LIX, École Polytechnique

Software Modelling and Architecture: Exercises

Leo Liberti
liberti@lix.polytechnique.fr

Last update: October 8, 2010

Contents

1 Introduction **7**

1.1 Structure of this book 7

2 Software Design and Engineering **9**

2.1 Good practices 9

2.2 A global software design 10

2.2.1 Ariane 10

2.2.2 Patriot missile 11

2.2.3 Smashing the stack 11

2.3 A graph model for architectures 12

2.3.1 Edge complexity 12

2.3.2 Elementary definitions 12

2.3.3 Graph operations 13

2.3.4 Edge deletion 14

2.3.5 Closure 14

2.3.6 Contraction 14

2.3.7 Non-complete extended p-sum 14

2.4 Exercises 15

2.4.1 Graph complexity 15

2.4.2 Architecture complexity 16

2.4.3 Complexity comparison 16

2.4.4 NEPS 18

3 UML exercises **19**

3.1 Use case diagrams 19

3.1.1 Simplified ATM machine . . . 20
3.1.2 Vending machine . . . 20
3.2 Sequence diagrams . . . 22
3.2.1 The norm of a vector . . . 22
3.2.2 Displaying graphical objects . . . 23
3.2.3 Vending machine . . . 24
3.2.4 Boy/girl interaction . . . 25
3.3 State diagrams . . . 25
3.3.1 Vending machine . . . 26
3.4 Class diagrams . . . 28
3.4.1 Complex number class . . . 28
3.4.2 Singly linked list . . . 33
3.4.3 Doubly linked list . . . 33
3.4.4 Binary tree . . . 34
3.4.5 n-ary tree . . . 34
3.4.6 Vending machine . . . 35

4 Modelling **37**
4.1 The vending machine revisited . . . 37
4.1.1 Solution . . . 37
4.2 Mistakes in modelling a tree . . . 37
4.2.1 Solution . . . 41

5 Mathematical programming exercises **43**
5.1 Museum guards . . . 43
5.1.1 Solution . . . 44
5.2 Mixed production . . . 46
5.2.1 Solution . . . 46
5.3 Checksum . . . 48
5.3.1 Solution . . . 50
5.4 Network Design . . . 53
5.4.1 Solution . . . 54
5.5 Error correcting codes . . . 57

5.5.1 Solution 57
5.6 Selection of software components 58
5.6.1 Solution 58

6 Log analysis architecture **61**
6.1 Definitions 61
6.2 Software goals 61
6.3 Requirements 62
6.3.1 User interaction 62
6.3.2 Log file reading 62
6.3.3 Computation and statistics 62
6.4 The software architecture 62
6.4.1 Summary 63
6.4.2 Details 65

7 A search engine for projects **71**
7.1 The setting 71
7.2 Initial offer 72
7.2.1 Kick-off meeting 72
7.2.2 Brainstorming meeting 73
7.2.3 Formalization of a commercial offer 75
7.3 System planning 77
7.3.1 Understanding T-Sale's database structure 77
7.3.2 Brainstorming: Ideas proposal 79
7.3.3 Functional architecture 80
7.3.4 Technical architecture 88

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 2010/2011 (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.

2.2.3 Smashing the stack

Back in 1996, the online `Phrack` magazine, focusing on exploiting computer security, ran an article which soon became famous: *Smashing the stack for fun and profit*, by `Aleph1` (see Fig. 2.1. It is a technical paper explaining how to spawn a root shell on Unix systems. Such attacks are based on the fact that several programmers use C functions such as `strcpy`, which does not check the length of the destination array before copying the source. Because of the way compilers arrange data in memory, statically allocated arrays are allocated in the same memory area as the address for executing the next machine language instruction. By copying on a static arrays more characters than its length, it is possible to overwrite the bytes storing the next executable instruction address with arbitrary values. This allows one to execute arbitrary code stored in memory.

```
                    .oO Phrack 49 Oo.

             Volume Seven, Issue Forty-Nine

                      File 14 of 16

           BugTraq, r00t, and Underground.Org
                        bring you

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          Smashing The Stack For Fun And Profit
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                      by Aleph One
                 aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine.  Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address.  This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.
```

Figure 2.1: The abstract from *Smashing the stack for fun and profit.*

Security-awareness is an essential coding practice. Moreover, the properties of programs must be verified formally.

2.3 A graph model for architectures

A software architecture is a map of the software modules and its 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*; the elements of E of cardinality 2 are called *edges*; the elements of E with cardinality 1 are called *loops*. 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*. Vertices, edges and arcs can be labelled by colours, numbers (also called *weights*) or other data types. In hierarchical architectures, each $v \in V$ is itself a graph (see Fig. 2.2).

2.3.1 Edge complexity

Since for n vertices there are n^2 potential arcs, 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.

2.3.2 Elementary definitions

1. A graph is simple if it has no loops.
2. For two graphs $G = (V, E)$ and $G' = (V', E')$, we have $G \subseteq G'$ if $V \subseteq V'$ and $E \subseteq E'$; G is a *subgraph* of G'.

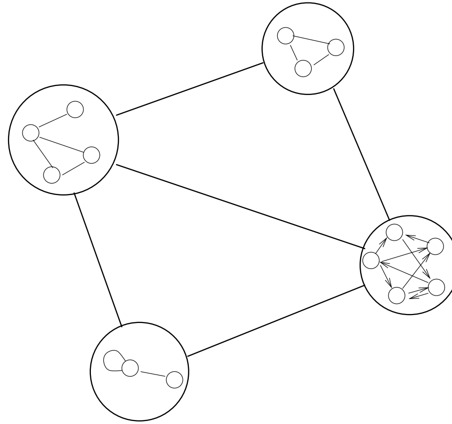


Figure 2.2: A hierarchical software architecture.

3. For a simple undirected graph $G = (V, E)$ and $U \subseteq V$ let $E[U] = \{\{u, v\} \in E \mid u, v \in U\}$; $G[U] = (U, E[U])$ is the subgraph of G *induced* by U.
4. For $v \in V$ let $\delta(v) = \{u \in V \mid \{u, v\} \in E\}$ be the *star* around v and $\bar{\delta}(v) = \{\{u, v\} \mid \{u, v\} \in E\}$ be the *cut* of v.
5. The *degree* of a vertex $v \in V$ is given by$|\delta(v)|$.
6. Stars and cuts can be extended to sets of vertices: for $U \subseteq V$ let $\delta(U) = \{v \in V \mid \exists u \in U\ (\{u, v\} \in E)$ and $\bar{\delta}(U) = \{\{u, v\} \in E \mid u \in U \wedge v \in V \smallsetminus U\}$.
7. A non-empty *path* p in G is a subgraph of G with exactly two vertices of degree 1, all other vertices having degree 2.
8. The two vertices of a path with degree one are its *endpoints*.
9. The *length* of a path is the cardinality of its edge set.
10. A graph G is *connected* if any distinct pair of its vertices is the set of endpoints of a path in G.
11. A non-empty *cycle* in G is a subgraph where each vertex has degree two.
12. Given a set V, a *clique* on V is a graph whose vertex set is V and whose edge set is $\{\{u, v\} \mid u \neq v \in V\}$.
13. A graph $G = (V, E)$ is *bipartite* if V can be partitioned in U, U' such that $U \cup U' = V$, $U \cap U' = \emptyset$, and $E[U] = E[U'] = \emptyset$.

We only consider connected graphs, as disconnected graphs can be handled by treating each connected component separately.

2.3.3 Graph operations

If architectures are represented as graphs, changing an architecture corresponds to a modification of the graph. For a graph G, we denote its vertex set by $\mathcal{V}(G)$ and its edge set by $\mathcal{E}(G)$.

Architecture analysis essentially requires two inverse operators which we might call *zoom in* and *zoom out*, whose technical names are actually *abstraction* and *concretization*.

2.3.4 Edge deletion

Since vertices represent modules and edges represent a relation such as data communication, an architecture can be simplified by identifying cycles in graphs and deleting one of their edges. The data travelling on the missing edge $\{u, v\}$ can be re-routed on the path whose endpoints are $\{u, v\}$.

2.3.5 Closure

Given a simple undirected graph $G = (V, E)$ its *k-closure* consists in adding to E all edges $\{u, v\}$ such that there is a path of length at most k with endpoints u, v. Fig. 2.3 shows the 2-closure of a graph on 3 vertices. Denote by $\mathcal{C}^k(G)$ the k-closure of G. Then for all $2 \leq h \leq k \leq |V|$ we have $\mathcal{C}^h(G) \subseteq \mathcal{C}^k(G)$.

Figure 2.3: A 2-closure.

$\mathcal{C}^{|V|}(G)$ is the *transitive closure* of G. Closure operators are inverse of edge deletion.

2.3.6 Contraction

Given an undirected graph $G = (V, E)$, it is sometimes useful to look at it "modulo" one of its subgraphs $H \subseteq G$. This can be done by *contracting* H. The contraction operator results in a *minor* G' of G where $\mathcal{V}(G') = \mathcal{V}(G) \smallsetminus \mathcal{V}(H) \cup \{v_H\}$ and $\mathcal{E}(G')$ is like $\mathcal{E}(G)$ with $\bar{\delta}(\mathcal{V}(H))$ replaced by a cut $\bar{\delta}(v_H)$. More precisely, if $\{u, v\} \in \bar{\delta}(\mathcal{V}(H))$ with $v \in \mathcal{V}(H)$, then $\{u, v\}$ is replaced by $\{u, v_H\}$. See the example in Fig. 2.4.

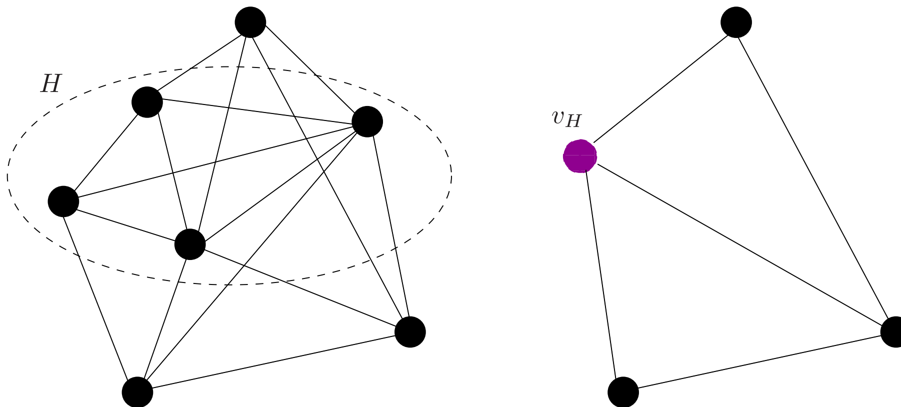


Figure 2.4: The subgraph H is contracted to v_H, leaving the minor on the right.

Contraction is an implementation of abstraction in the graph model of software system architectures.

2.3.7 Non-complete extended p-sum

The non-complete extended p-sum (NEPS) plays the role of inverse of the contraction operator, and therefore provides an implementation of concretization.

Given a set $B \subseteq \{0,1\}^p$ not including $(0,\ldots,0)$ and graphs $G_1,\ldots,G_p$, their NEPS is the graph G whose vertex set is $\mathcal{V}(G_1)\times\ldots\times\mathcal{V}(G_p)$; given two vertices $u=(u_1,\ldots,u_p)$ and $v=(v_1\ldots,v_p)$ in $\mathcal{V}(G)$ (with $u_i,v_i\in\mathcal{V}(G_i)$ for all $i\le p$), the edge $\{u,v\}$ is in $\mathcal{E}(G)$ if $\exists b=(b_1,\ldots,b_p)\in B$ such that:

- $\forall i \le p\ (b_i = 0 \to u_i = v_i)$
- $\forall i \le p\ (b_i = 1 \to \{u_i,v_i\}\in\mathcal{E}(G_i))$.

The example in Fig. 2.5 shows a contraction operation resulting in the graph top right, and an inverse NEPS between the typical contracted triangle and the minor, with $B=\{(0,1),(1,0)\}$.

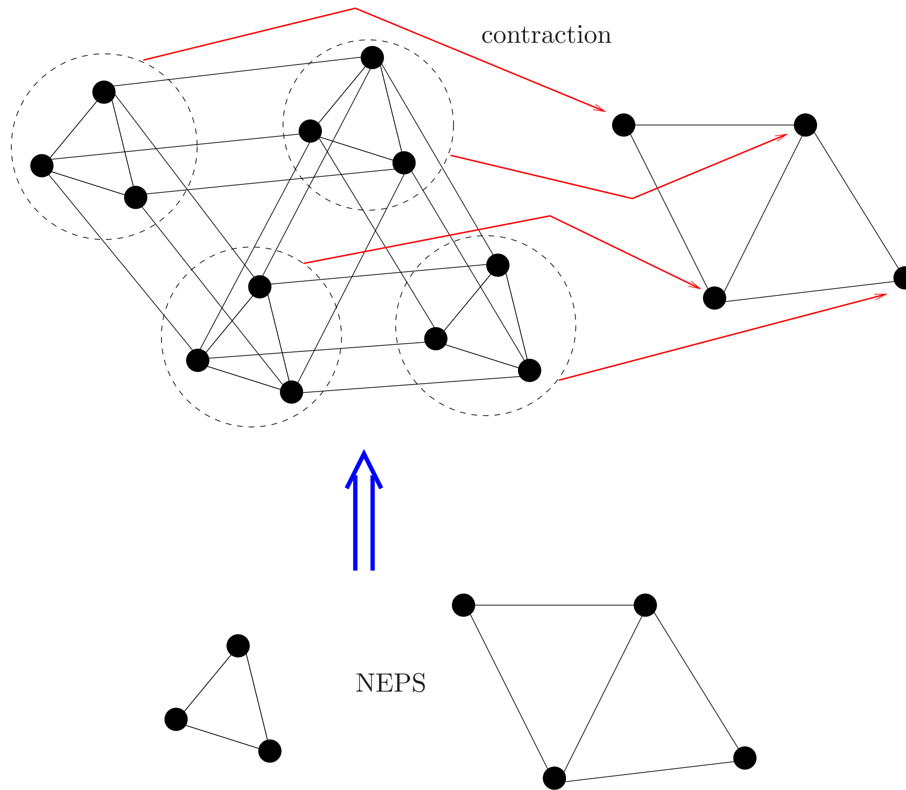


Figure 2.5: A contraction operation with its NEPS inverse.

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. 2.6.

2.4.1.1 Solution

In the given architecture, each software module has a binary relation with every other module. This is wasteful in terms of relations. It is generally better to introduce one *interface* module which relates to all

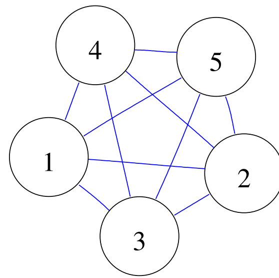


Figure 2.6: Exercise 2.4.1.

other modules, as in Fig. 2.7. Communication between modules is not direct (it is a 2-edge path), but this is generally acceptable. The original architecture has complexity 15, whereas the improved architecture

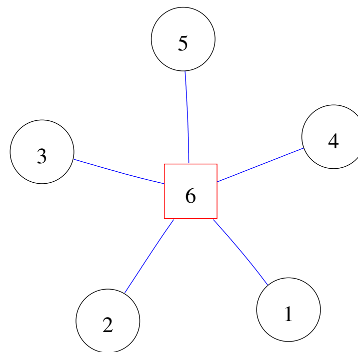


Figure 2.7: Solution to exercise 2.4.1.

has complexity 11. Is there any simpler solution?

2.4.2 Architecture complexity

Simplify the architecture given in Fig. 2.8. 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.

2.4.2.1 Solution

In the architecture in Fig. 2.8 there are 15 software modules (this includes entry, exit and split modules drawn in black), 11 black arcs and 24 blue arcs, totalling 158. By introducing the interface module marked "DBI" in Fig. 2.9 we increase the software modules to 16 and decrease the blue arcs to 14, for a new total of 133.

2.4.3 Complexity comparison

What is the most complex architecture among the ones given in Fig. 2.10? How can they be simplified?

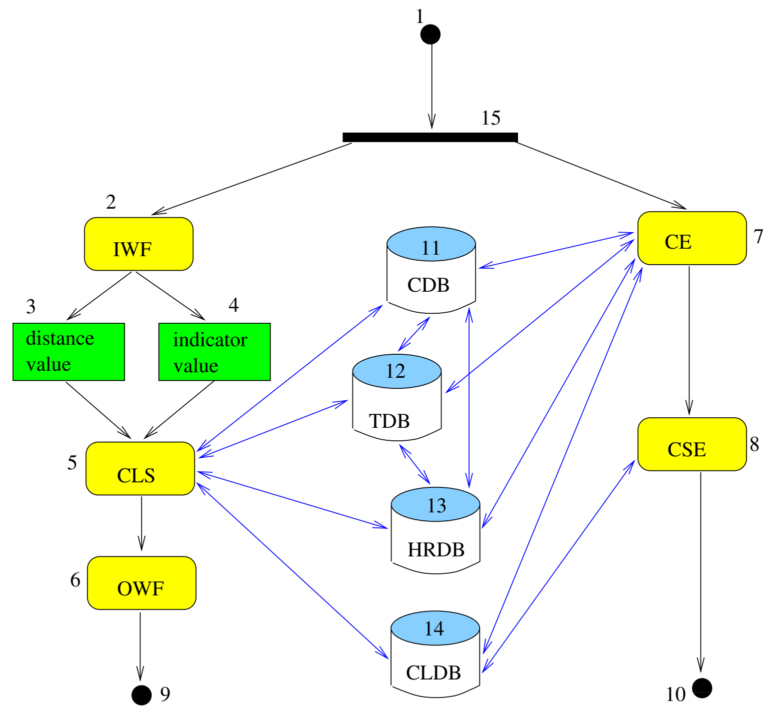


Figure 2.8: Exercise 2.4.2.

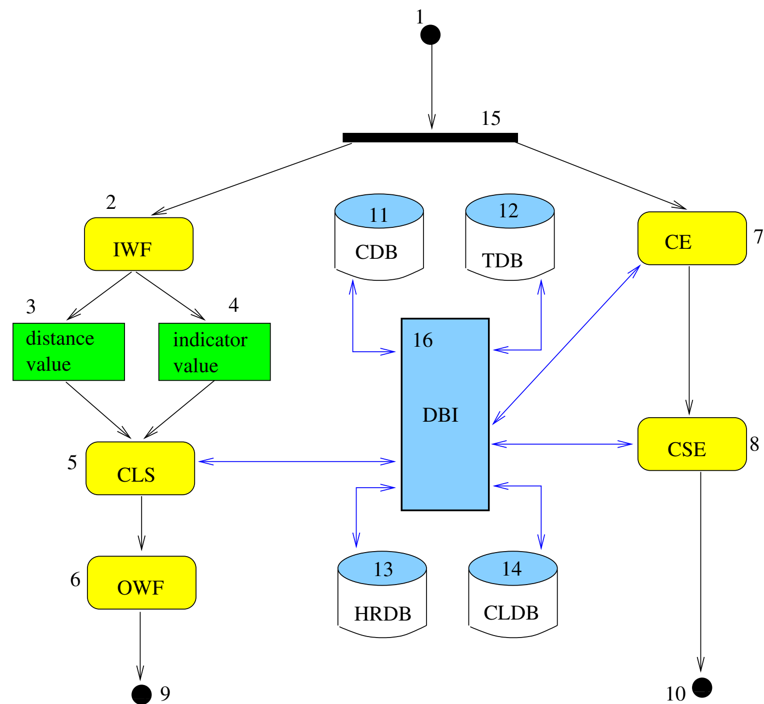


Figure 2.9: Solution to exercise 2.4.2.

2.4.3.1 Solution

The above pictures all represent the same graph.

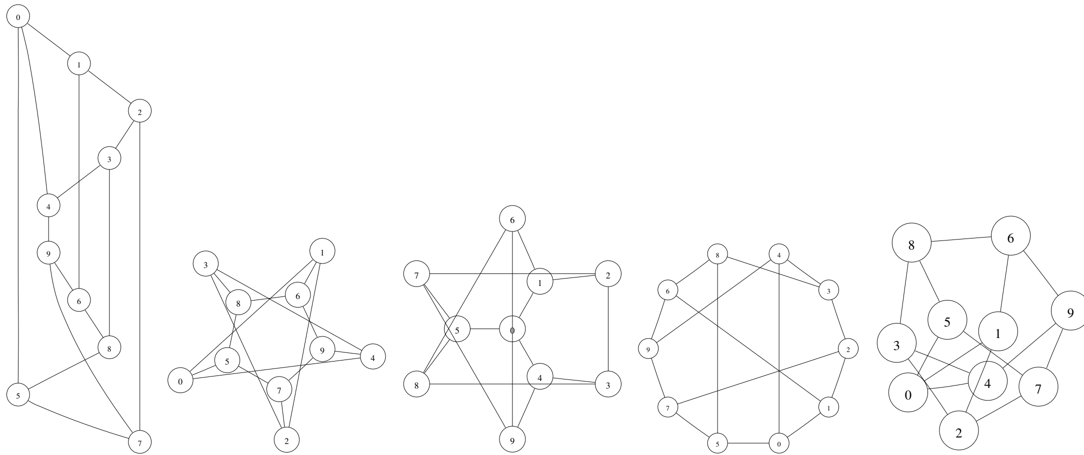


Figure 2.10: Exercise 2.4.3.

2.4.4 NEPS

1. What are the NEPS of a cycle of length 3 and a cycle of length 4 for all $B \subseteq \{0,1\}^2 \smallsetminus \{(0,0)\}$?
2. Can you find the NEPS operation inverse to the contraction operation shown in Sect. 2.4?

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.

3.1.1.1 Solution

The use case diagram is given in Fig. 3.1 (taken from [7], Fig. 16.5).

