Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology*

Jónathan Heras Vico Pascual Julio Rubio
Francis Sergeraert

{jonathan.heras, vico.pascual, julio.rubio}@unirioja.es,
francis.sergeraert@ujf-grenoble.fr

Abstract

Kenzo is a symbolic computation system devoted to Algebraic Topology. Written in Common Lisp, this program succeeded in computing homology and homotopy groups so far unreachable. The challenge is now to increase the number of users and to improve its usability. Instead of designing simply a friendly front-end, we have undertaken the task of devising a *mediated* access to the system, constraining its functionality, but providing guidance to the user in his navigation on the system. This objective is reached by constructing in Common Lisp an *intermediary layer*, allowing us an *intelligent* access to some features of the system. This intermediary layer is supported by XML technology and interplays between a graphical user interface and the *pure* Kenzo system.

1 Introduction

Kenzo [10] is a Common Lisp system, devoted to Symbolic Computation in Algebraic Topology. It was developed under the direction of the fourth author of this paper, and has been successful, in the sense that it has been capable of computing homology groups unreachable by any other means.

The main features of Kenzo as a Common Lisp system are: (1) the using of the Common Lisp Object System (CLOS) to organize a hierarchy of complex algebraic structures, and (2) the intensive use of higher-order functional programming, allowing us to represent and manipulate *infinite* spaces on a computer. Its power stems from an explicit link between (functional) infinite data structures and some finite counterparts. The first ones are used to encode the complex structures of Algebraic Topology; the second data (as lists, matrices, and the like) are used to compute effectively the invariants associated to the spaces.

Kenzo is in production since 1999. Having detected the accessibility and usability as two weak points in it (implying difficulties in increasing the number of users of the system), several proposals have been studied to interoperate with Kenzo (being the original user interface Common Lisp itself, the search for other

*Partially supported by Comunidad Autónoma de La Rioja, project Colabora2007/16, and Ministerio de Educación y Ciencia, project MTM2006-06513.

ways of interaction seems convenient to extend the use of the system). The aim of this paper is to present a report on our project for giving a new user interface to Kenzo.

Traditionally, symbolic computation systems, and Kenzo is no exception, have been oriented to research. This implies in particular, that development efforts in the area of Computer Algebra systems have been centered in aspects such as the improvement of the efficiency (or the accuracy, in symbolic-numerical systems) or the extension of the scope of the applications. Things are a bit different in the case of widely spread commercial systems such as Mathematica or Maple, where some attention is also payed to connectivity issues or to special-purpose user interfaces (usually related to educational applications). But even in these cases the central focus is on the results of the calculations and not on the interaction with other kind of (software or human) agents.

The situation is, in any sense, similar in the area of interoperability among symbolic computation systems (including here both computer algebra systems and proof assistants). The emphasis has been put in the *universality* of the middleware (see, for instance, [5]). Even if important advances have been achieved, severe problems have appeared, too, such as difficulties in reusing previous proposals and the final obstacle of the speculative existence of a *definitive mathematical interlingua*. The irruption of XML technologies (and, in our context, of MathML [2] and OpenMath [4]) has allowed standard knowledge management, but they are located at the *infrastructure* level, depending always on higher-level abstraction devices to put together different systems. Interestingly enough, the initiative SAGE [21] producing an integrated environment seems to have no use for XML standards, intercommunication being supported by ad-hoc SAGE mechanisms.

In summary, in the symbolic computation area, we are always looking for *more powerful* systems (with more computation capacities or with more general expressiveness). However, it is the case that our systems became so powerful, that we can lose some interesting kinds of users or interactions. We have encountered this situation when designing and developing the *TutorMates project* [13]. TutorMates is aimed at linking an educational front-end with the Maxima system [19]. Since the final users were students (and teachers) at the high school level it was clear from the beginning of the project that Maxima should be *weakened* in any sense, in order to make its outputs meaningful for non mathematics-trained users. This approach is now transferred to the field of symbolic computation in Algebraic Topology, where the Kenzo system [10] provides a complete set of calculation tools, which can be considered difficult to use by a non-Common Lisp trained user (typically, an Algebraic Topology student, teacher or researcher). The key concept is that of *mediated access* by means of an *intermediary layer* aimed at providing an *intelligent middleware* between a user interface and the kernel Kenzo system.

The paper is organized as follows. In the next section a short description of Kenzo as a Common Lisp system is presented. In Section 3 antecedents of our current project are commented, reporting on previous attemps to interoperate with Kenzo and on the TutorMates system. Section 4 gives some insights on methodological and architectural issues, both in the development of the client interface and in the general organization of the software systems involved. The central part of the paper can be found in Section 5, where the basics on the intermediary layer are explained. The concrete state of our project to interface

with Kenzo is the aim of Section 6. The paper ends with two sections devoted to open problems and conclusions, and finally the bibliography.

2 Kenzo as a Common Lisp system

The Kenzo program shows a concrete example of use of CLOS for a relative large implementation work (16000 Common Lisp lines and a 340pp documentation). It is the first significant *machine program* about classical Algebraic Topology. It is not only a program implementing various *known* algorithms; *new* methods have been developed to *transform* the main "tools" of Algebraic Topology, mainly the spectral sequences, not at all *algorithmic* in the traditional organization, into actual *computing* methods. With these "tools" the Kenzo program is able to produce mathematical results that are unreachable otherwise.

2.1 An example of Kenzo work.

Let us show a simple example to illustrate which is possible with this program. The homology group $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$[1] is "in principle" reachable thanks to old methods, see [6], but experience shows even the most skilful topologists meet some difficulties to determine it, see [18, 20]. With the Kenzo program, you construct the Moore space.

```
> (setf m4 (moore 2 4)) ✠
[K1 Simplicial-Set]
```

The program returns the Kenzo-object #1, a simplicial set, that is, a combinatorial version of the Moore space which is asked for, and this object is assigned to the symbol `m4`. Then you construct the third loop-space of this Moore space.

```
> (setf o3m4 (loop-space m4 3)) ✠
[K15 Simplicial-Group]
```

The combinatorial version of the loop space is *highly* infinite: it is a combinatorial version of the space of *continuous* maps $S^3 \to \text{Moore}(\mathbb{Z}_2, 4)$ but functionally coded as a small set of functions in a `simplicial-group` object, that is, a simplicial set with an added group structure compatible with the simplicial structure. Finally the fifth homology-group is asked for.

```
> (homology o3m4 5) ✠
Homology in dimension 5 :
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
---done---
```

and the result $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4) = \mathbb{Z}_2^5$ is obtained in some seconds in a standard PC. In natural situations a little more complicated, the Kenzo program has already computed new homology groups unreachable so far with "classical" Algebraic Topology, even from a theoretical point of view.

