new TreeNode);
  t.getLeftChild()->setOperatorLabel(10);
  t.getLeftChild()->setLevel(1);
  t.getRightChild()->setOperatorLabel(4);
  t.getRightChild()->setLevel(1);
  t.getRightChild()->setLeftChild(new TreeNode);
  t.getRightChild()->getLeftChild()->setOperatorLabel(10);
  t.getRightChild()->getLeftChild()->setLevel(2);

  // get right child type and level
  int theLevel = 0;
  int theOperatorLabel = -1;
  theOperatorLabel = t.getChildType(1, theLevel);

  // expect theOperatorLabel = 4, theLevel = 1;
  std::cout << theOperatorLabel << ", " << theLevel << std::endl;
  // actual output is 4,0

  return ret;
}
```

Compile the project by typing:

```
c++ -o TreeNode TreeNode_main.cxx TreeNode.cpp
```

and verify whether the output is as expected (`4, 1`). If not, why? Is this a bug or a modelling error?

We would now like to code in `TreeNode_main.cxx` a new function that accepts a tree node and returns the number of children of the root node of the expression tree. Convince yourself that you cannot do this easily, and explain why. How can you fix this modelling error? Change the UML diagram and the code accordingly.

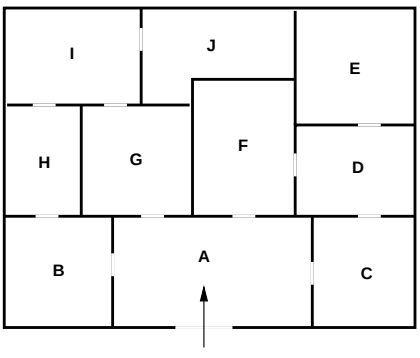
Chapter 5

Mathematical programming exercises

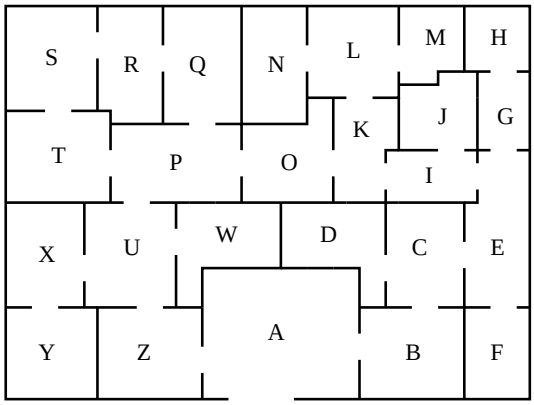
The mathematical programming formulation language is a very powerful tool used to formalize optimization problems by means of parameters, decision variables, objective functions and constraints. Such diverse settings as combinatorial, integer, continuous, linear and nonlinear optimization problems can be defined precisely by their corresponding mathematical programming formulations. Its power is not limited to its expressiveness, but usually allows hassle-free solution of the problem: most general-purpose solution algorithms solve optimization problems cast in their mathematical programming formulation, and the corresponding implementations can usually be hooked into language environments which allow the user to input and solve complex optimization problems easily. This chapter provides an introduction (by way of examples) to a mathematical programming software system, called AMPL (A Mathematical Programming Language) [6] which is interfaced with continuous mixed-integer linear (CPLEX [8]) and nonlinear solvers. See `www.ampl.com` for details on downloading and installing the student versions of AMPL and CPLEX.

5.1 Museum guards

A museum director must decide how many guards should be employed to control a new wing. Budget cuts have forced him to station guards at each door, guarding two rooms at once. Formulate a mathematical program to minimize the number of guards. Solve the problem on the map below using AMPL.



Also solve the problem on the following map.



[*P. Belotti, Carnegie Mellon University*]

5.2 Mixed production

A firm is planning the production of 3 products A_1,A_2,A_3. In a month production can be active for 22 days. In the following tables are given: maximum demands (units=100kg), price (\$/100Kg), production costs (per 100Kg of product), and production quotas (maximum amount of 100kg units of product that would be produced in a day if all production lines were dedicated to the product).