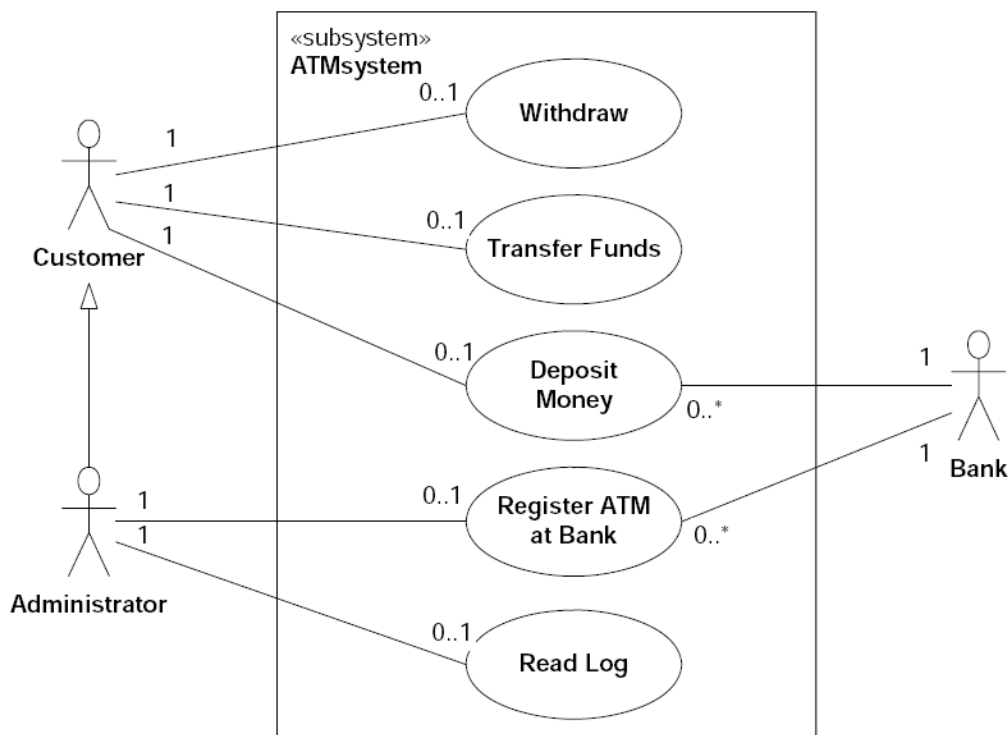


Figure 3.1: The simplified use case diagram of an ATM.

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.

3.1.2.1 Solution

The use case diagram is given in Fig. 3.2. The symbols at each end of relation edges are called *multiplicities*, and they indicate how many times a given action occurs linked to its actor(s), and how many times a

given actor carries out the corresponding actions; a multiplicity $n..*$ next to an action indicates that the actor can carry it out a minimum of n times and a maximum of ∞ times. Boxes carrying a binary logical operator label (and, or, xor) help inserting some logical meaining into a use case diagram. A relation (A, B) carrying a triangle-shaped arrowhead next to B stands for *generalization*: object (actor or action) B generalises object A, which means that A has all the properties of B and some more (thus, an "angry client" is a client with one more property, that of being angry — an angry client can thus carry out all the actions that can be carried out by any client, plus some more relative to his/her added properties). A dashed action-action relation (A, B) carrying a label "include" means that action B necessarily takes place within action A. For example, confirming that we want to buy a good necessarily includes retrieving that good — although one might imagine a case of somebody using the machine in a train station and abandoning the good (already paid for) because the train is about to leave, the "train" element is evidently completely removed from the "vending machine" system, and so need not be modelled. A dashed action-action relation (A, B) carrying the label "extends" means that action A may take place within action B (notice that this relation is reversed with respect to the "include" relation). For example, once we confirm we want to buy some good, we necessarily retrieve the good but there may be no change to collect.

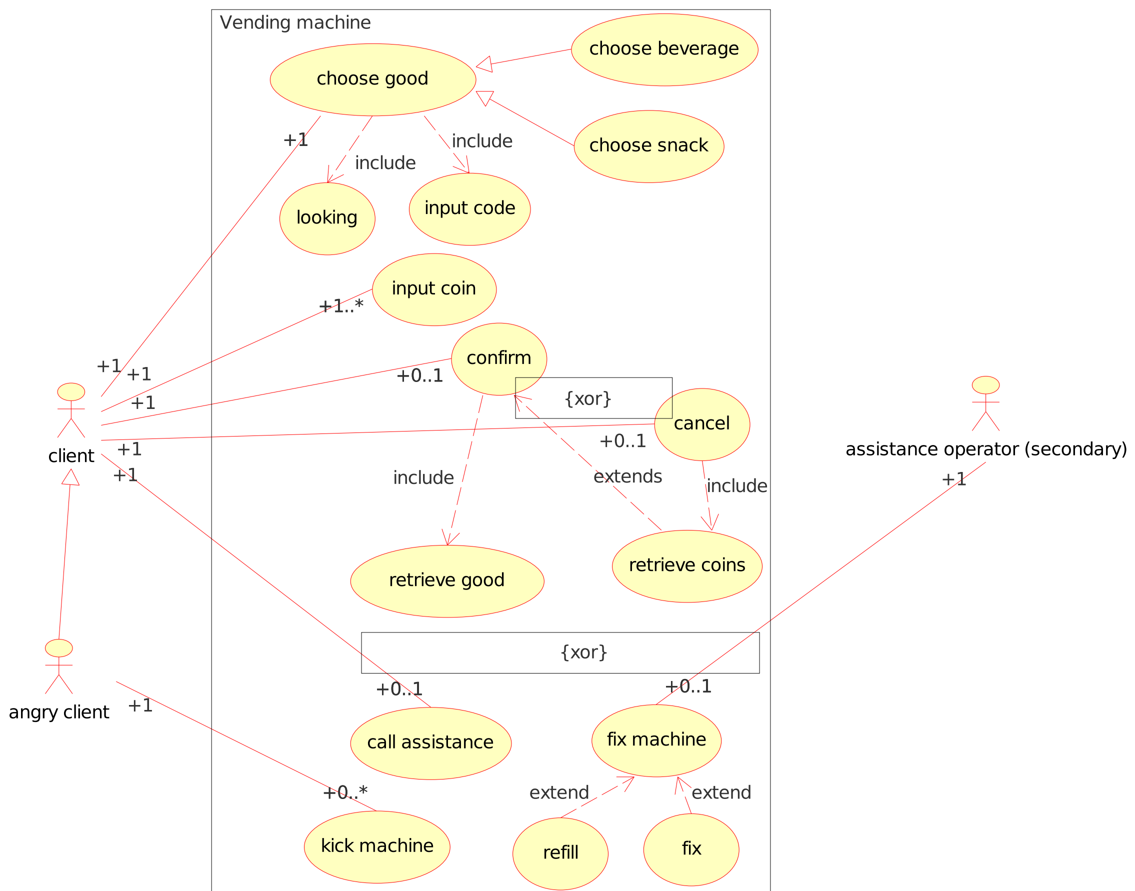


Figure 3.2: The use case diagram of a vending machine.

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.

3.2.1.1 Solution

In order to draw a sequence diagram we must first draw a use case diagram to help us identify system, actors and actions. In this case, drawing a use case diagram is not all that simple: we have a static piece of code. Where is the system? Who are the actors? It turns out that the code has a class and a function outside the class. In this setting, the class is passive. If no entity calls the class methods, the class will not carry out any action of its own accord. The class is therefore to be considered as a system rather than an actor. The `norm()` function, on the other hand, calls methods within the class, and can thus be likened to an actor. The actions are precisely those methods that are called by `norm()`. This arrangement is shown in Fig. 3.3

Having identified actors, actions and system, we can now build the sequence diagram. The `norm()` function calls two `Array` methods: `size()` and `get()`, both of which block `norm()` until a return value

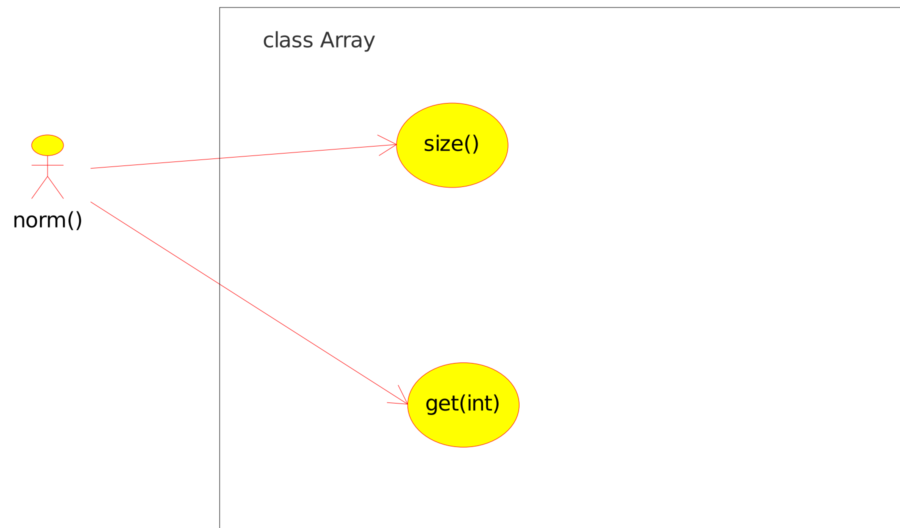


Figure 3.3: The use case diagram of the `norm()/Array` system.

is received (and can therefore be represented by synchronous messages). One other notable non-local action carried out by `norm()` is the call to the `sqrt()` method. This is external to `norm()` so it should be represented on the sequence diagram; on the other hand, we have no system component where `sqrt()` belongs — in practice, the function `sqrt()` is found in a system library called `libm` on UNIX systems, but this is hardly part of the system at hand so it has no representation. In such cases, we depict the `sqrt()` call as a *self-action* of the `norm()` actor onto itself. This is shown in Fig. 3.4.

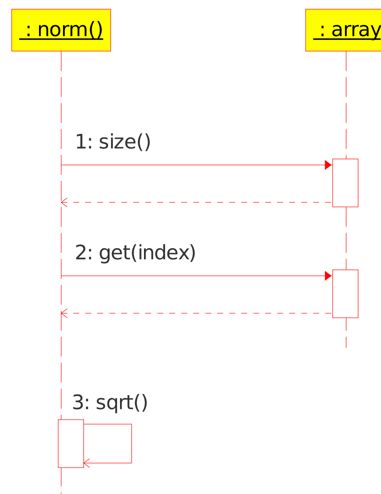


Figure 3.4: The sequence diagram describing the computation of the norm of a vector.

3.2.2 Displaying graphical objects

Write a sequence diagram for a program that displays Fig. 3.5 on the screen in the order left $\rightarrow$ right.

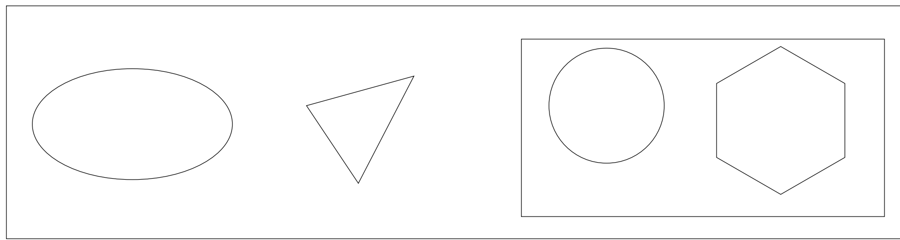


Figure 3.5:

3.2.2.1 Solution

The sequence diagram in Fig. 3.6 describes the required behaviour.

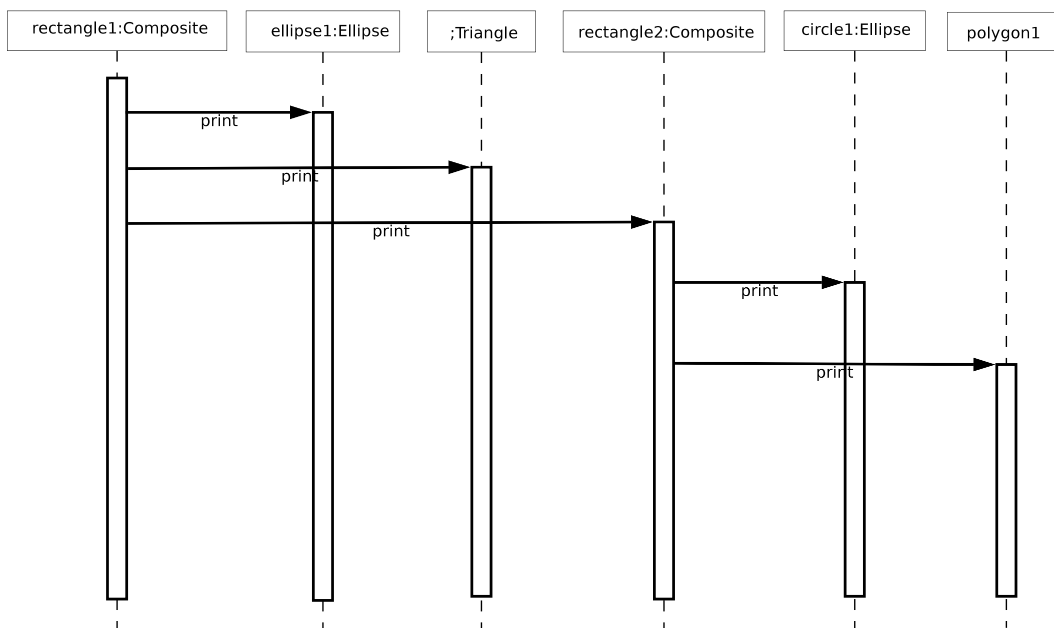


Figure 3.6: The sequence diagram describing the printing of the drawing in Fig. 3.5.

3.2.3 Vending machine

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

3.2.3.1 Solution

The sequence diagram in Fig. 3.7 describes the required behaviour. Notice the box-shaped elements in the diagram: these are used to endow the diagram with a modicum of logical representation: `alt` signals an *alternative* (an `if`...`else` block), `opt` signals an *option* (an `if` block) and `loop` a *loop* (a `for` or `while` block).

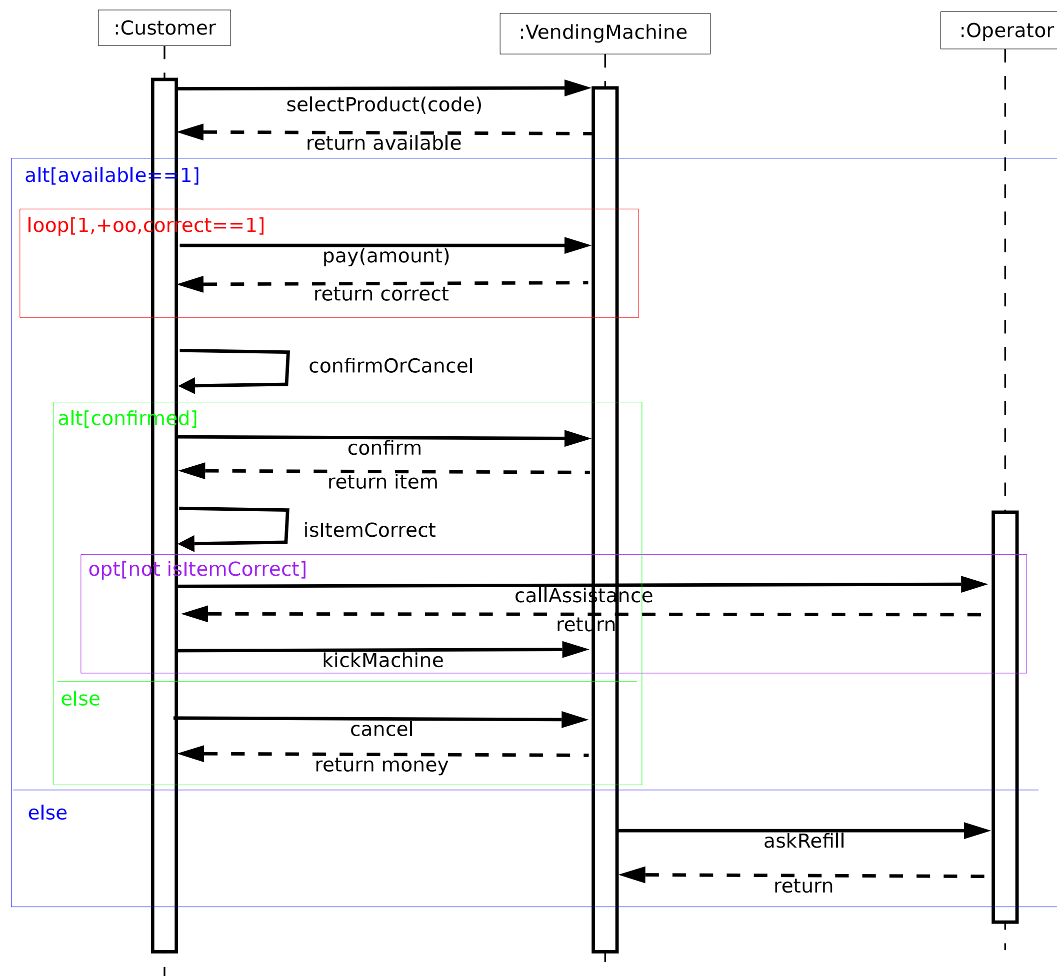


Figure 3.7: The sequence diagram for the vending machine of Sect. 3.1.2.

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).

3.2.4.1 Solution

The sequence diagram in Fig. 3.8 illustrates the interactions between boy and girl. Some other actors and system components have been added, mostly for fun. Boxes labelled `par` denote parallel actions.

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

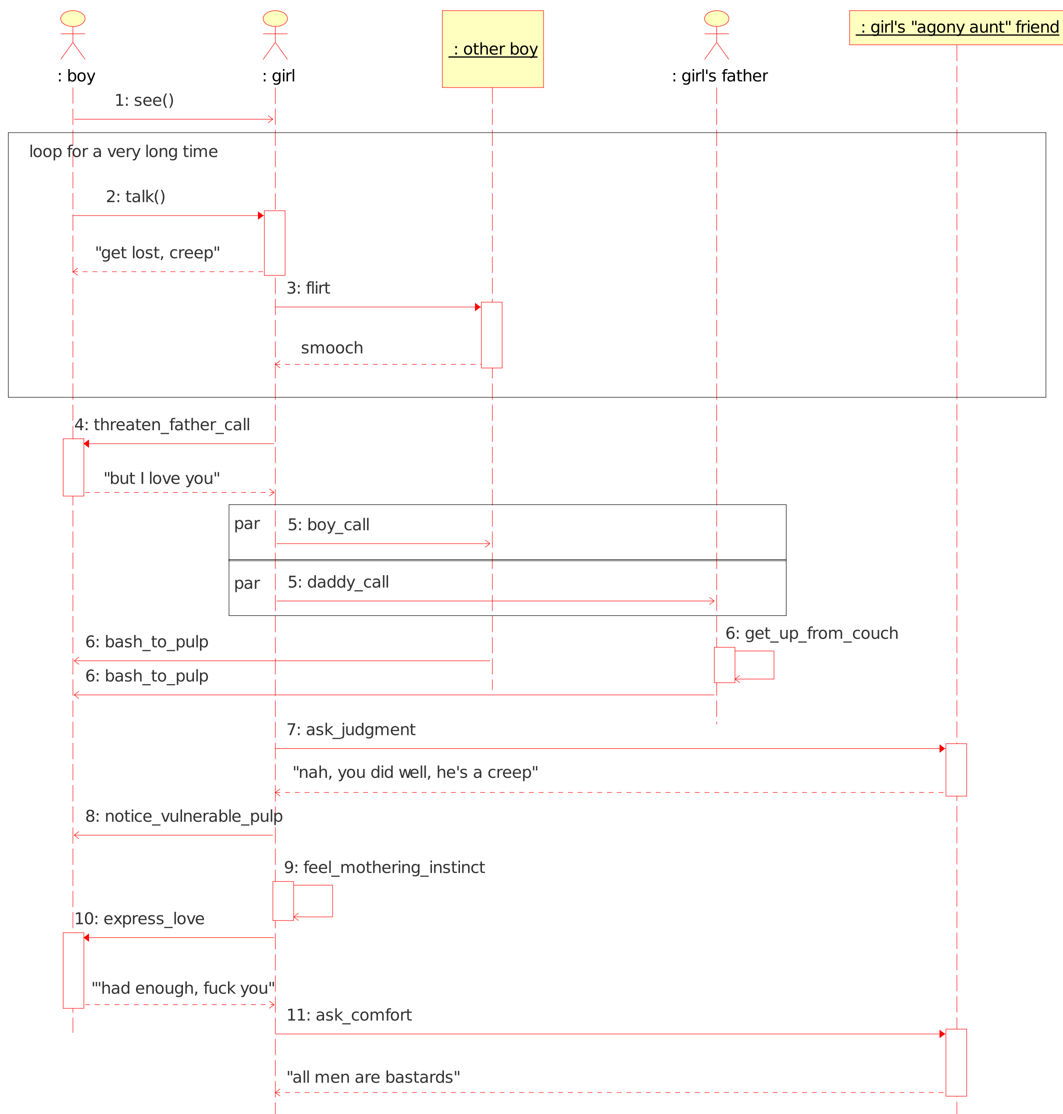


Figure 3.8: The sequence diagram for the boy/girl interactions.

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.

3.3.1.1 Solution

The state diagram is given in Fig. 3.9, and the activity diagram in Fig. 3.10.

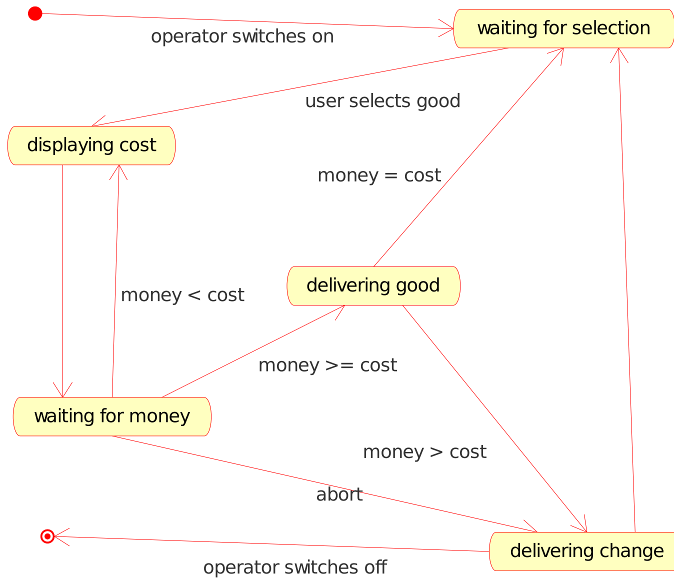


Figure 3.9: The state diagram for the vending machine.