[1] The space $\text{Moore}(\mathbb{Z}_2, 4)$ is a "canonical" space having only non-trivial homology in dimension 4, namely $\mathbb{Z}_2$, and $\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$, its third loop space, is the space of continuous maps from the 3-sphere S^3 to this Moore space; the challenge is to determine the fifth homology group of this functional space.

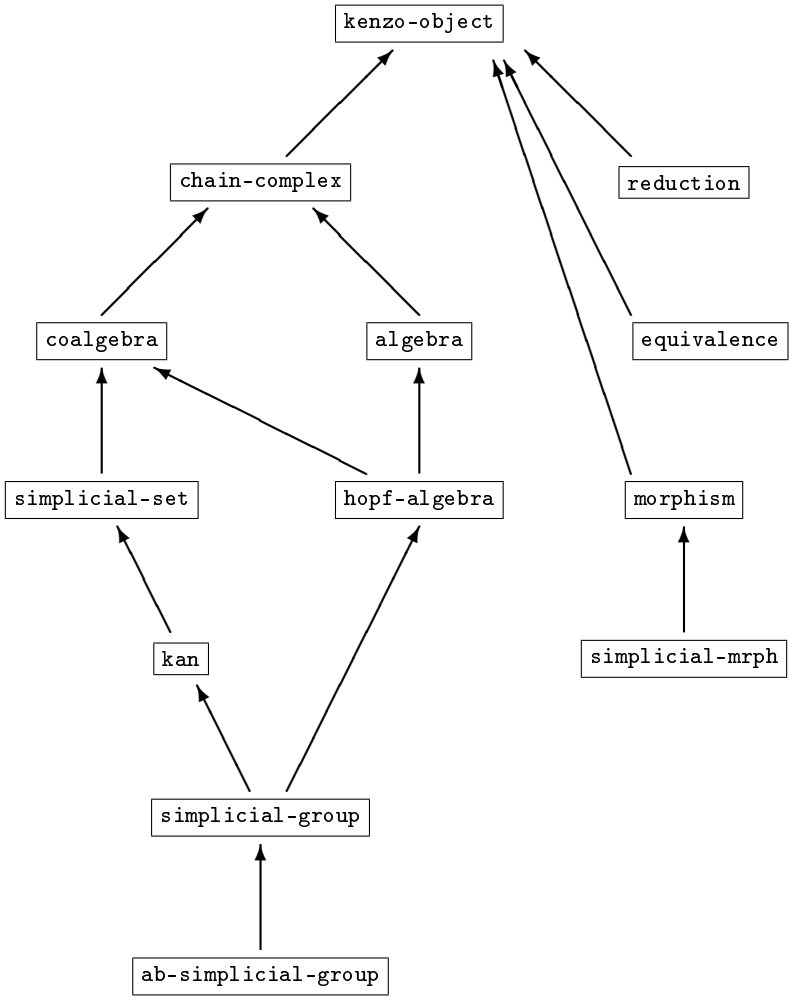


Figure 1: The Kenzo class diagram.

2.2 Kenzo classes.

Figure 1 shows the class diagram of Kenzo objects. The lefthand part of the class diagram is made of the main mathematical categories that are used in combinatorial Algebraic Topology. A *chain complex* is a graded differential module; an *algebra* is a chain complex with a compatible multiplicative structure, the same for a *coalgebra* but with a comultiplicative[2] structure. If a multiplicative and a comultiplicative structures are added and if they are compatible with each other in a natural sense, then it is a *Hopf algebra*, and so on.

The `hopf-algebra` and `simplicial-group` classes are typical cases where a *multi-heritage* situation is met; we show the *actual* Kenzo definitions of these classes.

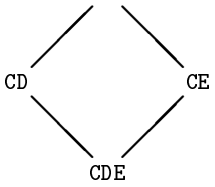
[2] That is, some *co*operator $A \to A \otimes A$.

```
(DEFCLASS HOPF-ALGEBRA (coalgebra algebra)
  ())

(DEFCLASS SIMPLICIAL-GROUP (kan hopf-algebra)
   ((grml :type simplicial-mrph :reader grml1)
    (grin :type simplicial-mrph :reader grin1)))
```

You see the definition of the `hopf-algebra` class is particularly striking; it explains that a Hopf-algebra is nothing but an algebra *and* a coalgebra; the compatibility conditions between both structures *cannot* be verified by a program and they necessarily depend on the programmer's "lucidity". In the same way, a *simplicial group* is a `kan` object and a `hopf-algebra` object sharing some common data, namely a coalgebra structure, with two further slots, `grml` (group multiplication) and `grin` (group inversion), those slots being some simplicial morphisms.

In such a multi-heritage situation, it is important the `call-next-method` function works as hoped-for. Look at this artificial situation just to show the process; the `C` class has two subclasses `CD` and `CE`, which have in common the subclass `CDE`; the artificial `initialize-instance` methods let you verify that `call-next-method` *remembers its story* when deciding what really the *next method* must be. Here, when processing the `CD`-level, `call-next-method` "remembers" the process was initiated from the `CDE`-level, so that the `CE`-level stage is not forgotten.




```
> (defclass C () ()) ✠
#<STANDARD-CLASS C>
> (defclass CD (C) ()) ✠
#<STANDARD-CLASS CD>
> (defclass CE (C) ()) ✠
#<STANDARD-CLASS CE>
> (defclass CDE (CD CE) ()) ✠
#<STANDARD-CLASS CDE>
> (defmethod initialize-instance ((c c) &rest rest)
    (print "C-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (C)>
> (defmethod initialize-instance ((cd cd) &rest rest)
    (print "beginning CD-initialization")
    (call-next-method)
    (print "finishing CD-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CD)>
> (defmethod initialize-instance ((ce ce) &rest rest)
    (print "beginning CE-initialization")
    (call-next-method)
    (print "finishing CE-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CE)>
> (defmethod initialize-instance ((cde cde) &rest rest)
    (print "beginning CDE-initialization")
    (call-next-method)
    (print "finishing CDE-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CDE)>
```

```
> (make-instance 'C) ✠
"C-initialization"
#<C @ #x212184da>
> (make-instance 'CD) ✠
"beginning CD-initialization"
"C-initialization"
"finishing CD-initialization"
#<CD @ #x21220e8a>
> (make-instance 'CE) ✠
"beginning CE-initialization"
"C-initialization"
"finishing CE-initialization"
#<CE @ #x2122698a>
> (make-instance 'CDE) ✠
"beginning CDE-initialization"
"beginning CD-initialization"
"beginning CE-initialization"  ←——←——←——!!!
"C-initialization"
"finishing CE-initialization"
"finishing CD-initialization"  ←——←——←——!!!
"finishing CDE-initialization"
#<CDE @ #x2122c03a>
```