Product	A_1	A_2	A_3
Maximum demand	5300	4500	5400
Selling price	\$124	\$109	\$115
Production cost	\$73.30	\$52.90	\$65.40
Production quota	500	450	550

1. Formulate an AMPL model to determine the production plan to maximize the total income.
2. Change the mathematical program and the AMPL model to cater for a fixed activation cost on the production line, as follows:

Product	A_1	A_2	A_3
Activation cost	\$170000	\$150000	\$100000

3. Change the mathematical program and the AMPL model to cater for both the fixed activation cost and for a minimum production batch:

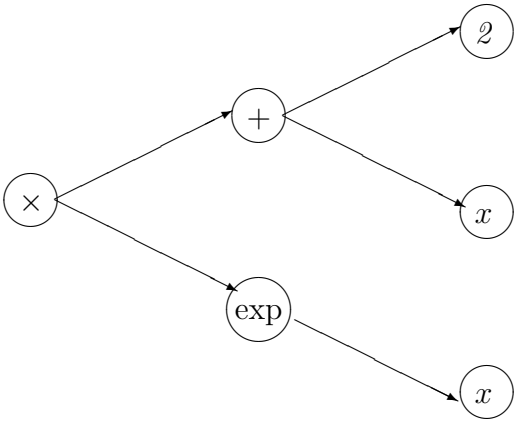
Product	A_1	A_2	A_3
Minimum batch	20	20	16

[*E. Amaldi, Politecnico di Milano*]

5.3 Checksum

An *expression parser* is a program that reads mathematical expressions (input by the user as strings) and evaluates their values on a set of variable values. This is done by representing the mathematical

expression as a directed binary tree. The leaf nodes represent variables or constants; the other nodes represent binary (or unary) operators such as arithmetic (+, -, *, /, power) or transcendental (sin, cos, tan, log, exp) operators. The unary operators are represented by a node with only one arc in its outgoing star, whereas the binary operators have two arcs. The figure below is the binary expression tree for $(x + 2)e^x$.



The expression parser consists of several subroutines.

- `main()`: the program entry point;
- `parse()`: reads the string containing the mathematical expression and transforms it into a binary expression tree;
- `gettoken()`: returns and deletes the next semantic token (variable, constant, operator, brackets) from the mathematical expression string buffer;
- `ungettoken()`: pushes the current semantic token back in the mathematical expression string buffer;
- `readexpr()`: reads the operators with precedence 4 (lowest: +,-);
- `readterm()`: reads the operators with precedence 3 (*, /);
- `readpower()`: reads the operators with precedence 2 (power);
- `readprimitive()`: reads the operators of precedence 1 (functions, expressions in brackets);
- `sum(term` a`, term` b`)`: make a tree $+ \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} a \\ b \end{array}$;
- `difference(term` a`, term` b`)`: make a tree $- \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} a \\ b \end{array}$;
- `product(term` a`, term` b`)`: make a tree $* \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} a \\ b \end{array}$;
- `fraction(term` a`, term` b`)`: make a tree $/ \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} a \\ b \end{array}$;
- `power(term` a`, term` b`)`: make a tree $\wedge \begin{array}{c} \nearrow \\ \searrow \end{array} \begin{array}{c} a \\ b \end{array}$;
- `minus(term` a`)`: make a tree $- \rightarrow a$;
- `logarithm(term` a`)`: make a tree make a tree $\log \rightarrow a$;

- `exponential(term a)`: make a tree make a tree exp $\rightarrow a$;
- `sine(term a)`: make a tree make a tree sin $\rightarrow a$;
- `cosine(term a)`: make a tree make a tree cos $\rightarrow a$;
- `tangent(term a)`: make a tree make a tree tan $\rightarrow a$;
- `variable(var x)`: make a leaf node x;
- `number(double d)`: make a leaf node d;
- `readdata()`: reads a table of variable values from a file;
- `evaluate()`: computes the value of the binary tree when substituting each variable with the corresponding value;
- `printresult()`: print the results.

For each function we give the list of called functions and the quantity of data to be passed during the call.