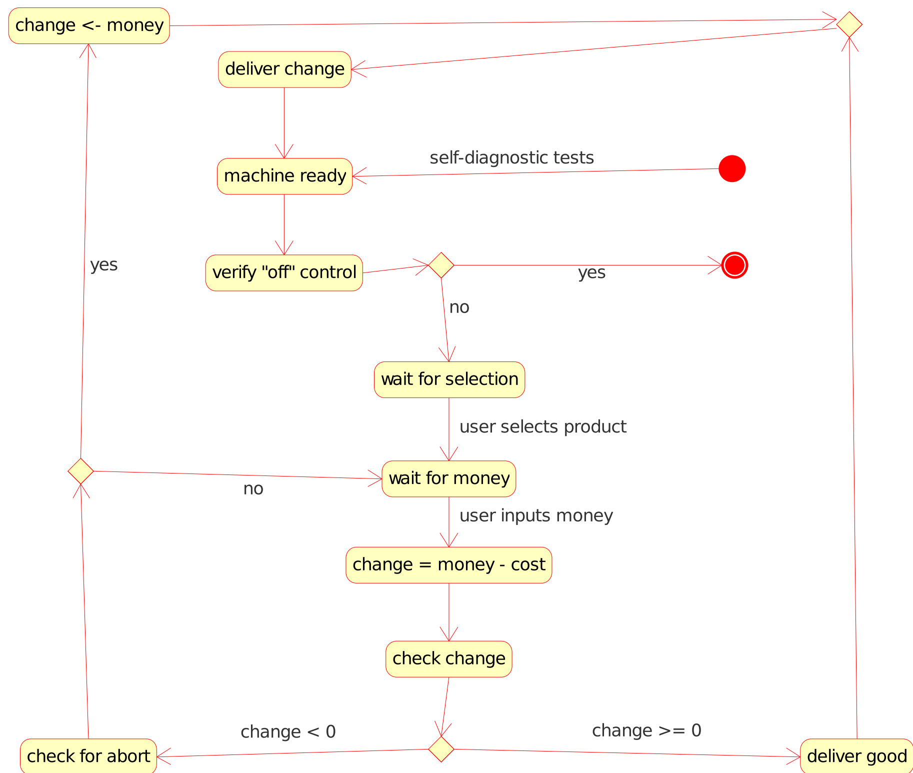


Figure 3.10: The activity diagram for the vending machine.

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.

3.4.1.1 Solution

The class diagram for the `Complex` class is given in Fig. 3.11.

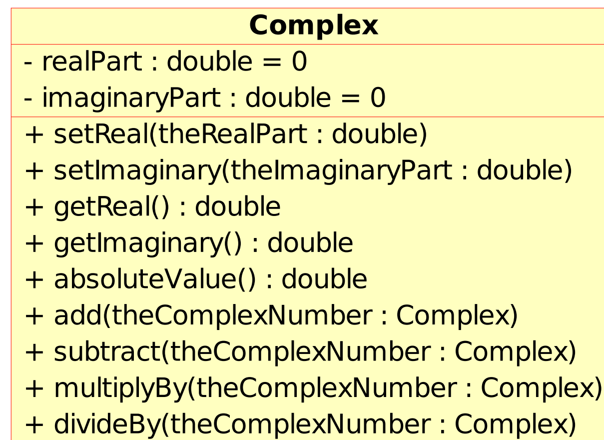


Figure 3.11: The class diagram for a complex number.

Most UML modellers can be used to automatically generate C++ (or Java) code from the class diagram. This results in "class skeleton" files (a header file `Complex.h` and a corresponding implementation file `Complex.cpp`). Both are given below.

```
/***************************************************************************
                        Complex.h.h - Copyright liberti

Here you can write a license for your code, some comments or any other
information you want to have in your generated code. To to this simply
configure the "headings" directory in uml to point to a directory
where you have your heading files.

or you can just replace the contents of this file with your own.
If you want to do this, this file is located at

/usr/share/apps/umbrello/headings/heading.h

-->Code Generators searches for heading files based on the file extension
   i.e. it will look for a file name ending in ".h" to include in C++ header
   files, and for a file name ending in ".java" to include in all generated
   java code.
   If you name the file "heading.<extension>", Code Generator will always
   choose this file even if there are other files with the same extension in the
   directory. If you name the file something else, it must be the only one with that
   extension in the directory to guarantee that Code Generator will choose it.

you can use variables in your heading files which are replaced at generation
```

```
time. possible variables are : author, date, time, filename and filepath.
just write %variable_name%

This file was generated on %date% at %time%
**************************************************************************/


#ifndef COMPLEX_H
#define COMPLEX_H

#include <string>

/**
  * class Complex
  */

class Complex
{
public:

  // Constructors/Destructors
  //


  /**
   * Empty Constructor
   */
  Complex ( );

  /**
   * Empty Destructor
   */
  virtual ~Complex ( );

  // Static public attributes
  //

  // public attributes
  //


  // public attribute accessor methods
  //


  // public attribute accessor methods
  //



  /**
   * @param  theRealPart
   */
  void setReal (double theRealPart );


  /**
   * @param  theImaginaryPart
   */
  void setImaginary (double theImaginaryPart );


  /**
   * @return double
   */
  double getReal ( );


  /**
   * @return double
   */
  double getImaginary ( );


  /**
   * @return double
   */
  double absoluteValue ( );


  /**
```

```
   * @param  theComplexNumber
   */
  void add (Complex theComplexNumber );


  /**
   * @param  theComplexNumber
   */
  void subtract (Complex theComplexNumber );


  /**
   * @param  theComplexNumber
   */
  void multiplyBy (Complex theComplexNumber );


  /**
   * @param  theComplexNumber
   */
  void divideBy (Complex theComplexNumber );

protected:

  // Static protected attributes
  //

  // protected attributes
  //

public:


  // protected attribute accessor methods
  //

protected:

public:


  // protected attribute accessor methods
  //

protected:


private:

  // Static private attributes
  //

  // private attributes
  //

  double m_realPart;
  double m_imaginaryPart;
public:


  // private attribute accessor methods
  //

private:

public:


  // private attribute accessor methods
  //


  /**
   * Set the value of m_realPart
   * @param new_var the new value of m_realPart
   */
  void setRealPart ( double new_var );

  /**
   * Get the value of m_realPart
   * @return the value of m_realPart
```

```
   */
  double getRealPart ( );


  /**
   * Set the value of m_imaginaryPart
   * @param new_var the new value of m_imaginaryPart
   */
  void setImaginaryPart ( double new_var );

  /**
   * Get the value of m_imaginaryPart
   * @return the value of m_imaginaryPart
   */
  double getImaginaryPart ( );

private:


  void initAttributes ( ) ;

};

#endif // COMPLEX_H



/************************************************************************
                        Complex.h.cpp - Copyright liberti

Here you can write a license for your code, some comments or any other
information you want to have in your generated code. To to this simply
configure the "headings" directory in uml to point to a directory
where you have your heading files.

or you can just replace the contents of this file with your own.
If you want to do this, this file is located at

/usr/share/apps/umbrello/headings/heading.cpp

-->Code Generators searches for heading files based on the file extension
   i.e. it will look for a file name ending in ".h" to include in C++ header
   files, and for a file name ending in ".java" to include in all generated
   java code.
   If you name the file "heading.<extension>", Code Generator will always
   choose this file even if there are other files with the same extension in the
   directory. If you name the file something else, it must be the only one with that
   extension in the directory to guarantee that Code Generator will choose it.

you can use variables in your heading files which are replaced at generation
time. possible variables are : author, date, time, filename and filepath.
just write %variable_name%

This file was generated on %date% at %time%
**************************************************************************/

#include "Complex.h"

// Constructors/Destructors
//

Complex::Complex ( ) {
initAttributes();
}

Complex::~Complex ( ) { }

//
// Methods
//


// Accessor methods
//


// public static attribute accessor methods
//


// public attribute accessor methods
//
```

```
// protected static attribute accessor methods
//


// protected attribute accessor methods
//


// private static attribute accessor methods
//


// private attribute accessor methods
//


/**
 * Set the value of m_realPart
 * @param new_var the new value of m_realPart
 */
void Complex::setRealPart ( double new_var ) {
  m_realPart = new_var;
}

/**
 * Get the value of m_realPart
 * @return the value of m_realPart
 */
double Complex::getRealPart ( ) {
  return m_realPart;
}

/**
 * Set the value of m_imaginaryPart
 * @param new_var the new value of m_imaginaryPart
 */
void Complex::setImaginaryPart ( double new_var ) {
  m_imaginaryPart = new_var;
}

/**
 * Get the value of m_imaginaryPart
 * @return the value of m_imaginaryPart
 */
double Complex::getImaginaryPart ( ) {
  return m_imaginaryPart;
}

// Other methods
//


/**
 * @param  theRealPart
 */
void Complex::setReal (double theRealPart ) {

}


/**
 * @param  theImaginaryPart
 */
void Complex::setImaginary (double theImaginaryPart ) {

}


/**
 * @return double
 */
double Complex::getReal ( ) {

}


/**
 * @return double
 */
```

```
double Complex::getImaginary ( ) {

}


/**
 * @return double
 */
double Complex::absoluteValue ( ) {

}


/**
 * @param  theComplexNumber
 */
void Complex::add (Complex theComplexNumber ) {

}


/**
 * @param  theComplexNumber
 */
void Complex::subtract (Complex theComplexNumber ) {

}


/**
 * @param  theComplexNumber
 */
void Complex::multiplyBy (Complex theComplexNumber ) {

}


/**
 * @param  theComplexNumber
 */
void Complex::divideBy (Complex theComplexNumber ) {

}

void Complex::initAttributes ( ) {
  m_realPart = 0;
  m_imaginaryPart = 0;
}
```

3.4.2 Singly linked list

Draw a class diagram representing a singly linked list.

3.4.2.1 Solution

The class diagram is given in Fig. 3.12. It consists of a class with a single unidirectional association (`next`) with multiplicity 1, because a node of a singly linked list only has one neighbouring node (the next node).

3.4.3 Doubly linked list

Draw a class diagram representing a doubly linked list.

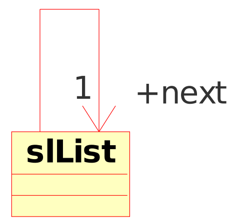


Figure 3.12: The class diagram of a singly linked list.

3.4.3.1 Solution

The class diagram is given in Fig. 3.13. It consists of a class with a single bidirectional association with reference names `previous` and `next` both with multiplicity 1, because a doubly linked list has a previous and a next node.

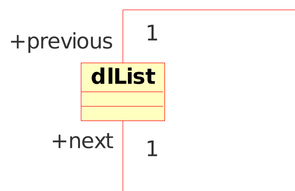


Figure 3.13: The class diagram of a doubly linked list.

3.4.4 Binary tree

Draw a class diagram representing a binary tree.

3.4.4.1 Solution

The class diagram is given in Fig. 3.14. It consists of a class with a single bidirectional association with reference names `child` (with multiplicity 2) and `parent` (with multiplicity 1), because a binary tree has two children and one parent node.

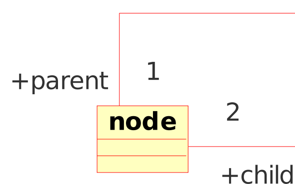


Figure 3.14: The class diagram of a binary tree.

3.4.5 n-ary tree

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

3.4.5.1 Solution

The class diagram is given in Fig. 3.15. It consists of a class with a single bidirectional association with reference names `child` (with multiplicity $*$) and `parent` (with multiplicity 1), because a binary tree has a variable number of children and one parent node.

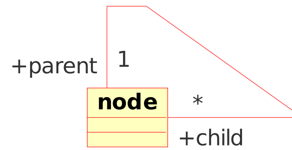


Figure 3.15: The class diagram of an n-ary tree.

3.4.6 Vending machine

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

3.4.6.1 Solution

The class diagram is given in Fig. 3.16.

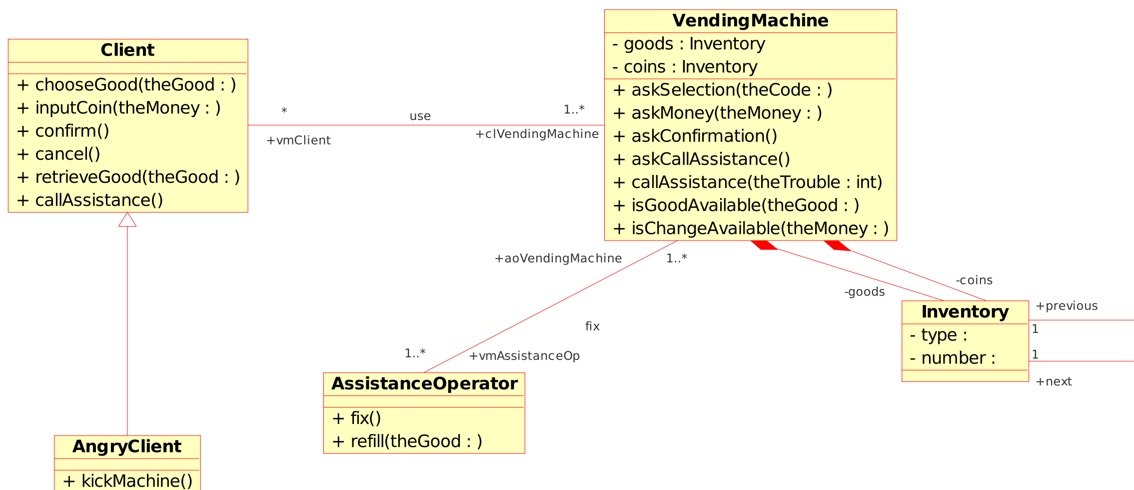


Figure 3.16: The class diagram of a vending machine.

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.

4.1.1 Solution

The use case diagram is found in Fig. 4.1. The sequence diagram is found in Fig. 4.2. Notice that they do not provide mechanisms for calling assistance operators on failure of providing change and/or food. How should these diagrams change to cater for these occurrences?

4.2 Mistakes in modelling a tree

Fig. 4.3 describes the class diagram of a tree node, which can be used recursively to build an expression tree.

Figure 4.1: The revised use case diagram of the vending machine.

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) {
  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;
```

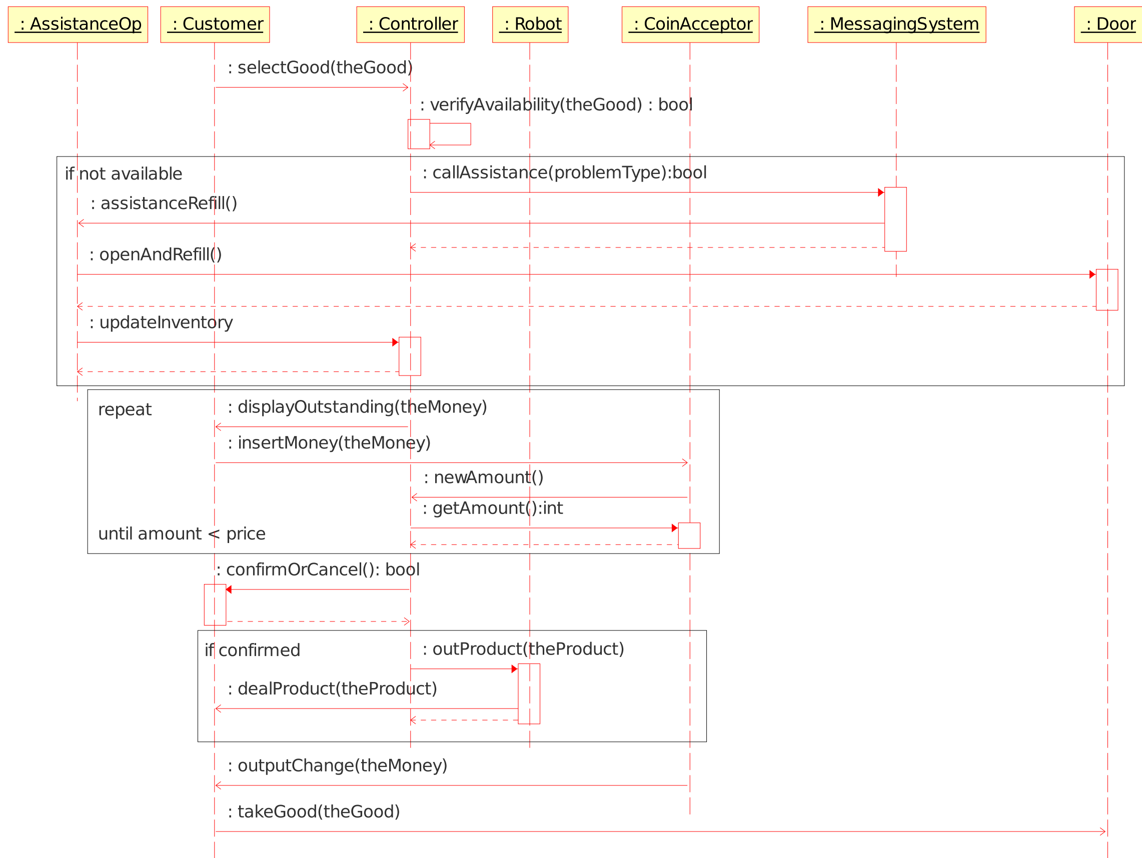


Figure 4.2: The revised sequence diagram of a vending machine.

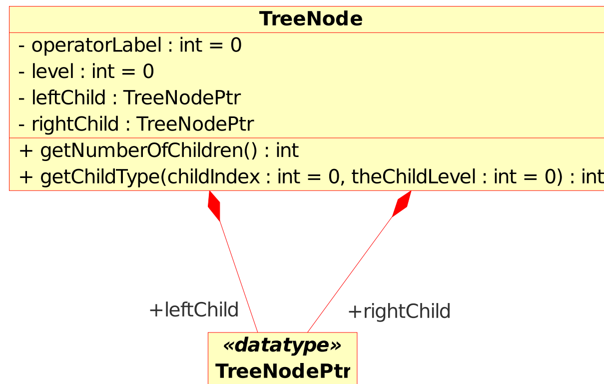


Figure 4.3: The UML class diagram for the `TreeNode` class.

```
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.

4.2.1 Solution

The output is `4, 0`. The problem is given by the fact that the second argument of `getChildType`, that is, `theLevel`, was not declared as an *inout* (read/write) parameter but as an *in* (read only) parameter instead, so it cannot be changed by the function itself (notice the compiler issues no warning about this occurrence: it is perfectly legal syntactically if not semantically).

The new function cannot be coded in because the model provide no mechanism for going from a given node to its parent node, much less the root node of the tree. The correct UML class diagram is given in Fig. 4.4.

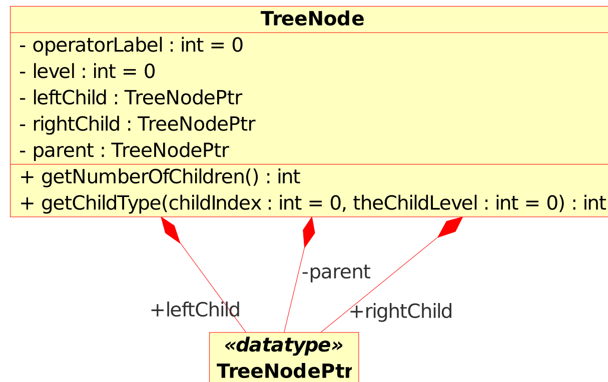


Figure 4.4: The corrected UML class diagram for the `TreeNode` class.

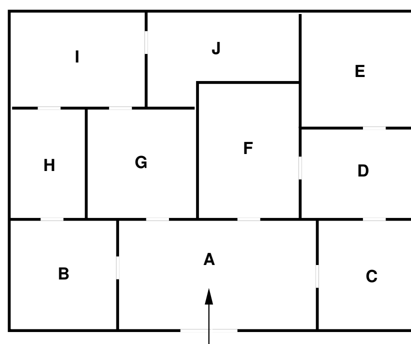
Chapter 5

Mathematical programming exercises

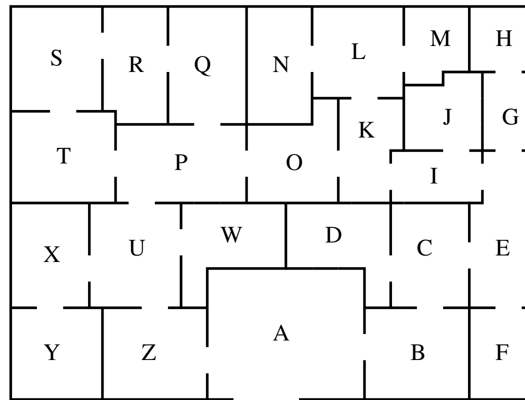
The mathematical programming formulation language is a very powerful tool used to formalize optimization problems by means of parameters, decision variables, objective functions and constraints. Such diverse settings as combinatorial, integer, continuous, linear and nonlinear optimization problems can 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.



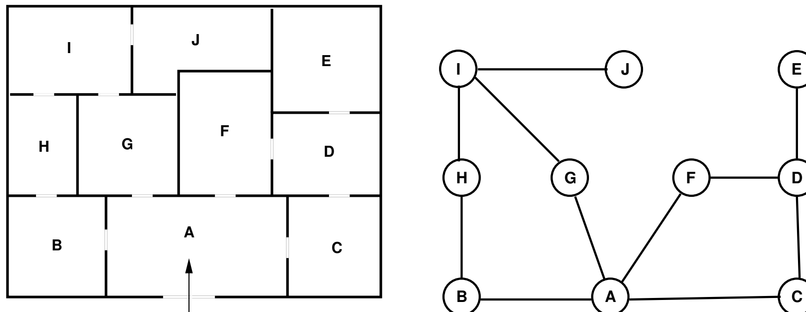
Also solve the problem on the following map.



[*P. Belotti, Carnegie Mellon University*]

5.1.1 Solution

The problem can be formalized by representing each museum room by a vertex $v \in V$ of an undirected graph $G = (V, E)$. There is an edge between two vertices if there is a door leading from one room to the other; this way, edges represent the possibility of there being a guard on a door. We want to choose the smallest subset $F \subseteq E$ of edges *covering* all vertices, i.e. such that for all $v \in V$ there is $w \in V$ with $\{v, w\} \in F$.



To each $\{i, j\} \in E$ we associated a binary variable x_{ij} is assigned the value 1 if there is a guard on the door represented by edge $\{i, j\}$ and 0 otherwise.

5.1.1.1 Formulation

- *Parameters.* $G = (V, A)$: graph description of the museum topology.
- *Variables.* x_{ij}: 1 if edge $\{i, j\} \in E$ is to be included in F, 0 otherwise.
- *Objective function*

$$\min \sum_{\{i,j\} \in E} x_{ij}$$

- *Constraints.* (Vertex cover): $\sum_{j \in V : \{i,j\} \in E} x_{ij} \geq 1 \quad \forall i \in V$.

5.1.1.2 AMPL model, data, run

```
# museum.mod

param n >= 0, integer;
set V := 1..n;
set E within {V,V};
var x{E} binary;
minimize cost : sum{(i,j) in E} x[i,j];
subject to vertexcover {i in V} :
  sum{j in V : (i,j) in E} x[i,j] + sum{j in V : (j,i) in E} x[j,i] >= 1;
```

```
# museum.dat

param n := 10;
set E :=
  1 2
  1 3
  1 6
  1 7
  2 8
  3 4
  4 5
  7 9
  8 9
  9 10 ;
```

```
# museum.run

model museum.mod;
data museum.dat;
option solver cplexstudent;
solve;
display cost;
display x;
```

5.1.1.3 CPLEX solution

```
CPLEX 7.1.0: optimal integer solution; objective 6
2 MIP simplex iterations
0 branch-and-bound nodes
cost = 6

x :=
1 2    0
1 3    1
1 6    1
1 7    1
2 8    1
3 4    0
4 5    1
7 9    0
8 9    0
9 10   1
;
```

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*]

5.2.1 Solution

5.2.1.1 Formulation

- *Indicex*: Let i be an index on the set $\{1, 2, 3\}$.
- *Parameters*:
  - P: number of production days in a month;
  - d_i: maximum market demand for product i;
  - v_i: selling price for product i;
  - c_i: production cost for product i;
  - q_i: maximum production quota for product i;
  - a_i: activation cost for the plant producing i;
  - l_i: minimum batch of product i.
- *Variables*:
  - x_i: quantity of product i to produce ($x_i \geq 0$);
  - y_i: activation status of product i (1 if active, 0 otherwise).