And you may also play with the *auxiliary* `:before`, `:after` and `:around` methods to order as you like the various initialization steps. As a typical example, when the essential part of the initialization work of any `kenzo-object` is done, then the object is *finally* pushed in a list which is used later as explained in the next section. This is obtained as follows.

```
(DEFMETHOD INITIALIZE-INSTANCE :after ((k kenzo-object) &rest rest)
  (push k *k-list*))
```

In this way this is done if and only if the initialization work is successfully finished, even for the more specialized structures: if for example the specialized initialization work for a simplicial set fails and stops on error, then the pushing statement concerning the weakest structure is not run.

2.3 Optimizing computations.

The Kenzo program is certainly a *functional* system. It is frequent that several thousands of functions are present in memory, each one being *dynamically* defined from other ones, which in turn are defined from other ones, and so on. In this quite original situation, the same calculations are frequently *asked again*. To avoid repeating these calculations, it is better to store the results and to systematically examine for each calculation whether the result is already available (*memoization* strategy).

Because of this situation, it is very important not to have *several copies* of the same function; otherwise it is impossible for one copy to guess some calculation has already been done by another copy. This is a very important question in this program, so that the following idea has been used. Each Kenzo object has a rigorous *definition*, stored as a list in the `orgn` slot of the object (`orgn` stands for *origin* of the object). This is the main reason of the top class `kenzo-object`: making easier this process. The actual definition of the `kenzo-object` class:

```
(DEFCLASS KENZO-OBJECT ()
  ((idnm :type fixnum :reader idnm)
   (orgn :type list :reader orgn)
   (prpr :type list :reader prpr)
   (cmmn :type list :reader cmmn)))
```

Then, when any `kenzo-object` is to be considered, its *definition* is constructed and the program firstly looks in `*k-list*` whether some object corresponding to this definition already exists; if yes, no `kenzo-object` is constructed, the already existing one is simply returned. Look at this small example where we construct the second loop space of S^3, then the first loop space, and then again the second loop space. In fact the initial construction of the second loop space required the first loop space, and examining the identification number `K??` of these objects shows that when the first loop space is later asked for, Kenzo is able to return the already existing one.

```
> (setf s3 (sphere 3)) ✠
[K372 Simplicial-Set]
> (setf o2s3 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (setf os3 (loop-space s3 1)) ✠
[K374 Simplicial-Group]
> (setf o2s3-2 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (eq o2s3 o2s3-2) ✠
T
```

The last statement shows the symbols `o2s3` and `o2s3-2` points to the same machine address. In this way we are sure any kenzo-object has no duplicate, so that the memory process for the values of numerous functions cannot miss an already computed result. Let us look some `orgn` slots:

```
> (orgn o2s3) ✠
(LOOP-SPACE [K374 Simplicial-Group])
> (orgn (k 374)) ✠
(LOOP-SPACE [K372 Simplicial-Set])
> (orgn (k 372)) ✠
(SPHERE 3)
```

You see in this way the history of the construction process can be freely examined by the user, which is important in the development stage.

2.4 Delaying initializations.

The complete structure of a Kenzo object is extremely complicated, and many components are often useless. Another CLOS feature is therefore used to avoid the maybe non-necessary initialization works. The following artificial example explains how this is possible; it is a kind of *autoloading* mechanism, elegant, easy to be used, and useful to avoid initializing needless slots. We assume a `F` class, where each `F` object has two slots, `sl1` and `sl2`; the first one is necessary, but the second one would be the result of a *complex* process here simulated as being 1000 times the value of the first one.

```
> (DEFCLASS F ()
    ((sl1 :type integer :initarg :sl1 :reader sl1)
     (sl2 :type integer :reader sl2))) ✠
#<STANDARD-CLASS F>
```

```
> (DEFMETHOD SLOT-UNBOUND (class (fi f) (slot-name (eql 'sl2)))
    (declare (ignore class))
    (setf (slot-value fi 'sl2) (* 1000 (sl1 fi)))
    (sl2 fi)) ✠
#<STANDARD-METHOD SLOT-UNBOUND (T F (EQL SL2))>
> (SETF FI (make-instance 'f :sl1 23)) ✠
#<F @ #x213a7b8a>
> (SLOT-BOUNDP fi 'sl2) ✠
NIL
> (sl2 fi) ✠
23000
> (SLOT-BOUNDP fi 'sl2) ✠
T
```

You see the generic function `slot-unbound` is available which is called by the error manager when a non-initialized slot is asked for. The standard process finally does generate an error. But the user can write specialized methods for this generic function, allowing him instead to initialize the missing slot by some process using the available information. You see the initialization process lets uninitialized the `sl2` slot of the `F`-instance located by `fi`, but when this slot is asked for, the "right" value is in fact returned! A new examination by `slot-boundp` shows the slot is now bound.

This process is extremely convenient to organize the data as a living object where each time some missing component is questionned, an automatic "repairing process" is started, computing the missing information. The process may be recursive, so that if, in the repairing process, some other datum is again missing, an other repairing process is recursively started, and so on.