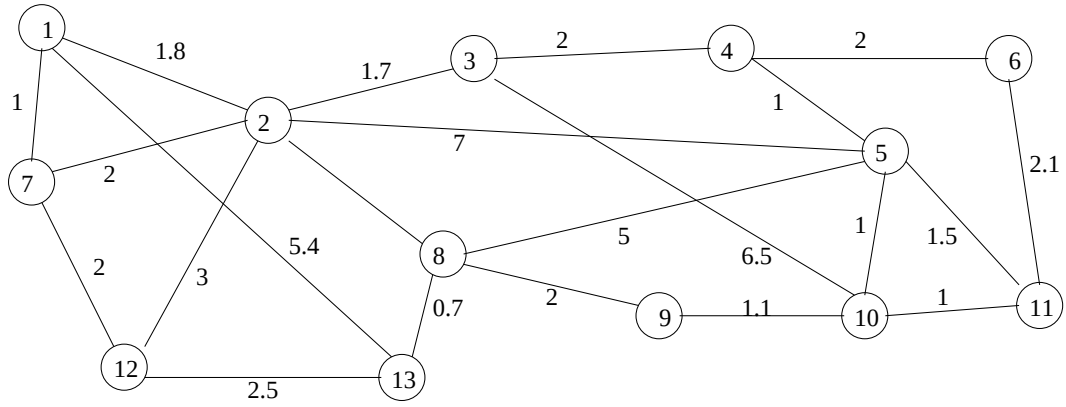
- main: readdata (64KB), parse (2KB), evaluate (66KB), printresult(64KB)
- evaluate: evaluate (3KB)
- parse: gettoken (0.1KB), readexpr (1KB)
- readprimitive: gettoken (0.1KB), variable (0.5KB), number (0.2KB), logarithm (1KB), exponential (1KB), sine (1KB), cosine (1KB), tangent (1KB), minus (1KB), readexpr (2KB)
- readpower: power (2KB), readprimitive (1KB)
- readterm: readpower (2KB), product (2KB), fraction (2KB)
- readexpr: readterm (2KB), sum (2KB), difference (2KB)
- gettoken: ungettoken (0.1KB)

Each function call requires a bidirectional data exchange between the calling and the called function. In order to guarantee data integrity during the function call, we require that a checksum operation be performed on the data exchanged between the pair (calling function, called function). Such pairs are called *checksum pairs*. Since the checksum operation is costly in terms of CPU time, we limit these operations so that no function may be involved in more than one checksum pair. Naturally though, we would like to maximize the total quantity of data undergoing a checksum.

1. Formulate a mathematical program to solve the problem, and solve the given instance with AMPL.
2. Modify the model to ensure that `readprimitive()` and `readexpr()` are a checksum pair. How does the solution change?

5.4 Network Design

Orange is the unique owner and handler of the telecom network in the figure below.



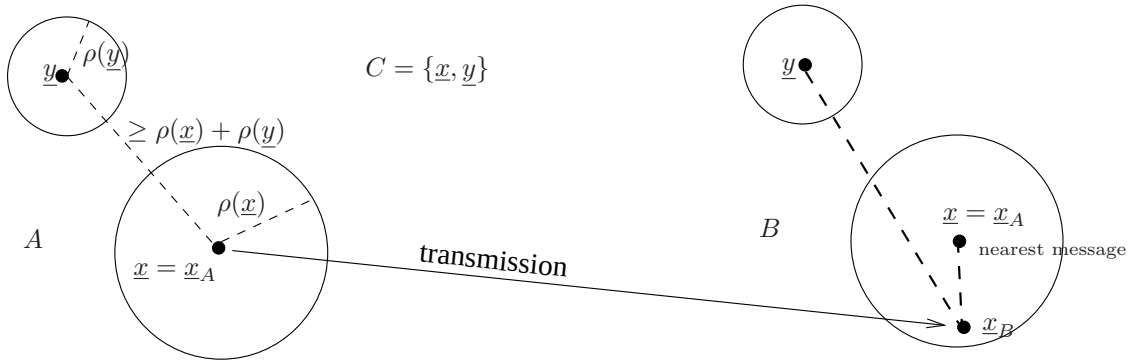
The costs on the links are proportional to the distances $d(i,j)$ between the nodes, expressed in units of 10km. Because of anti-trust regulations, Orange must delegate to SFR and Bouygtel two subnetworks each having at least two nodes (with Orange handling the third part). Orange therefore needs to design a backbone network to connect the three subnetworks. Transforming an existing link into a backbone link costs $c = 25$ euros/km. Formulate a mathematical program to minimize the cost of implementing a backbone connecting the three subnetworks, and solve it with AMPL. How does the solution change if Orange decides to partition its network in 4 subnetworks instead of 3?