- *Objective function*:
$$\max \sum_i ((v_i - c_i)x_i - a_i y_i)$$

- *Constraints*:
  1. (demand): for each i, $x_i \leq d_i$;
  2. (production): $\sum_i \frac{x_i}{q_i} \leq P$;
  3. (activation): for each i, $x_i \leq P q_i y_i$;
  4. (minimum batch): for each i, $x_i \geq l_i y_i$;

5.2.1.2 AMPL model, data, run

```
# mixedproduction.mod

set PRODUCTS;

param days >= 0;
param demand { PRODUCTS } >= 0;
param price { PRODUCTS } >= 0;
param cost { PRODUCTS } >= 0;
param quota { PRODUCTS } >= 0;
param activ_cost { PRODUCTS } >= 0;    # activation costs
param min_batch { PRODUCTS } >= 0;     # minimum batches

var x { PRODUCTS } >= 0;               # quantity of product
var y { PRODUCTS } >= 0, binary;       # activation of production lines

maximize revenue: sum {i in PRODUCTS}
((price[i] - cost[i]) * x[i] - activ_cost[i] * y[i]);

subject to requirement {i in PRODUCTS}:
x[i] <= demand[i];

subject to production:
sum {i in PRODUCTS} (x[i] / quota[i]) <= days;

subject to activation {i in PRODUCTS}:
x[i] <= days * quota[i] * y[i];

subject to batches {i in PRODUCTS}:
        x[i] >= min_batch[i] * y[i];


# mixedproduction.dat

set PRODUCTS := A1 A2 A3 ;

param days := 22;
param : demand  price   cost  quota  activ_cost min_batch :=
     A1 5300    124     73.30   500      170000        20
     A2 4500    109     52.90   450      150000        20
     A3 5400    115     65.40   550      100000        16 ;


# mixedproduction.run
```

```
model mixedproduction.mod;
data mixedproduction.dat;
option solver cplexstudent;
solve;
display x;
display y;
```

5.2.1.3 CPLEX solution

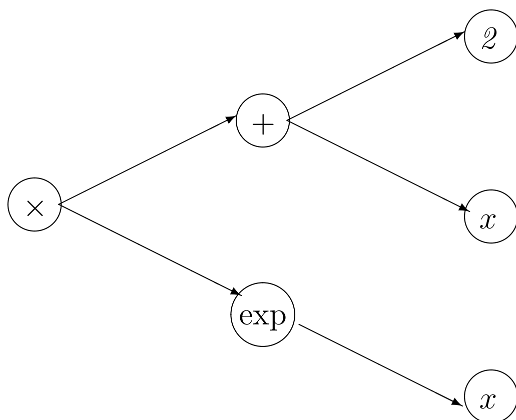
.

```
CPLEX 7.1.0: optimal integer solution; objective 220690
5 MIP simplex iterations
0 branch-and-bound nodes
ampl: display x;
x [*] :=
A1     0
A2  4500
A3  5400
;

ampl: display y;
y [*] :=
A1  0
A2  1
A3  1
;
```

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 $+ \begin{smallmatrix} \nearrow & a \\ \searrow & b \end{smallmatrix}$;
- `difference(term` a`, term` b`)`: make a tree $- \begin{smallmatrix} \nearrow & a \\ \searrow & b \end{smallmatrix}$;
- `product(term` a`, term` b`)`: make a tree $* \begin{smallmatrix} \nearrow & a \\ \searrow & b \end{smallmatrix}$;
- `fraction(term` a`, term` b`)`: make a tree $/ \begin{smallmatrix} \nearrow & a \\ \searrow & b \end{smallmatrix}$;
- `power(term` a`, term` b`)`: make a tree $\wedge \begin{smallmatrix} \nearrow & a \\ \searrow & b \end{smallmatrix}$;
- `minus(term` a`)`: make a tree $- \to a$;
- `logarithm(term` a`)`: make a tree make a tree $\log \to a$;
- `exponential(term` a`)`: make a tree make a tree $\exp \to a$;
- `sine(term` a`)`: make a tree make a tree $\sin \to a$;
- `cosine(term` a`)`: make a tree make a tree $\cos \to a$;
- `tangent(term` a`)`: make a tree make a tree $\tan \to a$;
- `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.

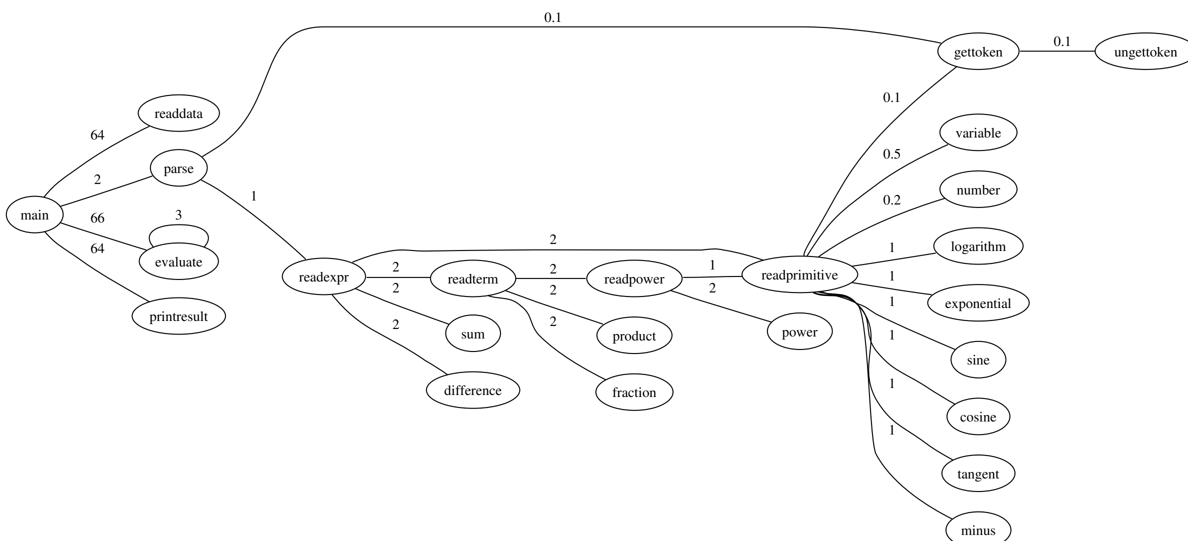
- main: readdata (64KB), parse (2KB), evaluate (66KB), printresult(64KB)
- evaluate: evaluate (3KB)
- parse: gettoken (0.1KB), readexpr (1KB)
- readprimitive: gettoken (0.1KB), variable (0.5KB), number (0.2KB), logarithm (1KB), exponential (1KB), sine (1KB), cosine (1KB), tangent (1KB), minus (1KB), 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?

5.3.1 Solution

We represent each subroutine with a vertex in an undirected graph $G = (V, E)$. For each $u, v \in V$, $\{u, v\} \in E$ if subroutine u calls subroutine v (or vice versa). Each edge $\{i, j\} \in E$ is weighted by the quantity p_{ij} of data exchanged between the subroutines. We want to choose a subset $L \subseteq E$ such that for each $u \in V$ there is $v \in V$ with $\{u, v\} \in L$ (i.e. L covers V), such that each vertex $v \in V$ is adjacent to exactly 1 edge in L and such that the total weight $p(L) = \sum_{\{i,j\} \in L} p_{ij}$ is maximum. G is shown below.



5.3.1.1 Formulation

- *Parameters*: for each $\{i,j\} \in E$, p_{ij} is the weight on the edge.
- *Variables*: for each $\{i,j\} \in E$, $x_{ij} = 1$ if $\{i,j\} \in L$ and 0 otherwise.
- *Objective function*:
$$\max \sum_{e=\{i,j\}\in E} p_{ij} x_{ij}$$
- *Constraints*:
$$\forall i \in V \qquad \sum_{j\in V\,|\,\{i,j\}\in E} x_{ij} = 1; \tag{5.1}$$
$$\forall \{i,j\} \in E \qquad x_{ij} \in \{0,1\}. \tag{5.2}$$

5.3.1.2 AMPL model, data, run

```
# checksum.mod

set V;
set E within {V,V};
param p{E};
var x{E} binary;
maximize data : sum{(i,j) in E} p[i,j] * x[i,j];
subject to assignment {i in V} :
  sum{j in V : (i,j) in E} x[i,j] + sum{j in V : (j,i) in E} x[j,i] <= 1;


# checksum.dat

set V := main readdata parse evaluate printresult gettoken readexpr
         readprimitive variable number logarithm exponential sine cosine
         tangent minus power readpower readterm product fraction sum ;
set E :=
 main   readdata
 main   parse
 main   evaluate
 main   printresult
 evaluate   evaluate
 parse   gettoken
 parse   readexpr
 readprimitive   gettoken
 readprimitive   variable
 readprimitive   number
 readprimitive   logarithm
 readprimitive   exponential
 readprimitive   sine
 readprimitive   cosine
 readprimitive   tangent
 readprimitive   minus
 readprimitive   readexpr
 readpower   power
 readpower   readprimitive
 readterm   readpower
 readterm   product
 readterm   fraction
 readexpr   readterm
```

```
 readexpr   sum   ;

param p :=
 main   readdata    64
 main   parse    2
 main   evaluate    66
 main   printresult    64
 evaluate   evaluate    3
 parse   gettoken    0.1
 parse   readexpr    1
 readprimitive   gettoken    0.1
 readprimitive   variable    0.5
 readprimitive   number    0.2
 readprimitive   logarithm    1
 readprimitive   exponential    1
 readprimitive   sine    1
 readprimitive   cosine    1
 readprimitive   tangent    1
 readprimitive   minus    1
 readprimitive   readexpr    2
 readpower   power    2
 readpower   readprimitive    1
 readterm   readpower    2
 readterm   product    2
 readterm   fraction    2
 readexpr   readterm    2
 readexpr   sum    2 ;
```

```
# checksum.run

model checksum.mod;
data checksum.dat;
option solver cplexstudent;
solve;
display data;
printf "L = {\n";
for {(i,j) in E : x[i,j] = 1} {
  printf "      (%s,%s)\n", i, j;
}
printf "    }\n";
```

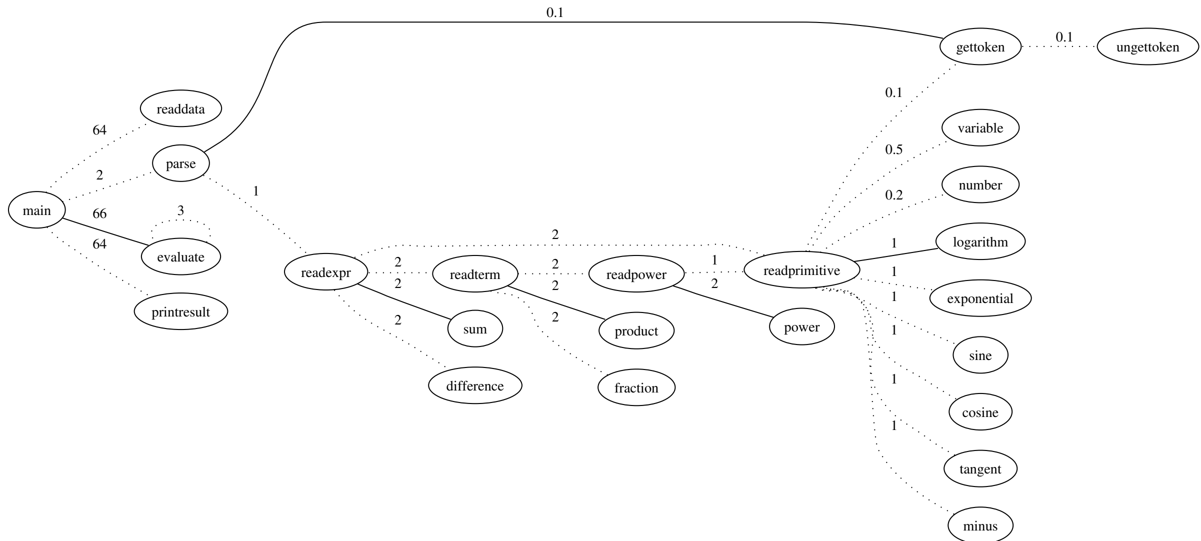
5.3.1.3 CPLEX solution

```
CPLEX 8.1.0: optimal integer solution; objective 73.1
3 MIP simplex iterations
0 branch-and-bound nodes
data = 73.1

L = {
      (main,evaluate)
      (parse,gettoken)
      (readprimitive,cosine)
      (readpower,power)
      (readterm,product)
      (readexpr,sum)
```

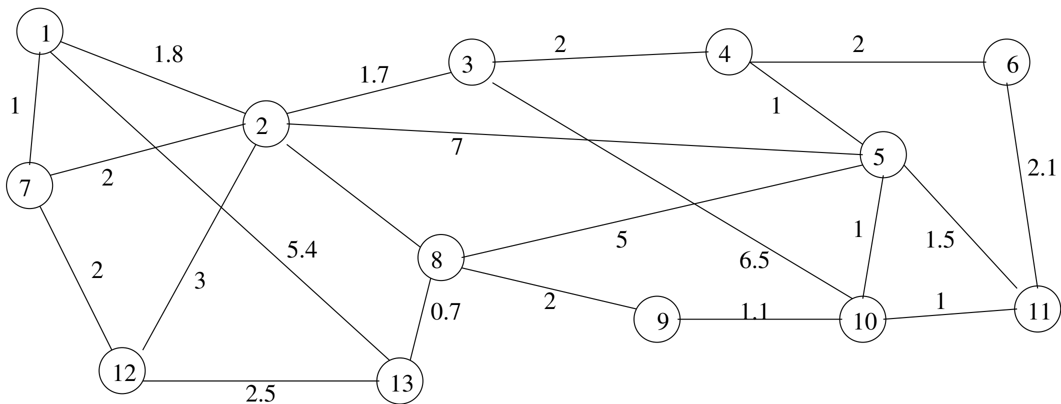
```
}
```

The picture below is the solution represented on the graph.



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?

5.4.1 Solution

Let $G = (V, E)$ be the graph of the network. The problem can be formalized as looking for the partition of V in three disjoint subsets V_1, V_2, V_3 such that the sum of the backbone update cost are minimum on the edges having one adjacent vertex in a set of the partition, and the other adjacent vertex in another set of the partition. This problem is often called GRAPH PARTITIONING or MIN-k-CUT problem.

5.4.1.1 Formulation and linearization

- *Indices*: $i, j \in V$ and $h, k \in K = \{1, 2, 3\}$.
- *Parameters*:
  - for each $\{i, j\} \in E$, d_{ij} is the edge weight (distance between i and j);
  - c: backbone updating cost;
  - m: minimum cardinality of the subnetworks.
- *Variables*: for each $i \in V$, $h \in K$, let $x_{ih} = 1$ if vertex i is in V_h, and 0 otherwise.
- *Objective function*:

$$\min \frac{1}{2} \sum_{h \neq k \in K} \sum_{\{i,j\} \in E} c d_{ij} x_{ih} x_{jk}$$

- *Constraints*:

$$\forall i \in V \qquad \sum_{k \in K} x_{ik} = 1; \text{ (assignment)} \tag{5.3}$$

$$\forall h \in K \qquad \sum_{i \in V} x_{ik} \geq m; \text{ (subnetwork cardinality).} \tag{5.4}$$

This formulation involves products between binary variables, and can therefore be classified as a Binary Quadratic Program (BQP). Its feasible region is nonconvex (due to the integrality constraints and the quadratic terms), and the continuous relaxation of its feasible region is also nonconvex (due to the quadratic terms). This poses additional problems to the calculation of the lower bound within Branch-and-Bound (BB) type solution algorithms. However, the formulation can be linearized exactly, which means that there exists a Mixed-Integer Linear Programming (MILP) formulation of the problem whose projection in the x-space of the BQP yields exactly the same feasible region. The above program can be reformulated as follows:

1. replace each quadratic product $x_{ih} x_{jk}$ by a continuous *linearization variable* w_{ij}^{hk} constrained by $0 \leq w_{ij}^{hk} \leq 1$;
2. add the following constraints to the formulation:

$$\forall \{i, j\} \in E, h \neq k \in K \quad w_{ij}^{hk} \geq x_{ih} + x_{jk} - 1 \quad (\text{if } x_{ih} = x_{jk} = 1,\ w_{ij}^{hk} = 1) \tag{5.5}$$

$$\forall \{i, j\} \in E, h \neq k \in K \quad w_{ij}^{hk} \leq x_{ih} \quad (\text{if } x_{ih} = 0,\ w_{ij}^{hk} = 0) \tag{5.6}$$

$$\forall \{i, j\} \in E, h \neq k \in K \quad w_{ij}^{hk} \leq x_{jk} \quad (\text{if } x_{jk} = 0,\ w_{ij}^{hk} = 0). \tag{5.7}$$

Constraints (5.5)-(5.7) are a way to express the equation $w_{ij}^{hk} = x_{ih} x_{jk}$ (i.e. the condition vertex i assigned to subnetwork h and vertex j assigned to subnetwork k) without introducing quadratic products in the formulation. The resulting formulation is a MILP whose continuous relaxation is a Linear Programming problem (hence it is convex, which implies that each local optimum is also global — so it can be safely used to compute lower bounds in BB algorithms such as that implemented in CPLEX).

5.4.1.2 AMPL model, data, run

```
# network design
param n >= 0, integer;
param k >= 1, integer;
set V := 1..n;
set K := 1..k;
param c >= 0;
param m >= 0, integer;
param d{V,V} >= 0 default 0;

var x{V,K} binary;
var w{V,V,K,K} >= 0, <= 1;

minimize cost :  sum{h in K, l in K, i in V, j in V :
                      h != l and i < j and d[i,j] > 0} c*d[i,j]*w[i,j,h,l];
subject to assignment {i in V} : sum{h in K} x[i,h] = 1;
subject to cardinality {h in K} : sum{i in V} x[i,h] >= m;
subject to linearization {h in K, l in K, i in V, j in V :
                            h != l and i < j and d[i,j] > 0} :
                            w[i,j,h,l] >= x[i,h] + x[j,l] - 1;
```

```
# netdes.dat
param n := 13;
param k := 3;
param c := 25;
param m := 2;
param d :=
  1  2  1.8
  1  7  1
  2  3  1.7
  2  5  7
  2  7  2
  2  12 3
  3  4  2
  3  10 6.5
  4  5  1
  4  6  2
  5  8  5
  5  10 1
  5  11 1.5
  6  11 2.1
  7  12 2
  8  9  2
  8  13 0.7
  9  10 1.1
  10 11 1
  12 13 2.5 ;
```

```
# netdes.run

model netdes.mod;
data netdes.dat;
for {i in V, j in V : i < j} {
  let d[j,i] := d[i,j];
}
option solver cplexstudent;
solve;
```

```
display cost;
for {h in K} {
  printf "subnetwork %d:", h;
  for {i in V} {
    if (x[i,h] == 1) then {
      printf " %d", i;
    }
  }
  printf "\n";
}
```

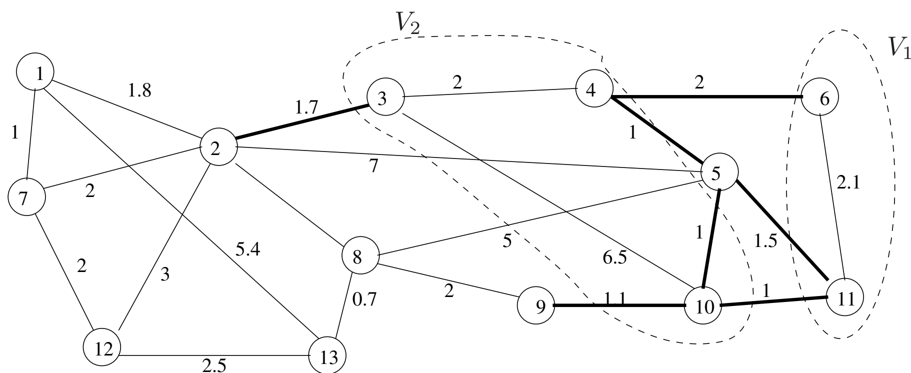
5.4.1.3 CPLEX solution

For $k = 3$:

```
CPLEX 8.1.0: optimal integer solution; objective 232.5
1779 MIP simplex iterations
267 branch-and-bound nodes
cost = 232.5

subnetwork 1: 6 11
subnetwork 2: 3 4 10
subnetwork 3: 1 2 5 7 8 9 12 13
```

The solution is in the picture below.

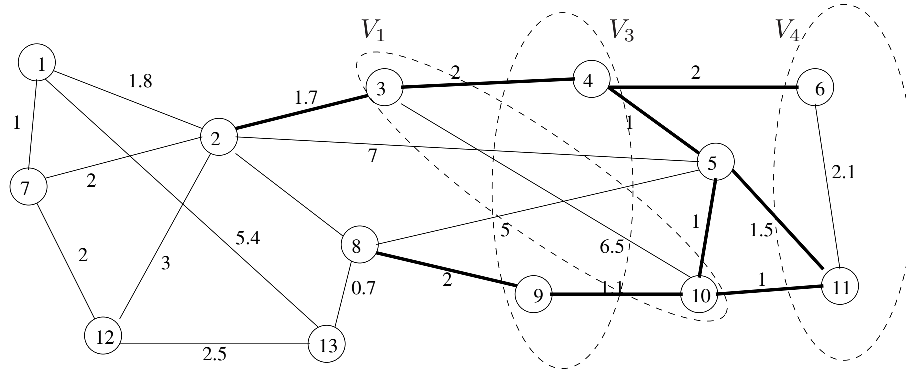


For $k = 4$:

```
CPLEX 8.1.0: optimal integer solution; objective 332.5
18620 MIP simplex iterations
1403 branch-and-bound nodes
cost = 332.5

subnetwork 1: 1 2 5 7 8 12 13
subnetwork 2: 4 9
subnetwork 3: 3 10
subnetwork 4: 6 11
```

The solution is in the picture below.