This possibility is intensively used in the Kenzo program. Look at this small experience. Firstly we reinitialize the environment by `cat-init`. When the fourth loop space $\Omega^4 S^5$ is constructed, you see only 26 Kenzo objects are present in the environment. Then the homology group $H_2\Omega^4 S^5$ is asked for. The answer, $\mathbb{Z}_2$ is quickly obtained, but the number of present Kenzo objects is now 504; an enormous set of `slot-unbound` calls has generated the construction of 478 new Kenzo objects, necessary to do the calculation. Furthermore a `:before` method had been added just to count the number of `slot-unbound` calls, a convenient debugging trick; you see the homology calculation has recursively generated 240 `slot-unbound` calls.

```
> (cat-init) ✠
---done---
> (setf s5 (sphere 5)) ✠
[K1 Simplicial-Set]
> (setf o4s5 (loop-space s5 4)) ✠
[K21 Simplicial-Group]
> (length *k-list*) ✠
26
> (setf counter 0) ✠
0
> (defmethod slot-unbound :before (class instance slot)
    (declare (ignore class instance slot))
    (incf counter)) ✠
#<STANDARD-METHOD SLOT-UNBOUND :BEFORE (T T T)>
> (homology o4s5 2) ✠
Homology in dimension 2 :
Component Z/2Z
---done---
```

```
> (length *k-list*) ✠
504
> counter ✠
240
```

2.5 Mixing low level and high level programming.

Computing time is crucial for the applications of the Kenzo program. The complexity of the implemented algorithms is highly exponential, so that the developer must carefully consider how he can improve the computing time of the written down Lisp code. In particular, if the heart of the program may be written close to the machine language, large amounts of computing time can be saved. But conversely this must not penalize the *readability* and the *modularity* of the program.

Which is striking with Common Lisp is the possibility of easily mixing *low level* and *high level* programming. The features about OOP show how Common Lisp is powerful in high level programming, allowing the user to directly handle the sophisticated objects of Algebraic Topology such as chain complexes, products and coproducts, Hopf algebras, simplicial sets and simplicial groups.

But on the other hand, the Kenzo program intensively uses the low level part of the Common Lisp language, that is, the quasi-assembler language which is the very root of the language, such as the popular `car`, `cdr`, and `cons`. This is possible thanks to the Common Lisp *macrogenerator*. Let us consider the case of the type `absm`, that is, *abstract simplex*. These objects are really the most elementary constituents of the Kenzo geometric objects, and they are so intensively used, billions of times for every significant Kenzo run, that you *must not* use CLOS for these kernel structures. Kenzo defines the `absm` type as follows:

```
(DEFUN ABSM-P (object)
   (declare (type any object))
   (the boolean
      (and (consp object)
           (eq :absm (car object))
           (typep (cdr object) 'iabsm))))

(DEFTYPE ABSM () '(satisfies absm-p))
```

The `absm-p` function explains an `absm` is a `cons` (pair) where the lefthand component is the keyword `:absm` and the righthand one is an `iabsm`, that is, an *internal* `absm`; in the same way, elsewhere in the program, it is explained an `iabsm` is again a `cons` where the righthand component is anything and the lefthand component is a fixnum coding a *degeneracy operator*. Most of computations in Algebraic Topology are in fact low level computations about degeneracy operators where such an operator is a decreasing list of small integers, like `(5 2 0)`; because this list is *strictly* decreasing, it can be represented by the fixnum 37 because $37 = 2^5 + 2^2 + 2^0$, so that all the standard calculations about degeneracy operators become fine calculations *at the bit level* on binary fixnums. But Common Lisp has all the predefined functions to do such a job, so that the programmer can efficiently work according to this strategy. A considerable memory space is saved so and furthermore the calculations are much faster.

If a degeneracy operator is to be extracted from an `absm`, the `dgop` macro is used:

```
> (DEFMACRO DGOP (absm)
   `(the dgop (cadr (the cadr ,absm))) ✠
DGOP
> (macroexpand '(dgop argument)) ✠
(THE DGOP (CADR (THE ABSM ARGUMENT)))
```

which explains that in fact the call of `dgop` is synonymous with a call of the assembler-like `cadr`, but the types of argument and result are verified:

```
> (dgop (absm 37 'something)) ✠
37
> (dgop 'not-an-absm) ✠
Error: object "NOT-AN-ABSM" is not of type "ABSM".
[condition type: PROGRAM-ERROR]
```

When the program is compiled, the compiler firstly translates the source code when a macro call is found, so that it is an assembler-like statement which is compiled; furthermore an appropriate compiler option allows the compiled code to ignore or not the type verifications through the '`the`' statements. When the program is finalized for production work, of course these type verifications are discarded to save computing time. You see in this way the Lisp code is *readable*, this code being firstly translated in low level Lisp statements, therefore very efficiently compiled, without loosing if necessary the type verifications.

3 Antecedents of our project

As explained in the Introduction, several proposals have been studied to interoperate with Kenzo. The most elaborated approach was reported in [1]. There, we devised a remote access to Kenzo, using CORBA [17] technology. An XML extension of MathML played a role there too, but just to give genericity to the connection (avoiding the definition in the CORBA Interface Description Language [17] of a different specification for each Kenzo class and datatype). There was no intention of taking profit from the semantics possibilities of MathML. Being useful, this approach ended in a prototype, and its enhancement and maintenance were difficult, due both to the low level characteristics of CORBA and to the pretentious aspiration of providing *full* access to Kenzo functionalities. We could classify the work of [1] in the same line as [5] or the initiative IAMC [15], where the emphasis is put into powerful and generic access to symbolic computation engines.