5.5 Error correcting codes

A message sent by A to B is represented by a vector $\underline{z} = (\underline{z}_1, \ldots, \underline{z}_m) \in \mathbb{R}^m$. An *Error Correcting Code* (ECC) is a finite set C (with $|C| = n$) of messages with an associated function $\rho : C \to \mathbb{R}$, such that for each pair of distinct messages $\underline{x}, \underline{y} \in C$ the inequality $||\underline{x} - \underline{y}|| \geq \rho(\underline{x}) + \rho(\underline{y})$ holds. The *correction radius* of code C is given by

$$R_C = \min_{\underline{x} \in C} \rho(\underline{x}),$$

and represents the maximum error that can be corrected by the code. Assume both A and B know the code C and that their communication line is faulty. A message $\underline{x}_A \in C$ sent by A gets to B as $\underline{x}_B \notin C$ because of the faults. Supposing the error in $\underline{x}_B$ is strictly less than R_C, B is able to reconstruct the original message $\underline{x}_A$ looking for the message $\underline{x} \in C$ closest to $\underline{x}_B$ as in the figure below.



Formulate a (nonlinear) mathematical program to build an ECC C of 10 messages in $\mathbb{R}^1 2$ (where all message components are in $[0, 1]$) so that the correction radius is maximized.

5.6 Selection of software components

In this example we shall see how a large, complex Mixed-Integer Nonlinear Programming (MINLP) problem (taken from [12]) can be reformulated to a Mixed-Integer Linear Programming (MILP) problem. It can be subsequently modelled and solved in AMPL.

Large software systems consist of a complex architecture of interdependent, modular software components. These may either be built or bought off-the-shelf. The decision of whether to build or buy software components influencese the cost, delivery time and reliability of the whole system, and should therefore be taken in an optimal way. Consider a software architecture with n component slots. Let I_i be the set of off-the-shelf components and J_i the set of purpose-built components that can be plugged in the i-th component slot, and assume $I_i \cap J_i = \emptyset$. Let T be the maximum assembly time and R be the minimum reliability level. We want to select a sequence of n off-the-shelf or purpose-built components compatible with the software architecture requirements that minimize the total cost whilst satisfying delivery time and reliability constraints.

Chapter 6

Log analysis architecture

Some firms currently handle project management in an innovative way, letting teams interact freely with each other whilst trying to induce different teams and people to converge towards the ultimate project goal. In this "liberal" framework, a continual assessment of team activity is paramount. This can be obtained by performing an analysis of the amount of read and write access of each team to the various project documents. Read and write document access is stored in the log files of the web server managing the project database. Such firms therefore require a software package which reads the webserver log files and displays the relevant statistical analyses in visual form on a web interface.

Propose a detailed software architecture consistent with the definitions, goals and requirements listed in Sections 6.1, 6.2, 6.3.

6.1 Definitions

An *actor* is a person taking part in a project. A *tribe* is a group of actors. A *document* is an electronic document uploaded to a central database via a web interface. Documents are grouped according to their semantical value according to a pre-defined map which varies from project to project. There are therefore various *semantical zones* (or simply *zones*) in each project: a zone can be seen as a semantically related group of documents.

A *visual map* of document accesses concerning a set of tribes T and a set of zones Z is a bipartite graph $B_T^Z = (T, Z, E)$ with edges weighted by a function $w : E \to \mathbb{N}$ where an edge $e = \{t, z\}$ exists if the tribe t has accessed documents in the zone z, and $w(e)$ is the number of accesses. There may be different visual maps for read or write accesses, and a union of the two is also envisaged.

A *timespan* is a time interval $\bar{\tau} = [s, e]$ where s is the starting time and e is the ending time. Visual maps clearly depend on a given timespan, and may therefore be denoted as $B_T^Z(\tau)$. For each edge $e \in E$ we can draw the coordinate *time graph* of $w(e)$ changing in function of time (denoted as $w_e(\tau)$ in this case).

6.2 Software goals

The log scanning software overall user goals are:

1. given a tribe t and a timespan $\bar{\tau}$, display a per-tribe visual map $B_{\{t\}}^Z(\bar{\tau})$;

2. given a zone z and a timespan $\bar{\tau}$, display a per-zone visual map $B_T^{\{z\}}(\bar{\tau})$;
3. given a timespan $\bar{\tau}$, display a global visual map $B_T^Z(\bar{\tau})$;
4. given a timespan $\bar{\tau}$ and an edge $e = \{t, z\}$ in $B_T^Z(\bar{\tau})$, display a time graph of $w(e)$.

The per-tribe and per-zone visual maps can be extended to the per-tribe-pair, per-tribe-triplet, per-zone-pair, per-zone-triplet cases.