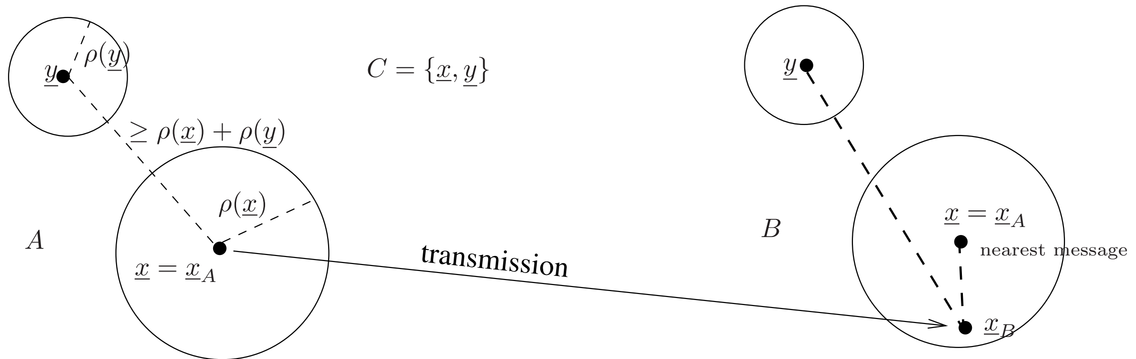


5.5 Error correcting codes

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 $\mathbb{R}^12$ (where all message components are in $[0, 1]$) so that the correction radius is maximized.

5.5.1 Solution

1. *Indices*: $j \leq m$, $i \leq n$.
2. *Variables*:
   - $\underline{x}^i \in \mathbb{R}^m$: position of i-th message;
   - $\rho_i \in \mathbb{R}_+$: value of ρ on $\underline{x}^i$
3. *Objective function*:

$$\max \min_{i \leq n} \rho_i$$

4. *Constraints*:
   - (coordinates limits)
   $$0 \leq \underline{x}^i_j \leq 1 \qquad \forall\, i \leq n, j \leq m$$
   - (distances)
   $$||\underline{x}^i - \underline{x}^k|| \geq \rho_i + \rho_k \qquad \forall\, i, k \leq n$$

The AMPL implementation and solution (to be carried out by the MINOS solver because the model is nonlinear) is left as an exercise.

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.

5.6.1 Solution

This problem can be modelled as follows.

- *Parameters*:
  1. Let $N \in \mathbb{N}$;
  2. for all $i \leq n$, s_i is the expected number of invocations;
  3. for all $i \leq n, j \in I_i$, c_{ij} is the cost, d_{ij} is the delivery time, and μ_{ij} the probability of failure on demand of the j-th off-the-shelf component for slot i;
  4. for all $i \leq n, j \in J_i$, $\bar{c}_{ij}$ is the cost, t_{ij} is the estimated development time, τ_{ij} the average time required to perform a test case, p_{ij} is the probability that the instance is faulty, and b_{ij} the testability of the j-th purpose-built component for slot i.
- *Variables*:
  1. Let $x_{ij} = 1$ if component $j \in I_j \cup J_i$ is chosen for slot $i \leq n$, and 0 otherwise;
  2. Let $N_{ij} \in \mathbb{Z}$ be the (non-negative) number of tests to be performed on the purpose-built component $j \in J_i$ for $i \leq n$: we assume $N_{ij} \in \{0, \ldots, N\}$.
- *Objective function*. We minimize the total cost, i.e. the cost of the off-the-shelf components c_{ij} and the cost of the purpose-built components $\bar{c}_{ij}(t_{ij} + \tau_{ij} N_{ij})$:
  $$\min \sum_{i \leq n} \left(\sum_{j \in I_i} c_{ij} x_{ij} + \sum_{j in J_i} \bar{c}_{ij}(t_{ij} + \tau_{ij} N_{ij}) x_{ij} \right).$$

- *Constraints*:

  1. assignment constraints: each component slot in the architecture must be filled by exactly one software component
  $$\forall i \le n \quad \sum_{j \in I_i \cup J_i} x_{ij} = 1;$$
  2. delivery time constraints: the delivery time for an off-the-shelf component is simply d_{ij}, whereas for purpose-built components it is $t_{ij} + \tau_{ij} N_{ij}$
  $$\forall i \le n \quad \sum_{j \in I_i} d_{ij} x_{ij} + \sum_{j \in J_i} (t_{ij} + \tau_{ij} N_{ij}) x_{ij} \le T;$$
  3. reliability constraints: the probability of failure on demand of off-the shelf components is μ_{ij}, whereas for purpose-built components it is given by
  $$\vartheta_{ij} = \frac{p_{ij} b_{ij} (1 - b_{ij})^{(1-b_{ij})N_{ij}}}{(1 - p_{ij}) + p_{ij}(1 - b_{ij})^{(1-b_{ij})N_{ij}}},$$
  so the probability that no failure occurs during the execution of the i-th component is
  $$\varphi_i = e^{-s_i \left(\sum\limits_{j \in I_i} \mu_{ij} x_{ij} + \sum\limits_{j \in J_i} \vartheta_{ij} x_{ij} \right)},$$
  whence the constraint is
  $$\prod_{i \le n} \varphi_i \ge R;$$
  notice we have three classes of reliability constraints involving two sets of added variables ϑ, φ.

This problem is a MINLP with no continuous variables. We shall now apply several reformulations to this problem (call it P).

1. We take the logarithm of both sides of the constraint $\prod_i \varphi_i \ge R$ to obtain:
$$-\sum_{i \le n} s_i \left(\sum_{j \in I_i} \mu_{ij} x_{ij} + \sum_{j \in J_i} \vartheta_{ij} x_{ij} \right) \ge \log(R).$$
2. We now make use of the fact that N_{ij} is an integer variable for all $i \le n, j \in J_i$. For $k \in \{0, \ldots, N\}$ we add assignment variables ν_{ij}^k so that $\nu_{ij}^k = 1$ if $N_{ij} = k$ and 0 otherwise. Now for all $k \in \{0, \ldots, N\}$ we compute the constants $\vartheta^k = \frac{p_{ij} b_{ij} (1-b_{ij})^{(1-b_{ij})k}}{(1-p_{ij}) + p_{ij}(1-b_{ij})^{(1-b_{ij})k}}$, which we add to the problem parameters. We remove the constraints defining ϑ_{ij} in function of N_{ij}. Since the following constraints are valid:
$$\forall i \le n, j \in J_i \quad \sum_{k \le N} \nu_{ij}^k = 1 \qquad (5.8)$$
$$\forall i \le n, j \in J_i \quad \sum_{k \le N} k \nu_{ij}^k = N_{ij} \qquad (5.9)$$
$$\forall i \le n, j \in J_i \quad \sum_{k \le N} \vartheta^k \nu_{ij}^k = \vartheta_{ij}, \qquad (5.10)$$

the second set of constraints are used to replace N_{ij} and the third to replace ϑ_{ij}. The first set is added to the formulation. We obtain:

$$\begin{array}{rcl}
\min \sum_{i\le n}\left(\sum_{j\in I_i} c_{ij}x_{ij} + \sum_{j\in J_i}\bar{c}_{ij}(t_{ij}+\tau_{ij}\sum_{k\le N} k\nu_{ij}^k)x_{ij}\right) & & \\
\forall i\le n \quad \sum_{j\in I_i\cup J_i} x_{ij} & = & 1 \\
\forall i\le n \quad \sum_{j\in I_i} d_{ij}x_{ij} + \sum_{j\in J_i}(t_{ij}+\tau_{ij}\sum_{k\le N}k\nu_{ij}^k)x_{ij} & \le & T \\
\sum_{i\le n} s_i\left(\sum_{j\in I_i}\mu_{ij}x_{ij} + \sum_{j\in J_i}x_{ij}\sum_{k\le N}\vartheta^k\nu_{ij}^k\right) & \ge & \log(R) \\
\forall i\le n, j\in J_i \quad \sum_{k\le N}\nu_{ij}^k & = & 1.
\end{array}$$

3. We distribute products over sums in the formulation to obtain the binary product sets $\{x_{ij}\nu_{ij}^k \mid k\le N\}$ for all $i\le n, j\in J_i$. We replace each binary product $x_{ij}\nu_{ij}^k$ (with indices ranging over all the appropriate ranges) by continuous *linearizing variables* w_{ij}^k defined over $[0,1]$ and add the following constraints: $w_{ij}^k \le x_{ij}$, $w_{ij}^k \le \nu_{ij}^k$, and $w_{ij}^k \ge x_{ij}+\nu_{ij}^k - 1$; these supply a well-known exact linearization for products of binary variables [5]. By repeatedly applying this reformulation to all binary products of binary variables, we get a MILP reformulation Q of P where all the variables are binary.

We remark that the MILP reformulation Q derived above has a considerably higher cardinality than $|P|$. More compact reformulations are applicable in step 3 because of the presence of the assignment constraints [11].

A semantic interpretation of step 3 is as follows. Notice that for $i\le n, j\in J_i$, if $x_{ij}=1$, then $x_{ij}=\sum_k \nu_{ij}^k$ (because only one value k will be selected), and if $x_{ij}=0$, then $x_{ij}=\sum_k\nu_{ij}^k$ (because no value k will be selected). This means that

$$\forall i\le n, j\in J_i \quad x_{ij}=\sum_{k\le N}\nu_{ij}^k \qquad (5.11)$$

is a valid problem constraint. We use it to replace x_{ij} everywhere in the formulation where it appears with $j\in I_i$, obtaining a opt-reformulation with x_{ij} for $j\in I_i$ and quadratic terms $\nu_{ij}^k\nu_{lp}^h$. Now, because of (5.8), these are zero when $(i,j)\ne(l,p)$ or $k\ne h$ and are equal to ν_{ij}^k when $(i,j)=(l,p)$ and $k=h$, so they can be linearized exactly by replacing them by either 0 or ν_{ij}^k according to their indices. What this really means is that the reformulation Q, obtained through a series of automatic reformulation steps, is a semantically different formulation defined in terms of the following decision variables: $\forall i\le n, j\in I_i$, $x_{ij}=1$ if $j\in I_i$ is assigned to i and 0 otherwise; and $\forall i\le n, j\in J_i, k\le N$, $\nu_{ij}^k=1$ if $j\in J_i$ is assigned to i and there are k tests to be performed, and 0 otherwise.

The AMPL implementation of the reformulation and consequent CPLEX solution is left as an exercise.

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.

6.4 The software architecture

According to the above requirements, the overall software architecture is based on three main modules:

1. *user interface*;
2. *log reading daemon*;

3. *log database server*;

plus an optional added *project interface* module to configure project-specific data into the log analysis database. The overall software architecture is depicted in Fig. 6.1.

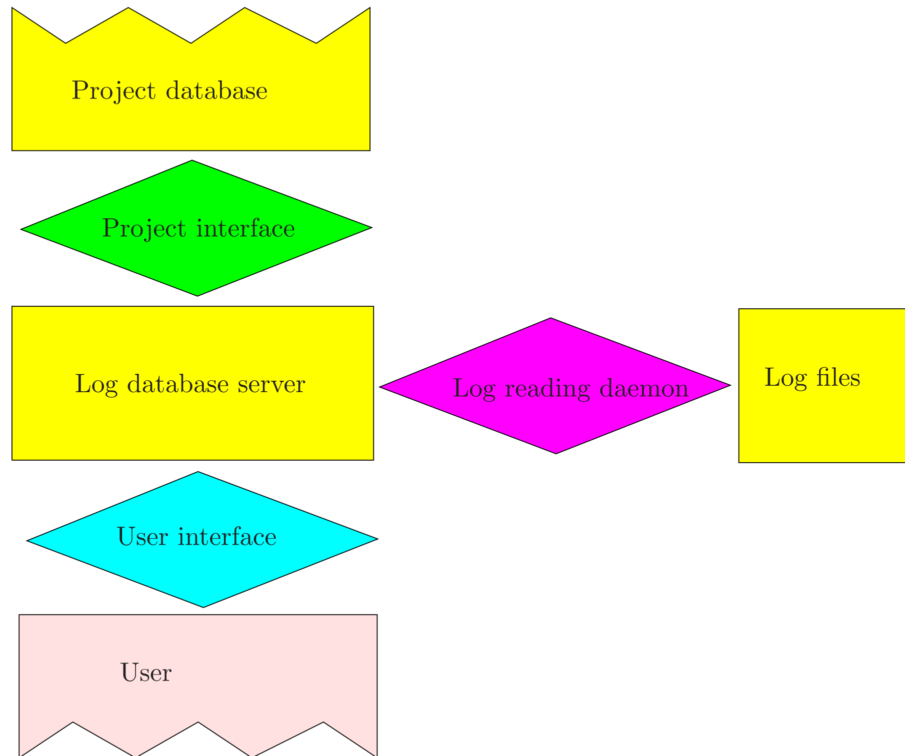


Figure 6.1: The overall software architecture.

Project-specific data are: (a) a set of tribes (possibly with hierarchical/functional relationships expressed via a set of edges in the graph induced by the tribes); (b) a set of zones (possibly with semantic relationships expressed via a set of edges in the graph induced by the zones); (c) a document-to-zone map (here we refer to the documents listed in the project webserver log files); (d) an IP-to-tribe map (where IP is the IP address requesting documents from the project webserver log files).

6.4.1 Summary

6.4.1.1 User interface

This is the most complex module. It needs to perform the following actions (in the given order):

1. configure its runtime parameters: project name, DB server information access, XSL specification for statistics visualization output;
2. get project-specific (list of tribes, list of zones) information from the log database server
3. ask the user the desired type of statistic (per-tribe, per-zone, global, time-graph);
4. ask the user the necessary input data (timespan, tribe(s), zone(s), tribe-zone pair), presenting lists of tribes, zones and pairs to choose from;

5. form the database query according according to user specification;
6. perform the database query;
7. gather output statistics;
8. form an XML representation of the statistics visualization
9. produce an output (HTML, other publishing formats).

Should any action fail in the events sequence, the correct error recovering procedure is simply to abort the sequence, display an error, and return to Step 2.

6.4.1.2 Log reading daemon

The log reading daemon simply waits in the background and every so often reads the tail of the log files, parses it, and records the data in the log database server. It needs to perform the following actions (in the given order):

1. configure its runtime parameters: project name, DB server information access, log file format specification, uniform resource identifiers (URIs) of log files, update information file name;
2. read log file sizes when last accessed from the update information file;

then, for each log file listed:

3. get tail of log file;
4. parse records according to log file format, to extract: (i) requesting IP address, (ii) requested document URL, (iii) access date/time, (iv) success/failure, (v) access type (read/write);
5. store those data to the log database server.

Care must be taken to read a whole number of records in Step 3, as the "tail" of a file is defined on the amount of bytes last read. This depends on the operating system, so it cannot be enforced a priori. One possible way around is to count the bytes used during data parsing, and add those bytes the file size stored in the update information file.

Should any action fail in the events sequence, the correct error recovering procedure is to abort the daemon and notify a system administrator immediately.

6.4.1.3 Log database server

The log database server is going to perform the necessary computation on the (stored) relevant information. It needs to store the following information:

1. project-specific information:
   - tribes table: tribe name, associated IP address pool
   - zones table: zone name, associated directory name in web site map
   - actors/tribes incidence information (optional)
   - documents/zones incidence information (optional)

- hierarchical/functional tribes/actors relationthips (optional)
- semantic zones relationships (optional);

2. log information table:
   - requesting IP address
   - tribe name
   - requested document URL
   - zone name
   - access date/time
   - type of access (read/write).

Note that the zone name can be inferred by the directory name of the document URL (contained in the zones table), and the tribe name can be inferred by the IP address according to the tribes table.

6.4.1.4 Project Interface

This module is optional in the sense that a prototype may well work without it. Its main function is to load incidence information of document URLs with zones and IP addresses with tribes on the database server. This can be carried out either as a web interface drawing input from the user or as an executable program configured through a text file. In both cases, this interface should hook into the project-specific database to build the document-to-zone and IP-to-tribe tables.

6.4.2 Details

The user interface and log reading daemon expose a C-like API. API entries are listed in the following format:

`ReturnType` **FunctionName** (*in* `InputArgument`, ..., *out* `OutputArgument`, ...)

In this document, all functions return an integer error status (this can be changed to using exceptions where applicable). The `TimeSpan` type is simply a pair of date/time records (starting and ending times).

6.4.2.1 User interface

The user interface is going to be coded in PHP. It is going to make use of several primitive PHP API subsets: text file handling, abstract DB connection and query, XML/XSL, vector image creation.

- `ErrorStatus` **ReadConfiguration** (*in* `String FileName`, *out* `DBConnectionData theDB`, *out* `String XSLVisualSpecFileName`)
  It opens a text configuration file named `FileName`; reads the following information: name of the project, DB server name, DB user name, DB password, DB database name, XSL visual specification file name; finally, it closes the configuration file.
- `ErrorStatus` **GetTribesList** (*out* `List TribesList`)
  Queries the DB engine to obtain the zones list.
- `ErrorStatus` **GetZonesList** (*out* `List ZonesList`)
  Queries the DB engine to obtain the zones list.

- `ErrorStatus` **GetUserSpecifications** (*out* `int StatisticsType`, *out* `String[3] SelectedTribes`, *out* `String[3] SelectedZones`, *out* `TimeSpan theTimeSpan`, *out* `int AccessType`)
  Gets the user specifications for the desired statistics from a web form. The `StatisticsType` will denote per-tribe, per-zone, global or time-graph. If per-tribe is selected, `SelectedTribes` contains up to three names of meaningful tribes. If per-zone is selected, `SelectedZones` contains up to three names of meaningful zones. If time-graph is selected, `SelectedTribes[0]` and `SelectedZones[0]` will contain the relevant tribe-zone pair. In all cases, `AccessType` will denote read access, write access or both.

- `ErrorStatus` **GetByTribeStatistics** (*in* `String[3] SelectedTribes`, *in* `TimeSpan theTimeSpan`, *in* `AccessType`, *in* `DBConnectionData theDB`, *out* `Map⟨⟨Tribe,Zone⟩,double⟩ Statistics`)
  Forms the SQL query to count how many accesses occurred during the specified timespan from the selected tribes to each zone; performs the query; organizes the data in the specified output map.

- `ErrorStatus` **GetByZoneStatistics** (*in* `String[3] SelectedZones`, *in* `TimeSpan theTimeSpan`, *in* `AccessType`, *in* `DBConnectionData theDB`, *out* `Map⟨⟨Tribe,Zone⟩,double⟩ Statistics`)
  Forms the SQL query to count how many accesses occurred during the specified timespan from each tribe to the selected zones; performs the query; organizes the data in the specified output map.

- `ErrorStatus` **GetGlobalStatistics** (*in* `TimeSpan theTimeSpan`, *in* `AccessType`, *in* `DBConnectionData theDB`, *out* `Map⟨⟨Tribe,Zone⟩,double⟩ Statistics`)
  Forms the SQL query to count how many accesses occurred during the specified timespan from each tribe to each zone; performs the query; organizes the data in the specified output map.

- `ErrorStatus` **GetTimeGraphStatistics** (*in* `String Tribe`, *in* `String Zone`, *in* `TimeSpan theTimeSpan`, *in* `AccessType`, *in* `DBConnectionData theDB`, *out* `Map⟨DateTime,double⟩ Statistics`)
  Forms the SQL query to track the access of `Tribe` towards `Zone` versus time; performs the query; organizes the data in the specified output map.

- `ErrorStatus` **PublishIncidenceStatistics** (*in* `Map⟨⟨Tribe,Zone⟩,double⟩ Statistics`, *in* `String XSLVisualSpecFileName`, *out* `TextBuffer XMLStatistics`)
  Transforms the incidence `Statistics` map into XML format; uses the PHP PNG vector graphics API subset to produce a JPEG image of the desired graph (in full colours); reads the specified XSL visual specification file to produce an HTML output (which also displays the GIF/JPEG image directly on the screen).

- `ErrorStatus` **PublishTimeGraphStatistics** (*in* `Map⟨DateTime,double⟩ Statistics`, *in* `String XSLVisualSpecFileName`, *out* `TextBuffer XMLStatistics`)
  Transforms the time-graph `Statistics` map into XML format; uses the PHP PNG vector graphics API subset to produce a PNG image of the desired graph (in full colours); transforms this to GIF or JPEG format; reads the specified XSL visual specification file to produce an HTML output (which also displays the GIF/JPEG image directly on the screen).

Note to implementors: some of the above functions are extensive pieces of coding. They should be implemented by breaking them up into smaller (protected) functions.

The transition state diagram for the user interface is given in Fig. 6.2. The class diagram is given in Fig. 6.3. Note to implementors: this class diagram is intended to give a semantic grouping of the required data and methods. PHP is not necessarily best used in object-oriented mode. Should the choice fall on a procedural PHP development, the class diagram should just be used for clarification and as general guidance.

6.4.2.2 Log reading daemon

The log file daemon is going to be coded in Java and will use the following primitive JAVA API subsets: process/timer handling, text file handling, abstract DB connection and query, and possibly an advanced

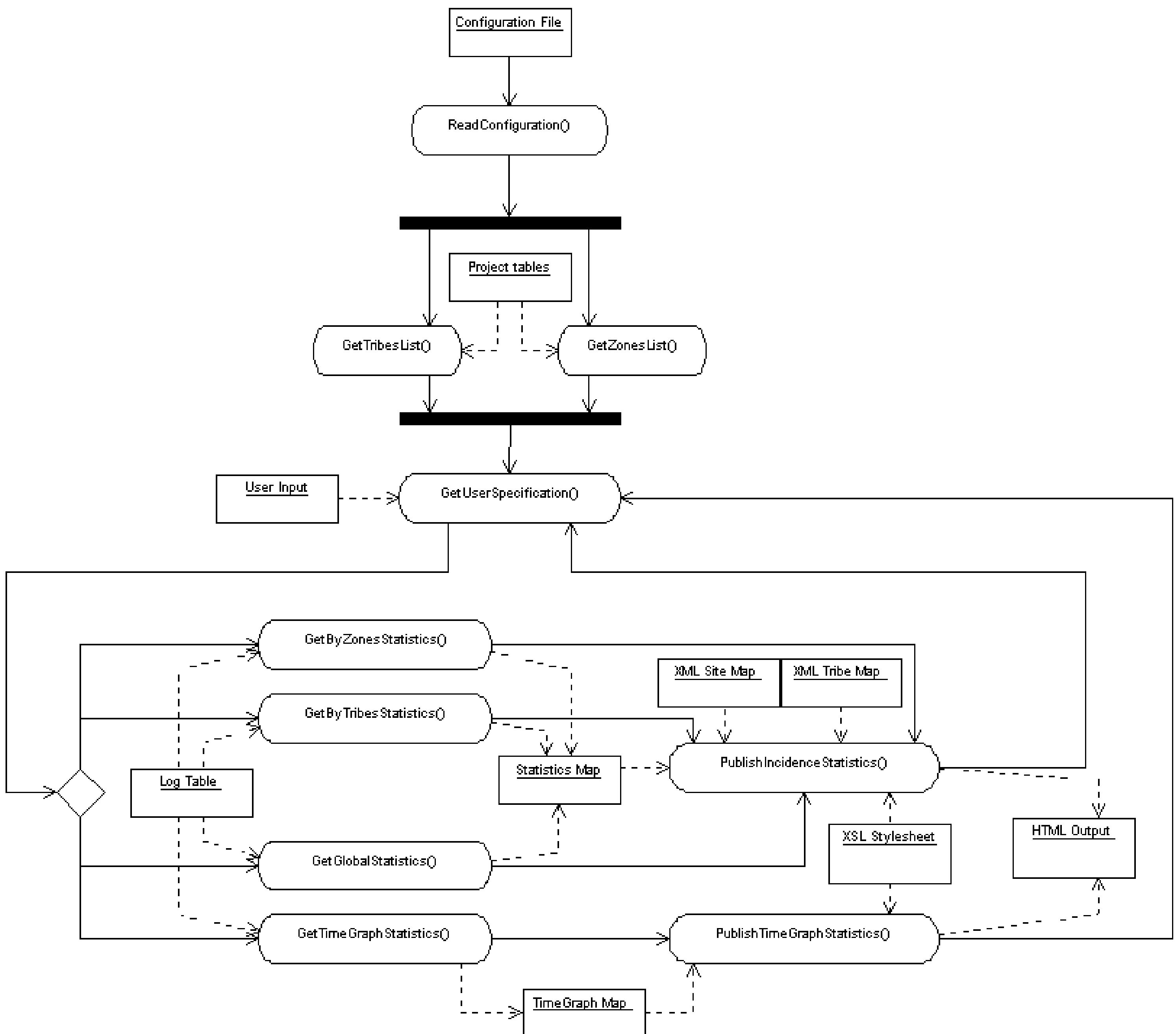


Figure 6.2: The transition state diagram for the user interface.