On the contrary, the TutorMates project [13] had, from its very beginning, a much more modest objective. The idea was to give access just to a part of Maxima, but guiding the user in his interaction. Since the purpose of TutorMates was educational (high school level), it was clear that many outputs given by Maxima were unsuitable for the final users, depending on the degree and the topic learned in each TutorMates session. To give just an example, an imaginary solution to a quadratic equation has meaning only in certain courses. In this way, a *mediated* access to Maxima was designed. The central concept is an intermediary layer that communicates, by means of an extension of XML, between the graphical user interface (Java based) and Maxima. The extension of MathML allows us to encode a *profile* for the interaction. A profile is composed of a role (student or teacher), a level and a lesson. In the case of a teacher (supposed to be preparing material for his students), full access to Maxima outputs is given, but a *warning* indicates to him whether the answer would be suitable

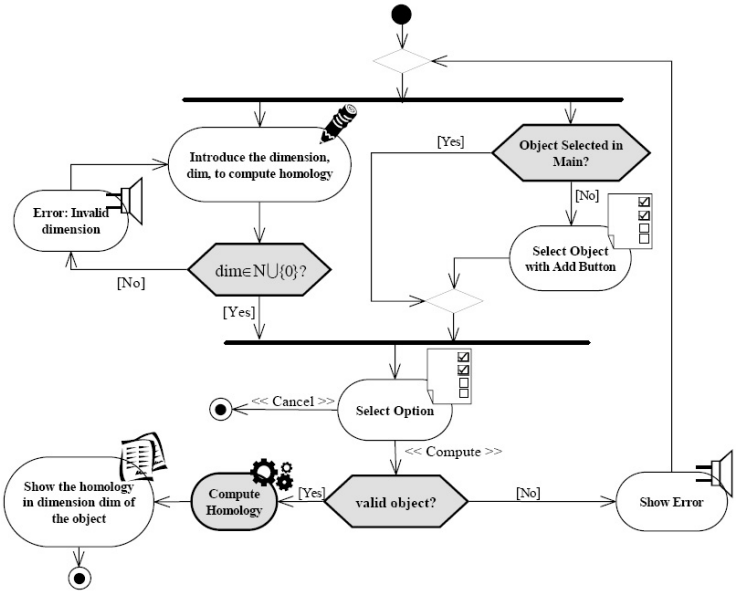


Figure 2: A fragment of the control and navigation graph.

inside the level and the lesson encoded in the profile. In this way, the intermediary layer allows the programmer to get an *intelligent* interaction, different from the "dummy" remote access obtained in [1].

Now, our objective is to emulate this TutorMates organization in the Kenzo context. The final users could be researchers in Algebraic Topology or students of this discipline. The problems to be tackled in the intermediary layer are different from those of TutorMates. The methodological and architectural aspects of this new product are presented in the following section.

4 Methodological and Architectural Issues

We have tried to guide our development with already proven methodologies and patterns. In the case of the design of the interaction with the user in our GUI front-end[3], we have followed the guidelines of the Noesis method [7]. In particular, our development has been supported by some Noesis models for control and navigation in user interfaces (see an example in Figure 2).

Even if graphical specification mechanisms have well-known problems (related with their scalability), Noesis models provide *modular tools*, allowing the designer to control the complexity due to the size of graphics. These models enable an exhaustive traversal of the interfaces, detecting errors, disconnected areas, lack of homogeneity, etc.

With respect to the general organization of the software system, we have been inspired by the *Microkernel* architectural pattern [3]. This pattern gives a global view as a *platform*, in terminology of [3], which implements a virtual

[3]The GUI has been implemented using the package *Common Graphics* and the *Integrated Development Environment* of Allegro Common Lisp [11].

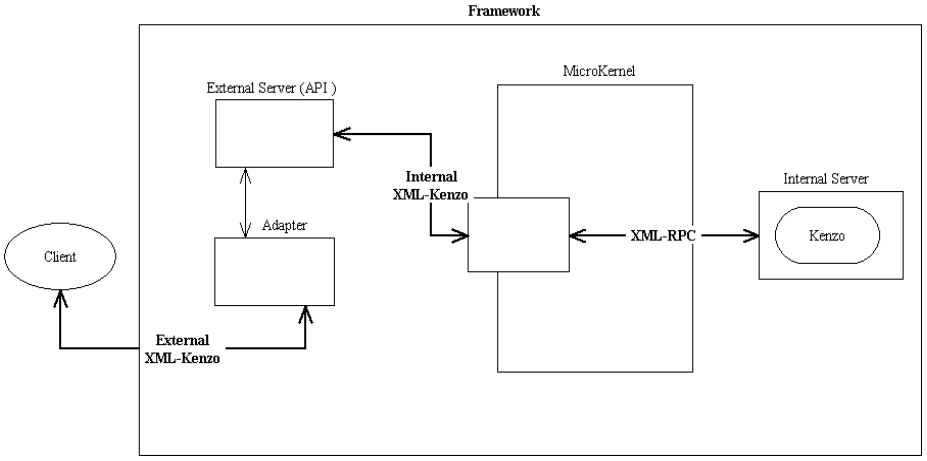


Figure 3: Microkernel architecture of the system.

machine with applications running on top of it, namely a *framework* (in the same terminology). A high level perspective of the system as a whole is shown in Figure 3. Kenzo itself, wrapped with an interface based on XML-RPC [22], is acting as *internal server*. The *microkernel* acting as intermediary layer is based on an XML processor, allowing both a link with the standard XML-RPC used by Allegro Common Lisp [11], and intelligent processing. The view of the *external server* is again based on an XML processor, with a higher level of abstraction (since mathematical knowledge is included there) which can map expressions from and to the microkernel, and which is decorated with an *adapter* (the *Proxy* pattern, [12], is used to implement the adapter), establishing the final connection with the client, a Graphical User Interface in our case. A simplified version of the Microkernel pattern (without the external server) would suffice if our objective was to build a GUI for Kenzo. But we also pursue extending Kenzo by wrapping it in a framework which will link any possible client (other GUIs, web applications, web services, ...) with the Kenzo system. In this sense, our GUI is a client of our framework. The framework should provide each client with all necessary mathematical knowledge.

Which aspects of the intelligent processing must be dealt with in the external server or in the microkernel, is still controversial (in the current version, as we will explain later, we have managed the questions related to the input specifications in the external server and the most important mediations are done at the microkernel level). Moreover, the convenience of a double level of processing is clear, being based on, at least, two reasons. On the one hand the more concrete one (microkernel) is to be linked to Kenzo (via XML-RPC) and the more abstract one is aimed at being exported and imported, rendered by (extended) MathML engines, and so on. On the other hand, this double level of abstraction reflects the different languages in which the knowledge has to be expressed. The external one is near to Algebraic Topology, and it should offer a communication based on the concepts of this discipline to the final clients (this gives a small type system; see Section 5). The internal part must communicate with Kenzo, and therefore a low level register of each session must be maintained (for instance, the unique identifier referring to each object, in order to avoid recalculations).