6.3 Requirements

The technical requirements of the software can be subdivided into three main groups: (a) user interaction, (b) log file reading, (c) computation and statistics.

6.3.1 User interaction

All user interaction (input and output) occurs via a web interface. This will:

1. configure the desired visual map (or time graph) according to user specification (input action);
2. delegate the necessary computation to an external agent (a log database server) and obtain the results (process action);
3. present the visual map or time graph in a suitable graphical format (output action).

6.3.2 Log file reading

Log file data will be gathered at pre-definite time intervals by a daemon, parsed according to the log file format, and stored in a database. The daemon will:

1. find the latest entries added the log files since last access (input action);
2. parse them according to the log file format (process action);
3. write them to suitable database tables (output action).

6.3.3 Computation and statistics

Actually counting the relevant numbers and types of accesses will be carried out by a database engine. This will receive a query, perform it, and output the desired results.

Chapter 7

A search engine for projects

This large-scale example comes from an actual industrial need. An industry manager once mentioned to me how nice it would be to have a search engine for projects, and how easy their work would be if they were able to come up with relevant past data "at a glance" whenever a decision on a new project has to be taken. Although this example does not use UML (although it does use some diagrams inspired to UML), it employs some novel, partially automatic graph reformulation techniques for manipulating the software architecture graph. This example also shows how optimization techniques and mathematical programming are useful tools in software architecture.

7.1 The setting

T-Sale is a large multinational firm which is often employed by national governments and other large institutions to provide very large-scale services. They will secure contracts by responding to the prospective customers' public tenders with commercial offers that have to be competitive. The upper management of T-Sale noticed some inefficiencies in the way these commercial offers are put together, in that very often the risk analysis are incorrect. They decided that they could improve the situation by trying to use stored information about past projects. More precisely, T-Sale keeps a detailed project database which allows one to see how an initial commercial offer became the true service that was eventually sold to the customer. The management hope that the preliminary customer requirements contained in the public tender may be successfully matched with the stored initial requirements to draw some meaningful inference on how the project actually turned out in the past.

T-Sale wants to enter into a contract with a smaller firm, called VirtualClass, to provide the following service, which was expressed in very vague terms from one senior vice-president of T-Sale to VirtualClass' sales department.

> *We want a sort of "Google" for starting projects. We want to find all past projects which were similar at the initial stage and we want to know how they developed; this should give us some idea of future development of the current project.*

VirtualClass must estimate the cost and time required to complete this task, and make T-Sale a competitive offer. Should T-Sale accept the offer, VirtualClass will then have to actually plan and implement the system. Note:

1. The commercial offer needs to be drawn quickly. The associated risks should be assessed. It should be as close as possible to the delivered product.

2. In general, the software engineering team should follow the "V" development process (left branch) for planning the system, as shown in Fig. 7.1. We shall limit the discussion to the leftmost branch of the "V" process.

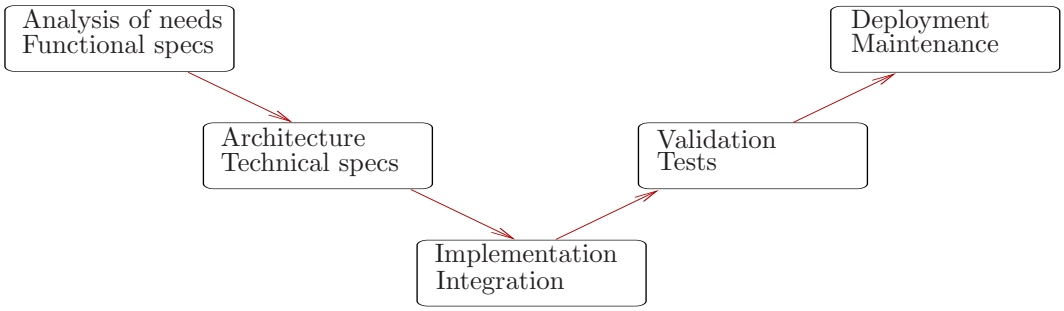


Figure 7.1: The "V" development process.

7.2 Initial offer

7.2.1 Kick-off meeting

Aims of the meeting:

1. Formalize the customer requirements as much as possible
   (a) What is the deliverable, i.e. what is actually sold to the customer?
   (b) What is the first coarse "common-sense" system breakdown?