(s)FTP API to retrieve log file tails.

- `ErrorStatus` **ReadConfiguration** (*in* `String FileName`, *out* `DBConnectionData theDB`, *out* `int LogFileFormat`, *out* `String[] LogURI`, *out* `String UpdateInfoFileName`)
  Opens a text configuration file named `FileName`; reads the following information: name of the project, DB server name, DB user name, DB password, DB database name, log file format, uni-

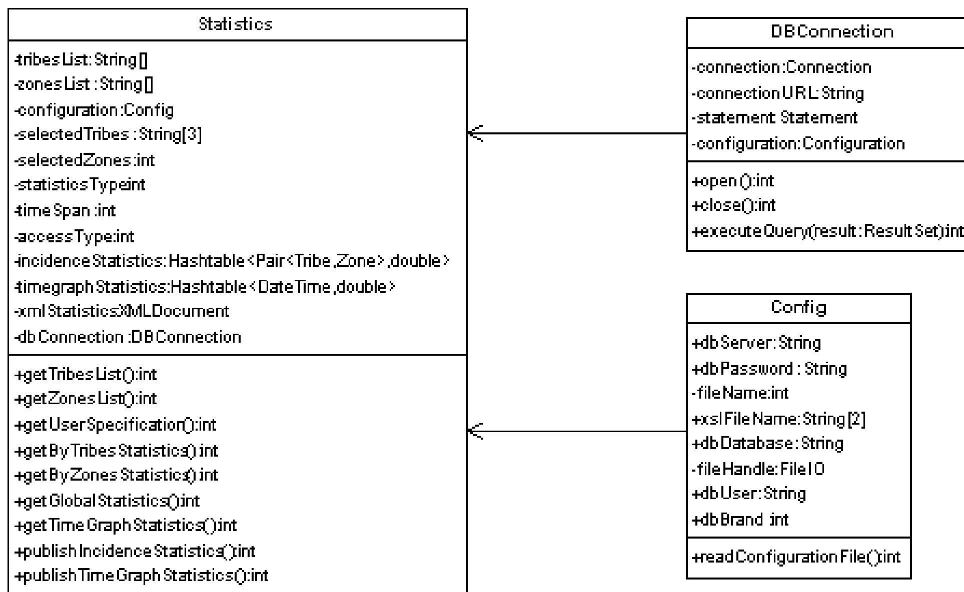


Figure 6.3: The class diagram for the user interface.

form resource identifiers (URIs) of log files, update information file name; finally, it closes the configuration file.

- `ErrorStatus` **ReadUpdateInfo** (*in* `String UpdateInfoFileName`, *in* `String[] LogURI`, *out* `long[] LogSize`)
  Opens the update information file; reads a "file read up to size" for each given log file URI; closes the file.

- `ErrorStatus` **SaveUpdateInfo** (*in* `String UpdateInfoFileName`, *in* `String[] LogURI`, *in* `long[] LogSize`)
  Writes a new update information file with the current log sizes; closes the file.

- `ErrorStatus` **ReadLogFileTail** (*in* `String LogURI`, *in* `long LogSize`, *out* `TextBuffer theTail`)
  Uses a network or filesystem transfer method to retrieve the last (filesize(`LogURI`) − `LogSize`) bytes of the log file `LogURI`.

- `ErrorStatus` **ParseLogData** (*in* `TextBuffer theTail`, *in* `int LogSize`, *in* `int LogFileFormat`, *out* `DBTable UpdatedLogData`, *out* `UpdatedLogSize`)
  Calls a lower-level parsing driver according to `LogFileFormat`; this driver must parse `theTail`, identify the relevant fields and organize them into a memory representation of a DB table `UpdatedLogData`; furthermore, it must return the exact number of bytes used during the parsing, add them to the `LogSize` and put the result into `UpdatedLogSize`. The format of the DB table is as in Section 6.4.1.3, Step 2 (the tribe and zone name fields are left blank).

- `ErrorStatus` **SaveLogData** (*in* `DBTable UpdatedLogData`, *in* `DBConnectionData theDB`)
  Connects to the log database server; finds the tribe corresponding to each IP address in the DB table; finds the zone corresponding to each document URL in the DB table; completes the table; saves the latest log data in the log database server, adding them to the relevant table.

Notes to implementors. (a) Some of the above functions are extensive pieces of coding. They should be implemented by breaking them up into smaller (protected) functions. (b) The `main` function of this

program should take a number of seconds s as argument, and configure and start a timer calling the log reading procedure every s seconds.

The transition state diagram for the log reading daemon is given in Fig. 6.4, and the class diagram in Fig. 6.5.

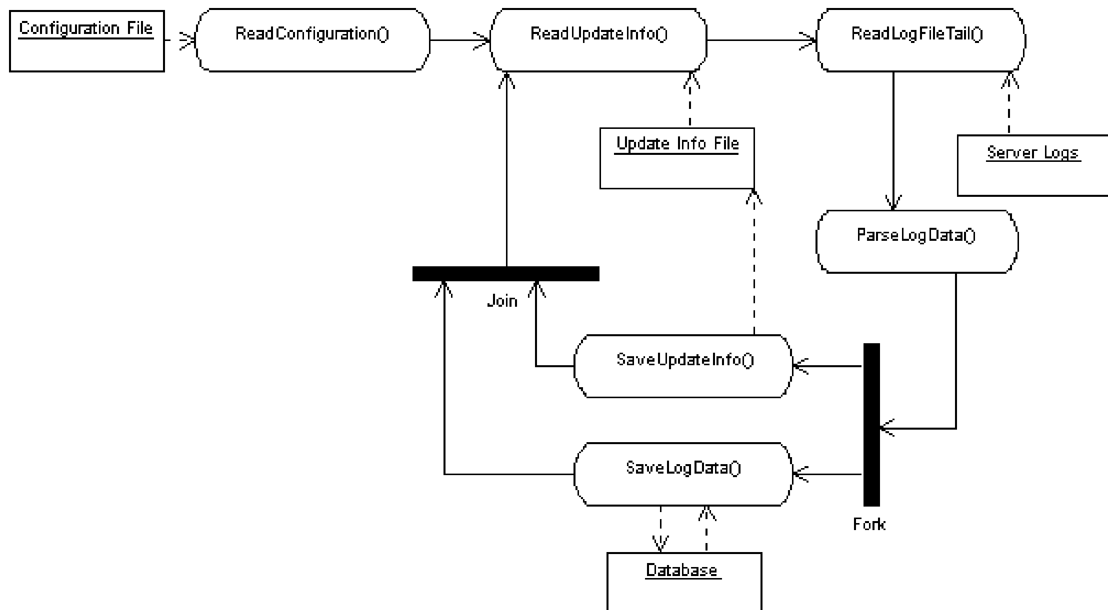


Figure 6.4: The transition state diagram for the log reading daemon.

6.4.2.3 Log database server

The database server of choice is MySQL (`www.mysql.com`), but this can be changed as desired with any other internet-enabled DB engine accepting SQL queries and exporting data through the normal standardized APIs.

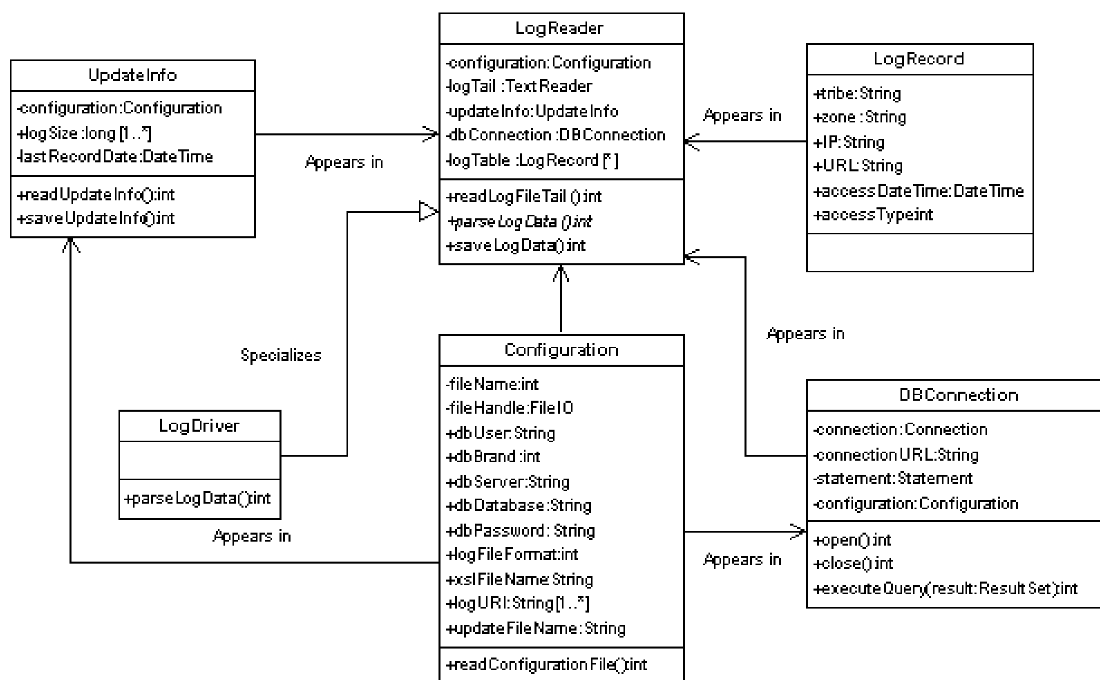


Figure 6.5: The class diagram for the log reading daemon.

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.

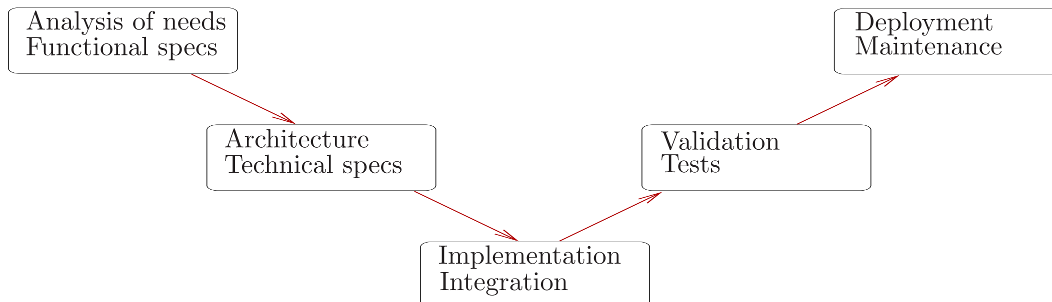


Figure 7.1: The "V" development process.

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?

7.2.1.1 Meeting output

1. 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 and look in a project database for all projects which were sufficiently similar at the initial stage and proceeded to completion with a uniform degree of success; we should then display a list of such projects so that the user can immediately glance at all important information concerning risk-assessment.
   (a) The deliverable is a software module that must be plugged in the existing T-Sale back-office network; it should have query access to some of the T-Sale databases and should be usable through a web interface.
   (b) At a first analysis, we shall need:
      - I/O user interface through a web browser;
      - a way to find meaningful indicators in the given project;
      - a system to query the databases for those indicators and return information about initial, intermediate and final cost, time and resources estimates.
2. We shall need T-Sales' data concerning:
   - project descriptions;

- project schedules;
- project costs;
- teams involved;
- people involved;
- other resources involved.

T-Sale's answer, as often happens, is rather vague.

> Dear VirtualClass Team,
> We are sorry to have to tell you that the structure of our databases is classified information, and we will only be able to give it to you at a later stage when and if we choose to employ your services. We can however describe the main features of what we think is useful to your job. We have an HR database detailing the usual information (salary, rank,. . .) abilities and skills. We have a technical database with project information (nature and cost of project, teams, people, schedule and associated changes). We naturally have a commercial database detailing customers and payments. Unfortunately the database which details hardware resources and costs may not be accessed as it contains some information classified at national level.
> Best regards,
> A. Smith

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.

7.2.2.1 Meeting output

Main idea. Collect all data from past projects and cluster data according to different indicators (i.e. technological area to which the project belongs, type of architecture topology, expertise needed, total projected cost, total actual cost, risk. . .) to get an idea of what it means for projects to be similar. Classify indicators according to whether they can be known at an early (i.e. technological area) or late stage in the project (e.g. total actual cost). Compare clusterings: if roughly same number of clusters and each cluster has roughly the same cardinality, we can infer that the two indicators are probably correlated. Assess correlations between all early/late indicator pairs. Classify new project according to early indicators, look at correlated late indicators and output the projects in the corresponding clusters (see Fig. 7.2).

1. User will input project indicators known at early stage
2. **Functionality**: an input web form (user interface)
3. Which among these "early indicators" are quantitative, which qualitative?
4. What sort of clustering of the project space do they lead to?
5. According to what other indicators ("late indicators") can project be clustered?

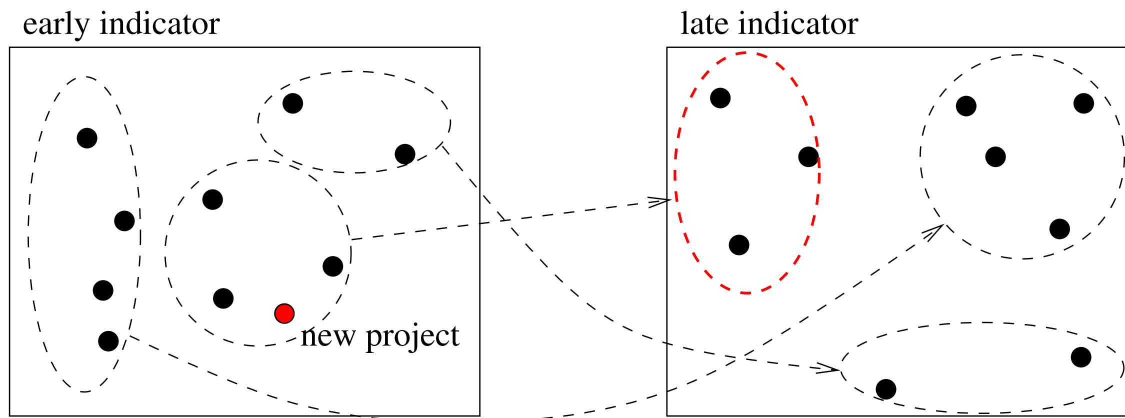


Figure 7.2: Main idea for the Proogle project.

6. **Functionality**: cluster projects according to a given quantitative/qualitative indicator (computational engine)

7. **Functionality**: access the customer database (database module)

8. How do we assess the quality of a clustering?

9. How "clear-cut" is a clustering?

10. **Functionality**: clustering significance evaluator (computational engine)

11. How are the early/late clusterings used later on?

12. **Functionality**: record a clustering (database module)

13. New projects must be classified according to early indicators: how do we use the information given by the clusterings obtained with late indicators? More in general, how do we pick a set of significant late clustering (which give the useful risk assessment information) given an early clustering?

14. **Functionality**: clustering compatibility evaluator (computational engine)

15. Literature review on clustering

16. How do we classify a new project according to the stored clusterings?

17. **Functionality**: query clusterings for

18. How do we present the output to the user?

19. **Functionality**: output form (user interface)

20. Name: how about "proogle" (the "project google"?)

The Proogle system will require the following functionalities:

- input/output web user interface;
- a computational engine for clustering according to a quantitative or qualitative indicator;
- a computational engine for evaluating clustering significance;
- a database module for storing clusterings;
- a computational engine for evaluating clustering compatibility;
- a database module for querying the stored clusterings.

Computational engines will require expertise in clustering techniques; database modules should be sufficiently straightforward; presenting output in a meaningful way will likely pose problems.

7.2.3 Formalization of a commercial offer

Aims of the meeting:

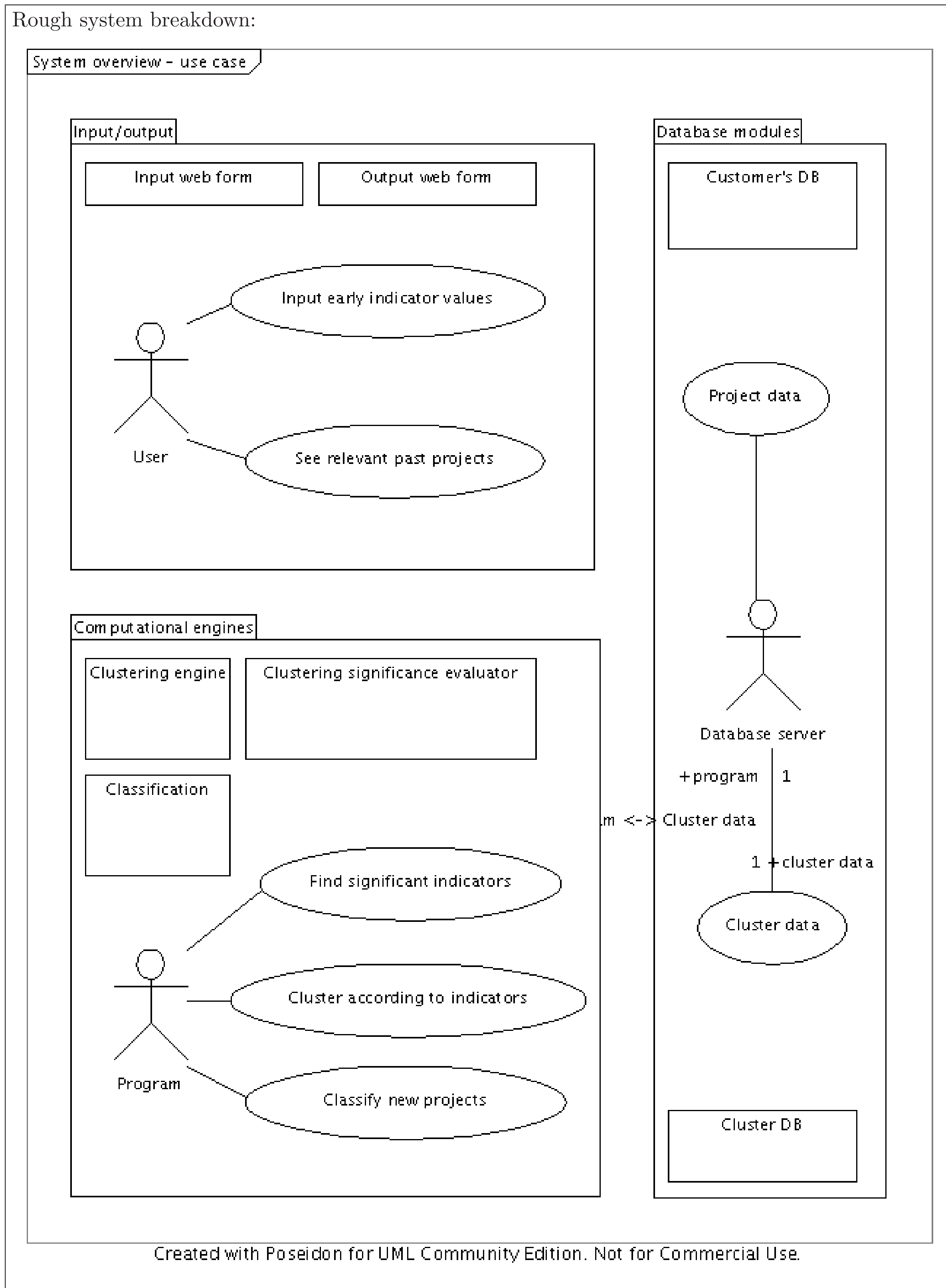
1. write a document (for internal use) which gives a rough overview of the system functionalities and of the system breakdown into sub-systems and interdependencies;

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.

7.2.3.1 Meeting output

Rough system breakdown:



Risks:

1. Failure to obtain necessary data/clearance from T-Sale — catastrophic, low probability
2. Not enough specific in-house clustering expertise — serious, high probability
3. Results not as useful as expected — low, medium probability

Address risks:

1. Insert clause in contract
2. Plan training
3. Insert clause in contract

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. 7.3.

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?