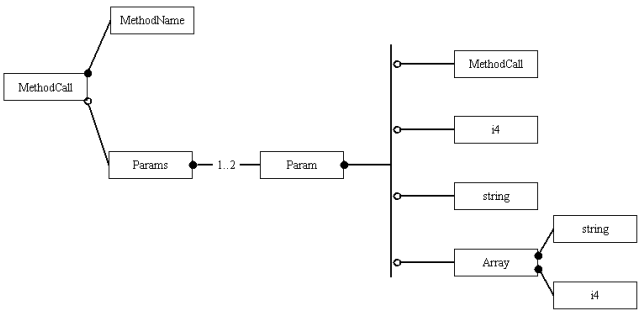


Figure 4: Description of the Internal XML Kenzo Schema.

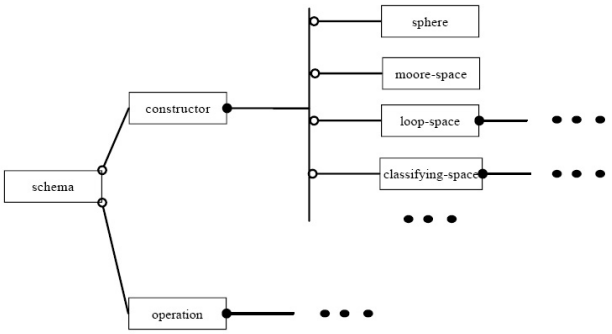


Figure 5: Fragment of the External XML Kenzo Schema.

There, a procedural language based on Kenzo conventions is needed.

As explained before, XML gives us the universal tool to transmit information along the different layers of the system. Besides the XML-RPC mechanism used by Allegro Common Lisp, two more XML formats (defined by means of XML schemas) are to be considered. The first one (used in the microkernel) is diagrammatically described in Figure 4, by using the Noesis method [9] again. The second format, used in the external server, will be (it is not completely defined yet) presented as an extension of the MathML schema [2]. Figure 5 shows a diagram corresponding to a part of this schema. The structure of this XML schema allows us to represent some knowledge on the process (for instance, it differentiates constructors from other kinds of algebraic manipulations); other more complex mathematical knowledge can not be represented in the syntax of the schema (see Section 5). In Figure 6, we show how a Kenzo command (namely, the calculation of the third group of homology of the sphere of dimension 3) will be transformed from the user command on the GUI (top part of the figure) to the final XML-RPC format (the conventional Lisp call is shown, too; however our internal server, Kenzo wrapped with an XML-RPC interface, will execute the command directly).

In the next section the behavior pursued with this architecture is explained.

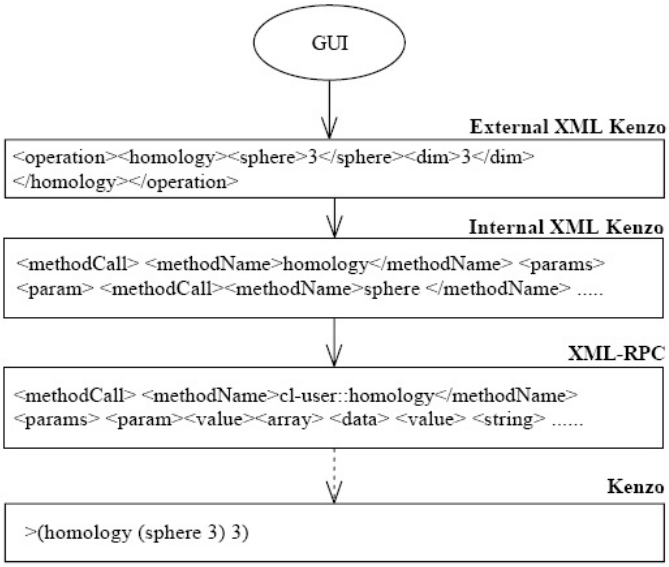


Figure 6: Transforming XML representations.

5 Knowledge Management in the Intermediary Layer

The system as a whole will improve Kenzo including the following "intelligent" enhancements:

1. Controlling the input specifications on constructors.
2. Avoiding some operations on objects which will raise errors.
3. Chaining methods in order to provide the user with new tools.
4. Determining if a calculation can be done in a local computer or should be derived to a remote server.

The first aspect is attained, in an integrated manner, inside the Graphical User Interface. The three last ones are dealt with in the intermediary layer. From another point of view, the first three items are already partially programmed in the current version of the system; the last one is further work.

In order to explain the differences between points 1 and 2, it is worth noting that in Kenzo there are two *kinds* of data. The first one is representing *spaces* in Algebraic Topology (by *spaces* we mean here, any data structure having both behavior and elements belonging to it, such as a simplicial set, a simplicial group, a chain complex, and so on). The second kind of data is used to represent *elements* of the spaces. Thus, in a typical session with Kenzo, the users proceed in two steps: first, constructing some spaces, and second, applying some operators on the (elements of the) spaces previously built. This organization in two steps has been described by using Algebraic Specification methods in [16] and [8], for instance. Therefore, the first item in the enumeration refers to the inputs for

the constructors of spaces, and the second item refers to some operations on *concrete* spaces. As we are going to explain, the first kind of control is naturally achieved in the GUI client (from the mathematical knowledge provided by the external XML format) but the second one, which needs some expert knowledge management, is better dealt with in the intermediary layer.

Kenzo is, in its pure mode, an untyped system (or rather, a dynamically typed system), inheriting its power and its weakness from Common Lisp. Thus, for instance, in Kenzo a user could apply a constructor to an object without satisfying its input specification. For example, the method constructing the classifying space of a simplicial group could be called on a simplicial set without a group structure over it. Then, at runtime, Common Lisp would raise an error informing the user of this restriction. This is shown in the following fragment of a Kenzo session:

```
> (loop-space (sphere 4)) ✠
[K6 Simplicial-Group]
> (classifying-space (loop-space (sphere 4))) ✠
[K18 Simplicial-Set]
> (sphere 4) ✠
[K1 Simplicial-Set]
> (classifying-space (sphere 4)) ✠
;; Error: No method in generic function CLASSIFYING-SPACE
;; is applicable to arguments: [K1 Simplicial-Set]
```

With the first command, namely `(loop-space (sphere 4))`, we construct a simplicial group. Then, in the next step we are verifying that a simplicial group has a classifying space (which is, in general, just a simplicial set). In the third command, we check that the sphere of dimension 4 is constructed in Kenzo as a simplicial set. Thus, when in the last command we try to construct the classifying space of a simplicial set, the Common Lisp Object System (CLOS) raises an error.