2. What data is needed from T-Sale's databases?

7.2.2 Brainstorming meeting

Aims of the meeting:

1. propose ideas for a system plan with sufficient details for a rough cost estimate;
2. collect these ideas in a formal document;
3. decide on a sexy project name.

7.2.3 Formalization of a commercial offer

Aims of the meeting:

1. write a document (for internal use) which gives a rough overview of the system functionalities and of the system breakdown into sub-systems and interdependencies;
2. write a document (for internal use) with projected sub-system costs (complexity) and a rough risk assessment;
3. write a commercial offer to be sent to T-Sale with functionalities and the total cost.

7.3 System planning

We shall now suppose that T-Sale accepted VirtualClass' offer and is now engaged in a contract. The next step is to actually plan the system. The contract clearly states that T-Sale is under obligation to provide T-Sale with database details, which are shown in Fig. 7.2.

7.3.1 Understanding T-Sale's database structure

Aims of the meeting: analysis and documentation of T-Sales' database structure. Note that the project's *condition* contains information about whether the project was a success or a failure, and other overall properties. Make sure every software engineer understands the database structure by answering the following questions:

1. How do we find the main occupation of an employee?
2. How do we find the expertises of an employee?
3. How do we find the condition of a project?
4. How do we find how many times a project was changed?
5. How do we find whether a project was paid for on time or late?
6. How do we find whether a customer usually pays on time or late?
7. How do we verify that the cost of all phases in a project sums up to the total project cost?
8. How do we evaluate the cost in function of time during the project's lifetime?
9. How do we discriminate between the phase cost due to human resources and the cost due to other reasons?
10. How do we find the expertises (with their levels) that were necessary in a given project?
11. How do we find out the abilities and skills (with their levels) that were necessary in a given project?
12. How do we find out which teams were most successful?
13. How do we find out the most dangerous personal incompatibilities?

7.3.2 Brainstorming: Ideas proposal

The commercial offer quotes: "Given some meaningful key-words or other well-defined indicators in the description of a new project, we want to classify it by some quantitative indices [...]". Such concepts as "meaningful key-words or other well-defined indicators" and "quantitative indices" are not well-defined, and therefore pose the most difficult problem to be solved in order to arrive at a software architecture. In order to solve the problem, a brainstorming meeting is called.