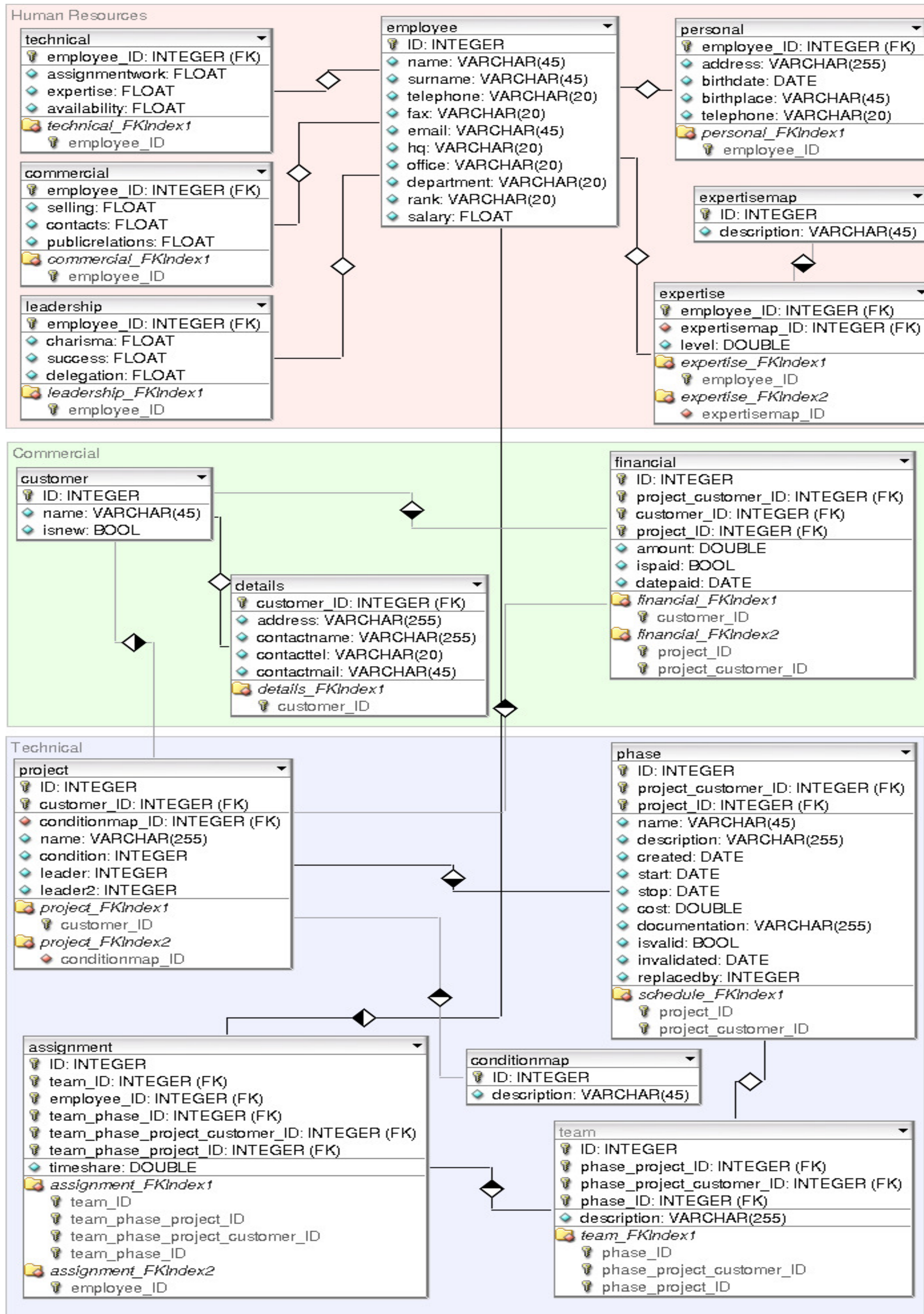


Figure 7.3: T-Sales' database structure.

7.3.1.1 Meeting output

1. The table among `technical`, `commercial`, `leadership` which contains the employee's ID gives its main occupation.

2. We consult the `expertise` table. For a description of the expertise, we consult `expertisemap`.

3. We simply look at the `condition` field of the `project` table, whose description is in `conditionmap`.

4. We scan the `phase` table for a given project and count the times the `isvalid` field contains 'true'.

5. We find the last phase of the project looking at the `phase` table and we compare the `stop` field with the `datepaid` field of the `financial` table.

6. We scan the `financial` table for a given customer, and find whether the completion date (`stop` field) of the last phase in the project (table `phase`, accessed through `project`) corresponds with the `datepaid` field of the `financial` table.

7. We sum the costs of all the non-invalidated phases in the project and compare it to the total project cost (`amount` field in `financial` table).

8. The function changes every time a phase is invalidated or created. The cost is the cumulative cost of all the phases which are valid at any given time.

9. Since we only know the costs due to human resources, we must find the salaries of all the people involved in the project and scale them by the percentage of their time they devoted to the project. In other words, we must sum the scaled salaries over all phases of the project, over all teams involved in the phase, and over all people associated to the team.

10. We find the people involved in the project through phases and teams, and we compute their expertise level vector.

11. Similar to the above.

12. We match the teams involved in a project with indicators such as the project's condition and the number of invalidated phases (the fewer, the better).

13. We find the subsets of people from a team which occur most often in the most unsuccessful projects.

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.

7.3.2.1 Meeting output

Here is one possible approach to solving these problems:

1. find all project indicators which are known at the initial stage (details of first phase, customer history, personal compatibilities in teams);
2. propose a sizable number of quantitative indices that can be associated to a project (initial projected cost, cost curve, required skill levels, total human resources cost, number of teams, number of people, total cost, ...);
3. cluster all projects with similar degree of success (i.e. look at the `condition` field and at the number of invalidated phases) and produce a partition of the set of projects such that all projects in a partition subset have the same degree of success;
4. the most meaningful quantitative indices in the proposed set are those having the least variance in each partition subset;
5. finding the variance of the project indicators in the partition subsets will give an idea of the indicator reliability.

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.

7.3.3.1 Solution

The only available point of departure for this analysis is the sketched architecture design contained in our commercial offer, which at this point should be used and expanded into a detailed and fully implementable software architecture. The following components are apparent:

1. **Input web form** (IWF): user inputs early indicator values concerning a new project
2. **Output web form** (OWF): user sees similar projects with relevant indicator values
3. **Clustering engine** (CE): given a set of objects and their pairwise distances, perform a clustering minimizing the inter-cluster distances
4. **Clustering significance evaluator** (CSE): Given a clustering, does it match well to another given clustering?
5. **Classification** (CLS): given an indicator for a given type of clustering, find the cluster it belongs to in the given and all similar clusterings
6. **Customer's DB**: split in *Commercial* (CDB), *Human resources* (HRDB), *Technical* (TDB) data repositories
7. **Clustering DB** (CLDB): repository for existing clusterings.

Fig. 7.4 shows a mixture of state, architecture and deployment diagram based on this modularization. Vertices are either logic anchors (black), actions (yellow), important data (green) and databases (blue). Arcs denote logic flow (black) or data flow (blue).

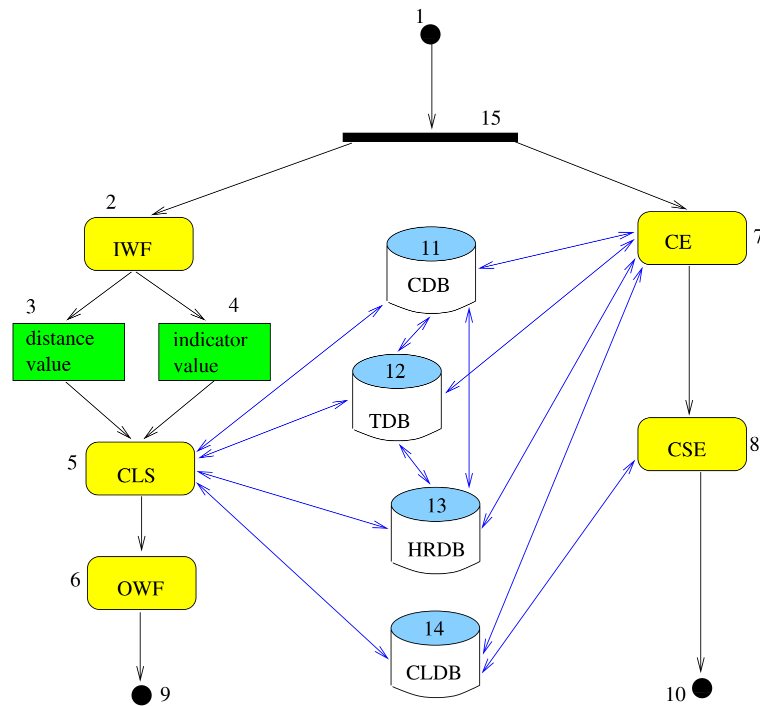


Figure 7.4: Initial diagram.

7.3.3.1.1 Interfacing An *interface* is a module whose only purpose is that of passing data between other module. Interfaces are useful to standardize function calls or message passing protocols. Functional/technical architectures may become entangled and "modularized" after prototype implementations have exhibited previously unsuspected module connection needs, resulting in architecture graph diagrams having many arcs/edges (connections) and relatively few nodes (modules). The *interfacing operator* is an automatic graph reformulation that adds interface modules in order to reduce the total number of connections.

Consider a digraph $G = (V, A)$ representing the existing architecture. We aim to construct a digraph $G' = (V', A')$ with fewer arcs but larger or equal transitive closure (i.e. no missing connection topology) by introducing interface modules. In our formalism, we represent the inter-modular connection/relation type as a colour on the corresponding arc.

Given an arc colouring $\mu : A \to \mathbb{N}$ of G and a connected subgraph $H = (U, F)$ of G such that $\mu(e) = \mu(f)$ for all $e, f \in F$, G' is defined as G with the subgraph H replaced by a subgraph $H' = (U', F')$ where $U' = U \cup \{\iota\}$; $(u, \iota), (\iota, v) \in F'$ if and only if $(u, v) \in F$. The aim of this reformulation is to simplify a set of interconnections of the same type. In the extreme case where H is a complete subgraph, the reformulation replaces it with a complete star around ι, which reduces the number of interconnections by a factor $|U|$. Naturally, in order for this reformulation to be worthwhile, we require $|F'| < |F|$. As $|F'|$ is bounded above by $2|U|$, we want to find a (not necessarily induced) subgraph $H = (U, F)$ of G whose arcs have the same colour and such that $|F| - |U|$ is maximum. We formulate and solve this problem using mathematical programming.

Let $\{1, \dots, K\}$ be the set of arc colours in G. For all $v \in V$ consider binary variables $x_v = 1$ if $v \in U$ and 0 otherwise. For any colour $k \leq K$, the problem of finding the "densest" uniformly coloured

subgraph $H_k = (U, F)$ of G can be formulated as follows.

$$\max_{x,y} \sum_{(u,v)\in A_{i-1}} x_u x_v - \sum_{v\in V} x_v \tag{7.1}$$

$$\forall\, (u,v) \in A \quad x_u x_v \;\le\; \min(\max(0, \mu_{uv} - k + 1), \max(0, k - \mu_{uv} + 1)) \tag{7.2}$$

$$x \;\in\; \{0,1\}^{|V|}. \tag{7.3}$$

The interfacing operator is implemented by algorithmically providing a solution to (7.1)-(7.3). We apply the interfacing operator to this graph on the blue arc color (data flow arcs, coded by the label 2 in the AMPL data file). The AMPL model file is as follows.

```
# interface.mod
# AMPL model for interface creation

# graph
param n >= 1, integer;
set V := 1..n;
set E within {V,V};

# edge weights
param c{E};

# edge inclusions
param I{E};

# vertex colours
param lambda{V};

# arc colours
param kmax default 10;
param k <= kmax, >= 0, integer, default 1;
param mu{E} >=0, integer, <= kmax;

# variables
var x{V} binary;
var y{(u,v) in E} >= 0, <= min(max(0, mu[u,v]-k+1), max(0,k-mu[u,v]+1));

# model

maximize densesubgraph : sum{(u,v) in E} I[u,v] * c[u,v] * y[u,v] -
                 sum{v in V} x[v];

# linearization constraints
subject to lin1 {(u,v) in E} : y[u,v] <= x[u];
subject to lin2 {(u,v) in E} : y[u,v] <= x[v];
subject to lin3 {(u,v) in E} : y[u,v] >= x[u] + x[v] - 1;
```

The AMPL data file is as follows.

```
# activity1.dat
# AMPL dat file from UML activity diagram 1

param n := 15;
param : E :   c I mu :=
       1  15  1 1 1
       2  15  1 1 1
       2   3  1 1 1
       2   4  1 1 1
       3   5  1 1 1
       4   5  1 1 1
       5   6  1 1 1
       5  11  1 1 2
       5  12  1 1 2
       5  13  1 1 2
       5  14  1 1 2
       6   9  1 1 1
       7   8  1 1 1
       7  11  1 1 2
       7  12  1 1 2
       7  13  1 1 2
       7  14  1 1 2
       7  15  1 1 1
```

```
       8  10  1 1 1
       8  14  1 1 2
       11 12  1 1 2
       11 13  1 1 2
       12 13  1 1 2
;

param lambda :=
1  1
2  2
3  3
4  3
5  2
6  2
7  2
8  2
9  1
10 1
11 4
12 4
13 4
14 4
15 1 ;
```

The AMPL run file is as follows.

```
# file interface.run
model interface.mod;
data activity1.dat;
let k := 2; # choose the colour
option solver cplexstudent;
solve;
display y;
display x;
```

We solve the problem by issuing the command `cat interface.run | ampl`. We find the interface subgraph $H = (U, F)$ where $U = \{5, 7, 11, 12, 13, 14\}$ and $F = \{\text{all blue arcs}\}$. We add a new vertex 16 representing the interface, remove the arcs F and add the (bidirected) arcs $F' = \{\{u, 16\} \mid u \in U\}$. Since 16 is a database interface, we assign it the database vertex colour (blue). The diagram evolves into Fig. 7.5.

7.3.3.1.2 Synthesis Modules in a software architecture need to be clustered for at least two good reasons: (a) to give an idea of the different independent (or nearly independent) “streams” in the architecture; (b) to be able to assign separate sets of modules to separate teams.

One of the most common ways to bootstrap a software architecture design process is to construct the initial graph G_0 by means of a brainstorming session: this will almost always give rise to a very “tangled” architecture. Modules will roughly correspond to the requirements list, and will be heavily interconnected. Clustering these modules in an arbitrary way according to their perceived semantics may give rise to clusters whose degree of inter-dependency is not minimal, which will greatly complicate team interactions and possibly impair the whole development process.

In its most basic form, the clustering procedure acts on a weighted, undirected graph $G = (V, E, c)$ (where $w : E \to \mathbb{R}$) and outputs an assignment of vertices in V to a set of clusters such that the weights of inter-cluster edges is minimized. Such a problem is known in the combinatorial optimization literature as the GRAPH PARTITIONING PROBLEM (GPP) [1, 9, 4, 2].

Its formulation in terms of mathematical programming is as as follows: given the weighted undirected graph G and an integer $K \le |V|$, the problem consists of finding a partition of k subsets (clusters) of V minimizing the total weight of edges $\{u, v\}$ where u, v belong to different clusters. To each vertex $v \in V$ and for each cluster $k \le K$, we associate a binary variable x_{vk} which is 1 if vertex v is in cluster h and 0

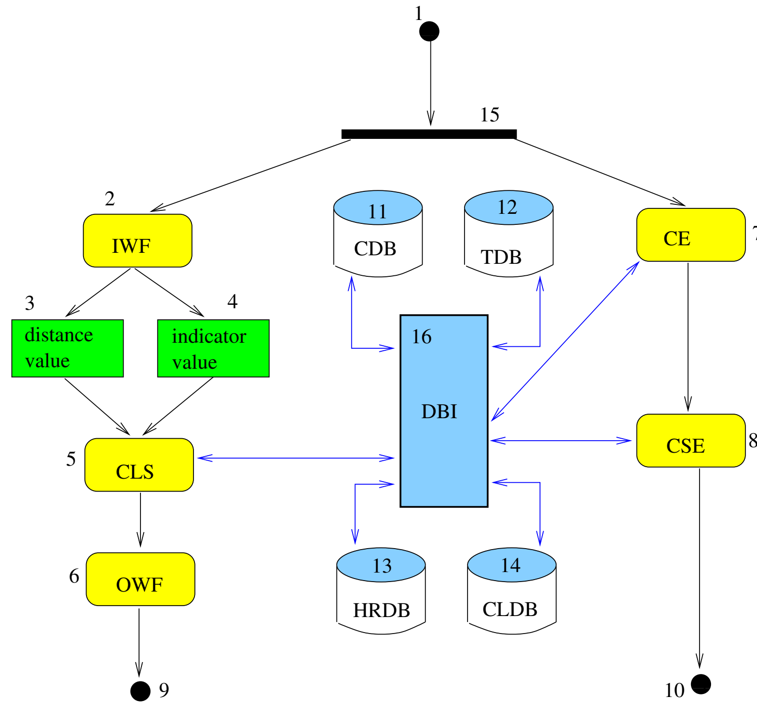


Figure 7.5: Diagram after interfacing.

otherwise. We formulate the problem as follows:

$$\min_x \frac{1}{2} \sum_{k\neq l \leq K} \sum_{\{u,v\}\in E} c_{uv} x_{uk} x_{vl} \tag{7.4}$$

$$\forall\, v \in V \quad \sum_{k\leq K} x_{vk} = 1 \tag{7.5}$$

$$\forall\, k \leq K \quad \sum_{v\in V} x_{vk} \geq 1 \tag{7.6}$$

$$\forall\, v \in V, k \leq K \quad x_{vk} \in \{0,1\}. \tag{7.7}$$

This model relies on binary variables and includes many (nonconvex) quadratic terms. Various ways to linearize the formulation have been suggested [3, 10]. The objective function (7.4) tends to minimize the total weight of edges with adjacent vertices in different clusters. Constraints (7.5) make sure that each vertex is assigned to exactly one cluster. Constraints (7.6) excludes the trivial solution (all the vertices in one cluster) and ensures each cluster exists. Further conditions, such as the clusters not exceeding a "balanced" cardinality, may also be imposed:

$$\forall\, k \leq K \quad \sum_{v\in V} x_{vk} \leq \left\lceil \frac{|V|}{2} \right\rceil. \tag{7.8}$$

A useful variant of the problem asks for all adjacent vertices with like colours to be clustered together. The vertex colours are defined by an integer-valued function $\lambda : V \to \mathbb{N}$ (we denote $\lambda(u)$ by λ_u). For all $u, v \in V$ (with $u \neq v$) we introduce binary parameters $\gamma_{uv} = 1$ if u, v have different colours. We must then add the following constraints:

$$\forall\, u \neq v \in V, k \neq l \leq K \;\; x_{uk} x_{vl} \leq \gamma_{uv}. \tag{7.9}$$

Should these constraints be too restrictive and make it too difficult for the solution algorithm to actually find a solution, we may want to relax them somewhat. We can do this by removing (7.9) and adding the term $\sum_{u\neq v\in V}\sum_{k\neq l\leq K}|x_{uk}x_{vl}-\gamma_{uv}|$ to the objective function (7.4).

Another useful variant allows the optimization process to determine the number of clusters actually used. For all $k\leq K$ we introduce binary variables $z_k=1$ if cluster k is non-empty and 0 otherwise. We change constraints (7.6) to

$$\forall\, k\leq K \quad \sum_{v\in V} x_{vk}\geq z_k, \tag{7.10}$$

to ensure that a cluster that does not exist need not have any vertices assigned to it, and then we add the term $\sum_{k\leq K} z_k$ to the objective function to be minimized, thus ensuring that the maximum number of clusters should be empty.

Once the set of clusters $\mathcal{K}$ have been identified, the current graph G may be replaced by a graph $G'=(V',A')$ where V' is the set of clusters $\mathcal{K}$ and $(u,v)\in A'$ if there are $w\in u, z\in v$ (recall u,v are subsets of V) such that $(w,u)\in A$. The *synthesis operator* performs such a reformulation to the architecture diagram.

We now apply the synthesis operator to the architecture, in order to identify some clusters with a small number of interconnections. Such clusters may help break down the architecture in logically disconnected parts; as system-wide faults usually emerge from inter-team lack of communication, assigning such parts to different teams will minimize the chances of ending up with system-wide faults.

The AMPL model is as follows.

```
# flexcolour_clustering.mod
# flexible coloured clustering (colours on vertices) - AMPL model

# graph
param n >= 1, integer;
set V := 1..n;
set E within {V,V};

# edge weights
param c{E};

# edge inclusions
param I{E};

# vertex colours
param lambda{V};
param gamma{u in V, v in V : u != v} :=
  if (lambda[u] = lambda[v]) then 0 else 1;

# arc colours
param mu{E};

# max number of clusters
param kmax default n;
set K := 1..kmax;

# original problem variables
var x{V,K} binary;

# linearization variables
var w{V,K,V,K} >= 0, <= 1;

# cluster existence variables
var z{K} binary;

# model
minimize intercluster :
  sum{k in K, l in K, (u,v) in E : k != l} I[u,v] * c[u,v] * w[u,k,v,l] +
  sum{k in K} z[k];

subject to assignment {v in V} : sum{k in K} x[v,k] = 1;

# use (ceil(card{V}/kmax)+1) as RHS for balanced multi-cluster cardinality
subject to cardinality {k in K} : sum{v in V} x[v,k] <= ceil(card{V}/2);
```

```
subject to existence {k in K} : sum{v in V} x[v,k] >= z[k];

subject to diffcolours {u in V, v in V, k in K, l in K : u != v and k != l} :
  w[u,k,v,l] <= gamma[u,v];

### linearization constraints
subject to lin1 {u in V, v in V, h in K, k in K : (u,v) in E or (v,u) in E} :
  w[u,h,v,k] <= x[u,h];
subject to lin2 {u in V, v in V, h in K, k in K : (u,v) in E or (v,u) in E} :
  w[u,h,v,k] <= x[v,k];
subject to lin3 {u in V, v in V, h in K, k in K : (u,v) in E or (v,u) in E} :
  w[u,h,v,k] >= x[u,h] + x[v,k] - 1;
```

The AMPL data file is as follows.

```
# activity2.dat
# AMPL dat file from UML activity diagram 2

param n := 16;
param : E :   c I mu:=
       1  15  1 1 1
       2  15  1 1 1
       2   3  1 1 1
       2   4  1 1 1
       3   5  1 1 1
       4   5  1 1 1
       5   6  1 1 1
       5  16  1 1 2
       6   9  1 1 1
       7   8  1 1 1
       7  16  1 1 2
       8  10  1 1 1
       8  16  1 1 2
       11 16  1 1 2
       12 16  1 1 2
       13 16  1 1 2
       14 16  1 1 2
;

param lambda :=
1  1
2  2
3  3
4  3
5  2
6  2
7  2
8  2
9  1
10 1
11 4
12 4
13 4
14 4
15 1
16 4 ;
```

The AMPL run file is as follows.

```
# file flexcolour_clustering.run
model flexcolour_clustering.mod;
data activity1.dat;
let kmax := 4; # maximum number of clusters
option solver cplexstudent;
solve;
display y;
display x;
```

We solve the problem by issuing the command `cat flexcolour_clustering.run | ampl`. We ask for at most 4 clusters (`let kmax := 4;`). We obtain two clusters: $C_1 = \{1, 2, 3, 4, 5, 6, 9, 15\}$ and $C_2 = \{7, 8, 10, 11, 12, 13, 14, 16\}$. The diagram is now as in Fig. 7.6.