In the current version of our system this kind of error is controlled, because the inputs for the operations between spaces can be only selected among the spaces with suitable characteristics. The equivalent in our system of the example introduced before in pure Kenzo, is shown in Figure 7, where it can be seen that for the classifying operation just the spaces which are simplicial groups are candidates to be selected. This enriches Kenzo with a small (semantical) type system which will be defined into the external XML schema.

With respect to the second item in the previous enumeration, the most important example in the current version is the management of the *connection degree* of spaces. Kenzo allows the user to construct, for instance, the loop space of a non simply connected space (as the sphere of dimension 1). The result is a simplicial set on which some operations (for instance, to compute the set of faces of a simplex) can be achieved without any problems. On the contrary, theoretical results ensure that the homology groups are not of finite type, and then they cannot be computed. In pure Kenzo, the user could ask for a homology group of such an space, catching a runtime error.

In our current version of the system, the intermediary layer includes a small expert system, computing, in a symbolic way (that is to say, working with the *description* of the spaces, and not with the spaces themselves considered as Common Lisp objects), the connection degree of a space. The set of rules gives a connection degree to each space builder (for instance, a sphere of dimension n has connection degree $n-1$), and then a rule for each operation on spaces. For

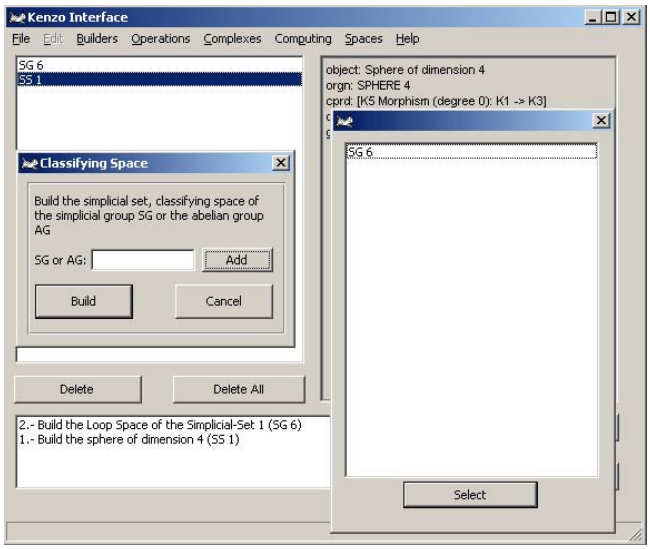


Figure 7: Screen-shot of Kenzo Interface with a session related to classifying spaces.

instance, loop space decreases the connection degree of its input in one unity, suspension increases it in one unity, a cartesian product has, as connection degree, the minimum of the connection degrees of its factors, and so on. From the design point of view, a *Decorator* pattern [12] was used, decorating each space with an annotation of its connection degree in the intermediary layer. Then, when a computation (of a homology group, for instance) is demanded by a user, the intermediary layer monitors if the connection degree allows the transferring of the command to the Kenzo kernel, or a warning must be sent through the external server to the user.

As for item three, the best example is that of the computation of *homotopy groups*. In pure Kenzo, there is no final function allowing the user to compute them. Instead, there is a number of complex algorithms, allowing a user to chain them to get some homotopy groups. Our current user interface has an option to compute homotopy groups. The intermediary layer is in charge of chaining the different algorithms present in Kenzo to reach the final objective. In addition, Kenzo, in its current version, has limited capabilities to compute homotopy groups (depending on the homology of Eilenberg-Mac Lane spaces that are only partially implemented in Kenzo), so the *chaining* of algorithms cannot be *universal* (in this case, a possibility would be to *wire* the enhancement in the GUI, by means of the external XML schema, as in the case of item 1). Thus, the intermediary layer should process the call for a homotopy group, making some consultations to the Kenzo kernel (computing some intermediary homology groups, for instance) before deciding if the computation is possible or not (this is still work in progress).

Regarding point four, our system can be distributed, at present, in two manners: (a) as a stand-alone application, with a heavy client containing the Kenzo kernel to be run in the local host computer; (b) as a light client, containing just the user interface, and every operation and computation is done in a re-

mote server (with the *AllegroServe* technology). The second mode has obvious drawbacks related to the reliability of Internet connections, to the overhead of management where several concurrent users are allowed, etc. But option (a) is not fully satisfactory since interesting Kenzo computations used to be very time and space consuming (requiring, typically, several days of CPU time on powerful computing servers). Thus a mixed strategy should be convenient: the intermediary layer should decide if a concrete calculation can be done in the local computer or it deserves to be sent to a specialized remote server. (In this second case, as it is not sensible to maintain open an Internet connection for several days waiting for the end of a computation, some reactive mechanism should be implemented, allowing the client to disconnect and to be subscribed in some way, to the process of computation in the remote server). The difficulties of this point have two sources: (1) the knowledge here is not based on well-known theorems (as was the case in our discussion on the *connection degree* in the second item of the enumeration), since it is context-dependent (for instance, it depends on the computational power of a local computer), and so it should be based on *heuristics*; (2) the technical problems to obtain an optimal performance are complicated, due, in particular, to the necessity of maintaining a *shared state* between two different computers. These technical aspects are briefly commented in the Open Problems section.

With respect to the kind of heuristic knowledge to be managed into the intermediary level, there is some part of it that could be considered obvious: for instance, to ask for an homology group $H_n(X)$ where the degree n is big, should be considered harder than if n is small, and then one could wonder about a limit for n before sending the computation to a remote server. Nevertheless, this simplistic view is to be moderated by some expert knowledge: it is the case that in some kinds of spaces, difficulties decrease when the degree increases. The heuristics should consider each operation individually. For instance, it is true that in the computation of homology groups of iterated loop spaces, difficulties increase with the degree of iteration. Another measure of complexity is related to the number of times a computation needs to call the Eilenberg-Zilber algorithm (see [10]), where a double exponential complexity bound is reached. Further research is needed to exploit the expert knowledge in the area suitably, in order to devise a systematic heuristic approach to this problem.