Aim of the meeting:

1. find a set of well-defined new project indicators which are suitable for searching similar terms in the T-Sale database;
2. find a set of quantitative indices to be computed using the T-Sale database information, which should shed light on the future life cycle of the new project;
3. document all ideas spawned during the meeting in a formal document.

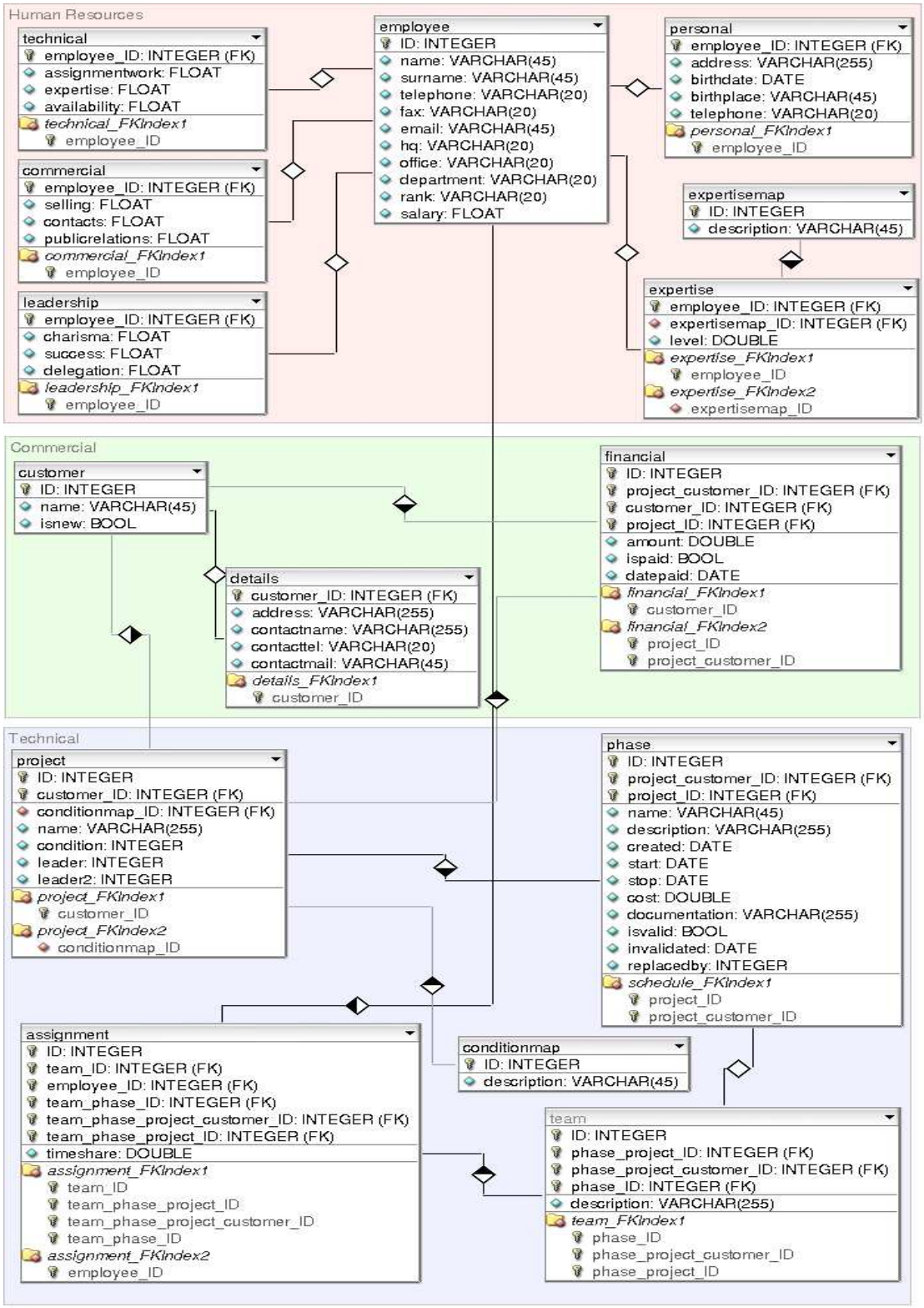


Figure 7.2: T-Sales' database structure.

7.3.3 Functional architecture

Propose a functional architecture for the software. This should include the main software components and their interconnections, as well as a break-down of the architecture into sub-parts so that development teams can be formed and assigned to each project part. Since system-wide faults arise from badly interacting teams, it is naturally wise to minimize the amount of team interaction needed.

7.3.4 Technical architecture

Propose a technical architecture detailing the inner working of each system component, as well as the system as a whole. This should include a class diagram and component APIs (application programming interfaces).

Bibliography

[1] R. Battiti and A. Bertossi. Greedy, prohibition and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[2] A. Billionnet, S. Elloumi, and M.-C. Plateau. Quadratic convex reformulation: a computational study of the graph bisection problem. Technical Report RC1003, Conservatoire National des Arts et Métiers, Paris, 2006.

[3] M. Boulle. Compact mathematical formulation for graph partitioning. *Optimization and Engineering*, 5:315–333, 2004.

[4] C.E. Ferreira, A. Martin, C. Carvalho de Souza, R. Weismantel, and L.A. Wolsey. Formulations and valid inequalities for the node capacitated graph partitioning problem. *Mathematical Programming*, 74:247–266, 1996.

[5] R. Fortet. Applications de l'algèbre de boole en recherche opérationelle. *Revue Française de Recherche Opérationelle*, 4:17–26, 1960.

[6] R. Fourer and D. Gay. *The AMPL Book*. Duxbury Press, Pacific Grove, 2002.

[7] Object Management Group. Unified modelling language: Superstructure, v. 2.0. Technical Report formal/05-07-04, OMG, 2005.

[8] ILOG. *ILOG CPLEX 8.0 User's Manual*. ILOG S.A., Gentilly, France, 2002.

[9] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, 37:865–892, 1989.

[10] L. Liberti. Compact linearization for bilinear mixed-integer problems. `www.optimization-online.org`, May 2005.

[11] L. Liberti. Compact linearization of binary quadratic problems. *4OR*, 5(3):231–245, 2007.

[12] P. Potena V. Cortellessa, F. Marinelli. Automated selection of software components based on cost/reliability tradeoff. In V. Gruhn and F. Oquendo, editors, *EWSA 2006*, volume 4344 of *LNCS*, pages 66–81. Springer-Verlag, 2006.