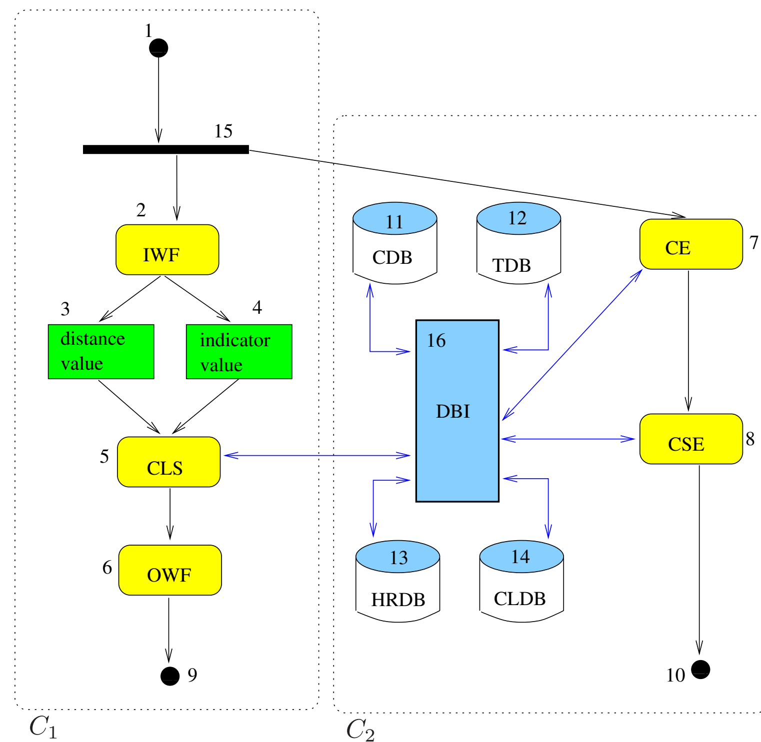


Figure 7.6: Diagram after clustering.

The architecture is composed of two main subsystems C_1, C_2, corresponding to two activity processes IWF→CLS→OWF (performed by the user) and CE↔CSE↔DBI (performed by the program) that we may call respectively the *foreground* and *background* processes. The foreground subsystem consists of three main components (input, classifier and output); the background subsystem consists of a database sub-subsystem (with an interface and four databases) and two main components (clustering engine and clustering significance evaluator). Very high-level specifications may now be given as follows:

1. IWF: input indicator(s) and clustering distance(s) from the user
2. CLS: classify new project according to given indicator(s) and distance(s) using a database of existing clusterings with cluster-matching information
3. OWF: output set of existing projects close to the new project w.r.t. given indicator(s)
4. CE: given a set of indicator values and an associated distance metric, cluster the values; pass the clustering to the DB interface for storage
5. CSE: given two clusterings, match them and verify their compatibility; pass the matching information to the DB interface for storage
6. DBI: interface to customer and clustering DBs.

The two processes (corresponding to C_1, C_2) are linked by arcs $(15, 7)$ (a logical flow arc) and $(5, 16)$ (a data flow arc). The logical path choices $(1, 15, 2)$ and $(1, 15, 7)$ identify the foreground and background processes respectively. If we consider two separate starting points for the two processes we can eliminate vertex 15 and all its adjacent arcs (including $(15, 7)$). We then introduce a starting vertex (labelled 15, since the old vertex 15 was reformulated out of the graph) for the background process (see Fig. 7.7). The data flow arc $(5, 16)$ is crucial to the process interplay and cannot be eliminated. It actually gives

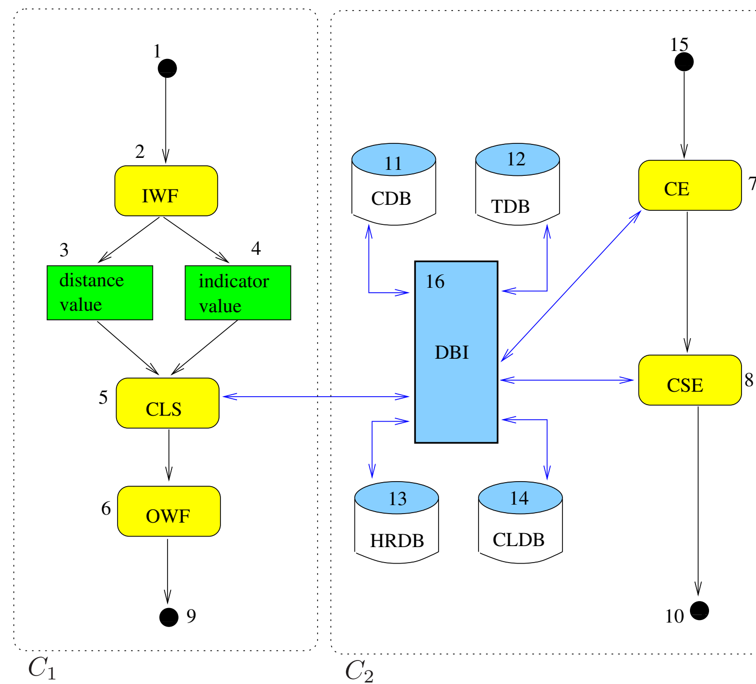


Figure 7.7: Final diagram: separating the processes.

the extent and the type of interconnection between the processes. It also suggests where the two teams developing the different process will need to interact, namely in the design of the database interface (DBI, vertex 16): more precisely, the background process team will need to explain to the other team what data is made available by the interface, and the foreground process team will need to require the appropriate data exchange formats and protocols.

The precise breakdown of each component into classes and methods is part of the technical architecture.

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).

7.3.4.1 Solution

In order to build a class diagram and the APIs, we need to know how input data are transformed into the output data, and exactly which data is passed from one component to another. As the background process is in some way a server to the foreground one, we shall model the latter first (top-down approach).

Informally, the data flow for the foreground process is as follows:

$$\begin{array}{ll}
\textbf{input} \stackrel{\text{IWF}}{\longrightarrow} & \text{(early indicator, indicator value, distance value)} \stackrel{\text{CLDB}}{\longrightarrow} \\
\rightarrow & \text{(corresponding early clustering, cluster)} \stackrel{\text{CLS}}{\longrightarrow} \\
\rightarrow & \text{(matching late clustering(s), cluster(s))} \stackrel{\text{OWF}}{\longrightarrow} \\
\rightarrow & \textbf{output} \text{ projects in identified cluster(s).}
\end{array}$$

In order for the foreground process data flow to make any sense, the CLDB database must contain all the clusterings relative to the given early indicator value and distance, and the CLS component must be able to match early and late clusterings (and to draw the appropriate "close" clusters from the matched clusterings). The background process must therefore supply the necessary information. Recall that the foreground process is ran by each user, and so should be as fast as possible. It is therefore necessary to delegate most of the computational work to the background process: all data transformation should draw from information that was pre-computed by the background process. In particular, finding a matching late clustering should be as simple as looking up a pre-computed boolean value in an array; in turn, this means that the background process must pre-compute all possible matching information and store it in the CLDB database.

Informally, the data flow for the background process is as follows:

$$\begin{array}{lll}
\textbf{start} & \rightarrow & \text{all possible pairs } ((\text{early indicator}, \text{distance}), (\text{late indicator}, \text{distance})) \stackrel{\text{CE}}{\longrightarrow} \\
& \rightarrow & \text{(clustering)} \stackrel{\text{DBI}}{\longrightarrow} \\
& \rightarrow & \textbf{store} \text{ (clustering)} \stackrel{\text{CSE}}{\longrightarrow} \\
& \rightarrow & (\text{do clustering match?}, \text{list of matching clusters}) \stackrel{\text{DBI}}{\longrightarrow} \\
& \rightarrow & \textbf{store} \text{ (matching info)} \rightarrow \textbf{stop}.
\end{array}$$

To make the data flow descriptions more formal, we must make clear what we mean precisely by such concepts as indicator, distance, cluster, clustering, clustering comparison.

7.3.4.1.1 Indicators An *indicator* is a non-negative real-valued function $v : P \rightarrow \mathbb{R}_+$ defined on the set of projects P. Given an indicator v, we let:

$$\begin{aligned}
\bar{v} &= \max_{p \in P} v(p) \\
\underline{v} &= \min_{p \in P} v(p).
\end{aligned}$$

Early indicators are indicators whose value can be defined before the project is started; *late* indicators may only be defined after the project ends. Consider early indicators v_i^E for $i \leq m$ and late indicators v_j^L for $j \leq n$. For each early indicator $i \leq m$ we also consider finite sets of distances[1] D_i^E with (and likewise for late indicators).

7.3.4.1.2 Clusterings Given an indicator v on P and a distance value $d \in \mathbb{R}_+$, a *clustering* γ_{vd} of P is a set of $K_{vd} = |\gamma_{vd}|$ subsets γ_{vdk} of P, where $k \leq K_{vd}$ and

$$K_{vd} = \left\lceil \frac{\bar{v}_i - \underline{v}_i}{d} \right\rceil,$$

such that:

[1]By *distance* we mean here a generic measure of similarity, without implying the triangular inequality.

(a) $\forall p \in P\ \exists k \le K_{vd}\ (p \in \gamma_{vdk})$ (covering condition).

(b) $\forall k \le K_{vd}\ \forall p \in \gamma_{vdk}\ (\underline{v}_i + kd \le v(p) \le \underline{v}_i + (k+1)d)$ (cluster extent).

Notice we define clusterings so that γ_{vd} is unique for each choice of v, d and can be computed in $O(|P|)$. This is not the only possible such definitions. Other definitions allow for non-uniqueness and for higher computational complexity orders.

Each subset γ_{vdk} of a clustering is called a *cluster*; because of (b), we can assign to each cluster γ_{vdk} an interval $I_{vdk} = [\underline{v}_i + kd, \underline{v}_i + (k+1)d]$. Let γ^E_{id} be the clustering of P corresponding to the early indicator v^E_i and distance $d \in D^E_i$, with $K_{id} = |\gamma^E_{id}|$; let γ^E_{idk} be the k-th cluster of γ^E_{id} for $k \le K_{id}$ (and likewise for late indicators). Fig. 7.8 shows an example of a clustering where $P = \{1, \dots, 10\}, \underline{v} = 0, \bar{v} = 3, d = 1$.

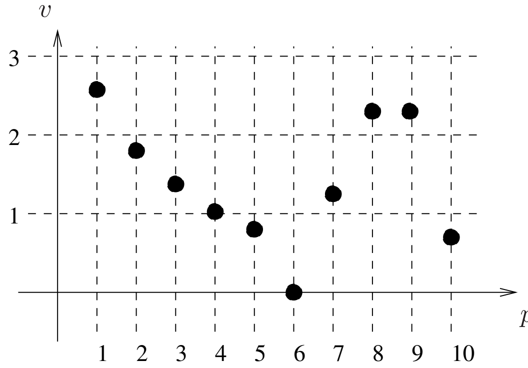


Figure 7.8: Clustering example. We obtain three clusters $\gamma_{vd1} = \{4, 5, 6, 10\}$ with $I_{vd1} = [0, 1]$, $\gamma_{vd2} = \{2, 3, 4, 7\}$ with $I_{vd2} = [1, 2]$ and $\gamma_{vd3} = \{1, 8, 9\}$ with $I_{vd3} = [2, 3]$.

7.3.4.1.3 Clustering comparison Our software relies on our ability to succesfully compare early and late clusterings and say if they match or not. Given two indicators u, v and distances d, δ with relative clusterings γ_{ud} and $\gamma_{v\delta}$, we first scale the indicator values and distances so that they are comparable. This can be easily done by scaling the two clustering intervals I_{ud} and $I_{v\delta}$ to the interval $[0, 1]$: for all $p \in P$ let

$$\begin{aligned}
\tilde{u}(p) &= \frac{u(p) - \underline{u}(p)}{\bar{u}(p) - \underline{u}(p)} \\
\tilde{v}(p) &= \frac{v(p) - \underline{v}(p)}{\bar{v}(p) - \underline{v}(p)} \\
\tilde{d} &= \frac{d - \underline{u}(p)}{\bar{u}(p) - \underline{u}(p)} \\
\tilde{\delta} &= \frac{\delta - \underline{v}(p)}{\bar{v}(p) - \underline{v}(p)}.
\end{aligned}$$

We define the dissimilarity between the two clusterings $\gamma_{ud}, \gamma_{v\delta}$ as:

$$\Delta(\gamma_{ud}, \gamma_{v\delta}) = (\tilde{d} - \tilde{\delta})^2 + \sum_{p \in P} (\tilde{u}(p) - \tilde{v}(p))^2.$$

Notice this definition does not actually consider the clustering itself, but just the indicator and the distance: this occurs because of the way our clusterings are defined. More precisely, this occurs because the cluster each $p \in P$ belongs to is determined by p alone and not by the other elements of P.

Given an overall tolerance $\varepsilon > 0$, an early indicator clustering γ^E_{id} (where $i \le m$ and $d \in D^E_i$) *matches* a late indicator clustering $\gamma^L_{j\delta}$ (where $j \le n$ and $\delta \in D^L_j$) if either one of the two conditions below is satisfied:

1. $\Delta(\gamma^E_{id}, \gamma^L_{j\delta}) \le \varepsilon$;
2. $\Delta(\gamma^E_{id}, \gamma^L_{j\delta}) = \min\limits_{h\le n, b\in D^L_h} \Delta(\gamma^E_{id}, \gamma^L_{hb})$;

we denote the matching by $M(\gamma^E_{id}, \gamma^L_{j\delta}) = 1$ and a mismatch by $M(\gamma^E_{id}, \gamma^L_{j\delta}) = 0$. If two matching clusterings satisfy the first condition, it is a close match. The second condition is a "catch-all" condition which ensures that we can match each early indicator clustering to at least one late indicator clustering.

Given two matching clusterings $\gamma_{ud}, \gamma_{v\delta}$, we must now find indices $h \le K_{ud}$ and $k \le K_{v\delta}$ such that γ_{udh} and $\gamma_{v\delta k}$ are "as close as possible". We extend the dissimilarity definition Δ to clusters as follows:

$$\Delta(\gamma_{udh}, \gamma_{v\delta k}) = (\tilde{d} - \tilde{\delta})^2 + \sum_{p\in\gamma_{udh}\cap\gamma_{v\delta k}} (\tilde{u}(p) - \tilde{v}(p))^2 + |\gamma_{udh} \triangle \gamma_{v\delta k}|,$$

where $A \triangle B = (A\cup B) \smallsetminus (A\cap B)$ is the symmetric difference of two sets A, B. This definition is justified by the fact that the difference in normalized indicator value for a project p in $\gamma_{udh}\triangle\gamma_{v\delta k}$ is simply the diameter of the corresponding normalized interval $[0,1]$, namely 1, and that $1^2 = 1$. With this extended definition, we can compute $\Delta(\gamma_{udh}, \gamma_{v\delta k})$ for each possible pair (h,k) and determine a pair of closest clusters. We denote the set of clusters $\gamma_{v\delta k}$ in $\gamma_{v\delta}$ closest to a given cluster γ_{udh} by $\Gamma(\gamma_{udh}, \gamma_{v\delta})$.

7.3.4.1.4 The foreground process The data transformation model of the foreground process is as follows: we are given a new project π; we select an early indicator v^E_i, compute $w = v^E_i(\pi)$, select a meaningful distance $d \in D^E_i$, find the corresponding clustering γ^E_{id} and the cluster γ^E_{idk} such that $w \in I_{idk}$. We then find a late indicator clustering $\gamma^L_{j\delta}$ (where $j \le n$ and $\delta \in D^L_j$) such that $M(\gamma^E_{id}, \gamma^L_{j\delta}) = 1$, and the corresponding closest clusters $\gamma^L_{j\delta h} \in \Gamma(\gamma^E_{idk}, \gamma^L_{j\delta})$. The formal data flow description of the foreground process is:

$$\begin{aligned}
\textbf{input} \stackrel{\text{IWF}}{\rightarrow}\ & (\pi, v^E_i, d) \stackrel{\text{CLDB}}{\longrightarrow} (\gamma^E_{id},\quad \gamma^E_{idk} : v^E_i(\pi) \in I_{idk}) \stackrel{\text{CLS}}{\longrightarrow} \\
\rightarrow\ & O = \{(j,\delta,h) \mid M(\gamma^E_{id}, \gamma^L_{j\delta}) = 1 \wedge \gamma^L_{j\delta h} \in \Gamma(\gamma^E_{idk}, \gamma^L_{j\delta}))\} \stackrel{\text{OWF}}{\longrightarrow} \\
\rightarrow\ & \forall (j,\delta,h) \in O\ \textbf{output}\text{ projects in } \gamma^L_{j\delta h}
\end{aligned}$$

The required data structures are:

- project (p, class): contains project attributes as defined in the T-Sale DB;
- cluster (list of projects);
- clustering (γ: list of clusters);
- indicator (v, class): contains
  - methods to retrieve the indicator value given a project
  - list of clustering distances D (floating point numbers)
  - extremal values $\bar{v}, \underline{v}$ (floating point numbers)
  - list of clusterings γ_{vd} for this indicator, relative to all distances $d \in D$
  - methods to scale the indicator values and distances in D to the interval $[0,1]$
- foreground process (class): contains
  - list of early indicators $(v^E_i \mid i \le m)$;
  - list of late indicators $(v^L_j \mid j \le n)$;
  - matching information (M, array of booleans indexed on $i \le m, d \in D^E_i, j \le n, \delta \in D^L_j$);
  - matching cluster information (Γ, maps clusters γ_{idk} for varying $k \le K_{id}$ to list of matching clusters $(\gamma_{j\delta h})$ for varying $h \in \{1,\ldots,K_{j\delta}\}$).

7.3.4.1.5 The background process The data transformation model of the background process is as follows: given an early indicator v_i^E $(i \le m)$, a distance $d \in D_i^E$, a late indicator v_j^L $(j \le n)$ and a distance $\delta \in D_j^L$:

- if γ_{id}^E is present in the CLDB database retrieve it, else compute it and store it;
- if $\gamma_{j\delta}^L$ is present in the CLDB database retrieve it, else compute it and store it.

Determine $M(\gamma_{id}^E, \gamma_{j\delta}^L)$ and store it in the CLDB database; if $M(\gamma_{id}^E, \gamma_{j\delta}^L) = 1$, for each $k \in K_{id}$ compute the set $\Gamma(\gamma_{idk}^E, \gamma_{j\delta}^L)$ and store it in the CLDB database. The formal data flow description of the background process is:

$$\begin{aligned}
\textbf{start} \quad &\rightarrow \quad \{((v_i^E, d), (v_j^L, \delta)) \mid i \le m \wedge d \in D_i^E \wedge j \le n \wedge \delta \in D_j^L\} \stackrel{\text{CE}}{\longrightarrow} \\
&\rightarrow \quad \mathcal{C} = \{(\gamma_{id}^E, \gamma_{j\delta}^L) \mid i \le m \wedge d \in D_i^E \wedge j \le n \wedge \delta \in D_j^L\} \stackrel{\text{DBI}}{\longrightarrow} \\
&\rightarrow \quad \textbf{store } \mathcal{C} \stackrel{\text{CSE}}{\longrightarrow} \\
&\rightarrow \quad \mathcal{M} = ((M(c) \mid c \in \mathcal{C}), (\Gamma(\gamma_{idk}^E, \gamma_{j\delta}^L) \mid (\gamma_{id}^E, \gamma_{j\delta}^L) \in \mathcal{C}, k \in K_{id})) \stackrel{\text{DBI}}{\longrightarrow} \\
&\rightarrow \quad \textbf{store } \mathcal{M} \rightarrow \textbf{stop}.
\end{aligned}$$

The required data structures are all those listed in Section 7.3.4.1.4 aside from the foreground process class, plus a background process class containing:

- list of early indicators $(v_i^E \mid i \le m)$
- list of late indicators $(v_j^L \mid j \le n)$
- matching information (M, array of booleans indexed on $i \le m, d \in D_i^E, j \le n, \delta \in D_j^L$)
- matching cluster information (Γ, maps clusters γ_{idk} for varying $k \le K_{id}$ to list of matching clusters $(\gamma_{j\delta h})$ for varying $h \in \{1, \ldots, K_{j\delta}\}$)
- methods for computing Δ applied to clusterings
- methods for computing intersections of clusters
- methods for computing symmetric differences of clusters
- methods for computing Δ applied to clusters.

7.3.4.1.6 Class structure The class structure is detailed in Fig. 7.9.

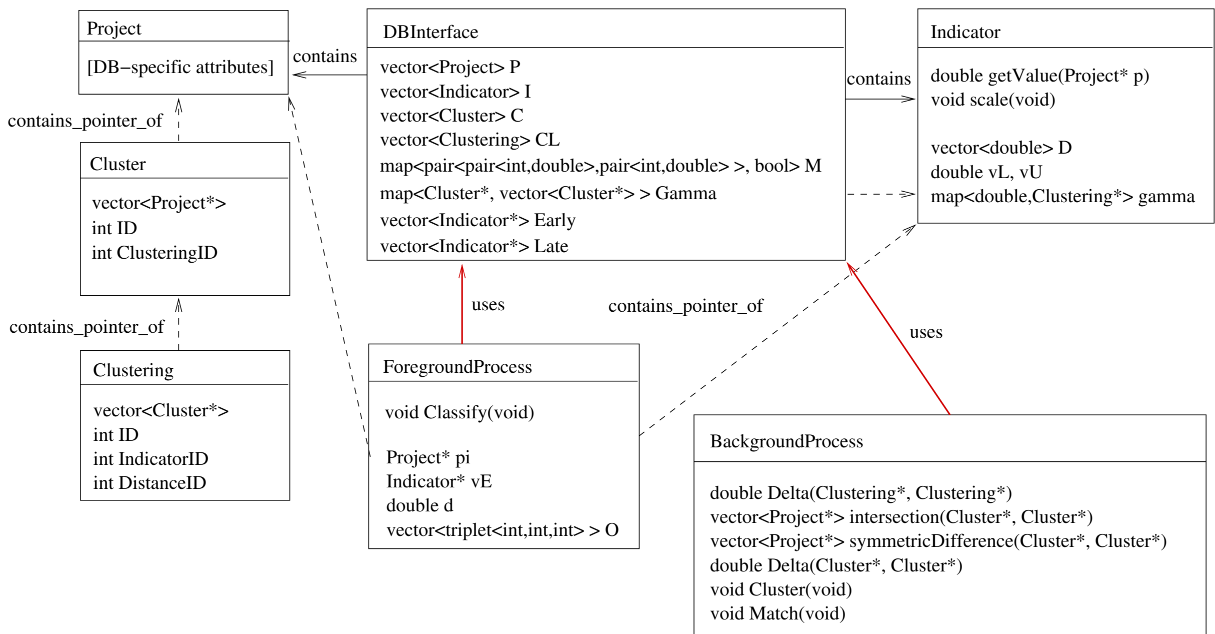


Figure 7.9: The class diagram of the fore- and background processes.

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.