6 State of the Project

The work done up to now has allowed us to reach one of the objectives: code reuse. This reusing has two aspects:

1. We have left the Kenzo kernel untouched. This was a goal since the team developing the framework and the user interface, and the team maintaining and extending Kenzo are different. Therefore, it is convenient to keep both systems as uncoupled as possible.

2. The intermediary level has been used, without changes, both in the standalone local version and in the light client with remote server version. A first partial prototype, moving the view towards a web application client (by using *AllegroWebActions*), seems to confirm that the degree of abstraction and genericity reached in our architecture (note that our framework

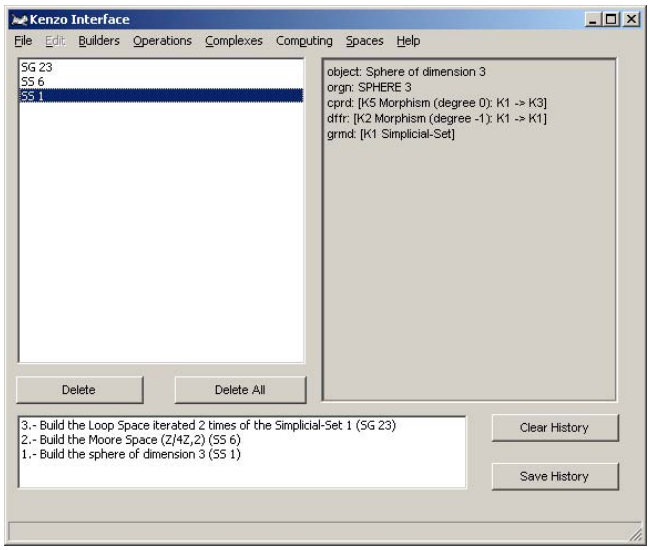


Figure 8: Screen-shot of Kenzo Interface with an example of session.

including several XML formats, each one with different abstraction level) is suitable.

In Figure 8, a screen-shot of our GUI is presented. The main toolbar is organized into 8 menus: *File*, *Edit*, *Builders*, *Operations*, *Complexes*, *Computing*, *Spaces* and *Help*. The rest of the screen is separated into three areas. On the left side, a list with the spaces already constructed during the current session is maintained. When a space is selected (the one denoted by `SS 1` in Figure 8), a description of it is displayed in the right area. At the bottom of the screen, one finds a *history* description of the current session, which can be cleared or saved into a file. It is important to understand that a *history file* is different from a *session file*. The first one is just a plain text description of the commands selected by the user. The second kind of files is described in the next paragraph.

In the current version the *File* menu has just three options: *Exit*, *Save Session* and *Load Session*. When saving a session, a file is produced containing an XML description of the commands executed by the user in that session. In Figure 9 an example of session file can be found, together with a correspondence with their Kenzo counter-parts. At this time, these session files are stored using the standard XML-RPC but our goal, as we show in Figure 9, is to use the external XML schema described in Section 4 (see Figure 5). In this way the session files will be exportable (to be rendered in standard displays, for instance) and even editable from different applications.

The constructors of the spaces we have referred to the first point of Section 5, are collected by the menus *Builders, Operations* and *Complexes*. More specifically, the menu *Builders* includes the main ways of constructing new spaces from scratch in Kenzo as options: spheres, Moore spaces, Eilenberg-Mac Lane spaces, and so on. The menu *Operations* refers to the ways where Kenzo allows the construction of new simplicial spaces from other ones: loop spaces, classifying spaces, Cartesian products, suspensions, etc. The menu *Complexes* is

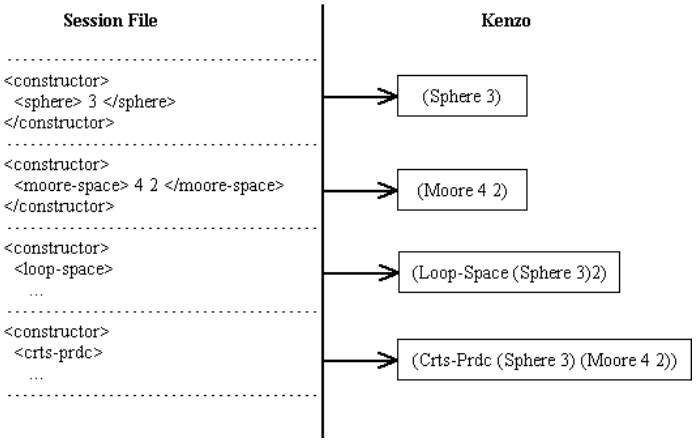


Figure 9: Sample of a session file.

similar, but related to chain complexes instead of simplicial objects (here, for instance, the natural product is the tensorial product instead of the cartesian one).

The menus *Computing* and *Spaces* collect all the operations on concrete spaces (instead of *constructing spaces*, as in the previous cases). Both of them provide their items with all the necessary "intelligence" in order to avoid raising runtime errors. In *Computing* we concentrate on calculations *over* a space. We offer to compute homology groups, to compute the same but with explicit generators and to compute homotopy groups, in this last case we find the third kind of enhancement. In menu *Spaces* currently we only offer the possibility of showing the structure of a simplicial object (this is only applicable to *effective*, finite type spaces).

To consider a first complete (beta) version of the system, it is necessary to complete the questions already mentioned in the text relating to finishing the external XML schema definition and to controlling the cases in which homotopy groups can be effectively computed by Kenzo.

Moreover, we have planned to develop two more tools:

1. In the menu *Builders*, there is a still inactivated slot called *Build-finite-ss*, aimed at emulating, in our environment, the utility present in pure Kenzo which allows the user to construct step-by-step, in an interactive manner, a finite simplicial set (checking, in each step, whether faces are glued together in a coherent way). To this aim, we are thinking of designing a graphical tool.

2. In the menu *Spaces*, it is necessary to include the possibility of operating locally *inside* a selected space. For instance, given a simplex to compute one of its faces or given two simplexes in the same dimension we can compute its product in a selected simplicial group. The difficulty here is related to designing an editor for elements (data of the second kind, using the terminology in Section 5), which can be given as inputs to the local operations. This will give content to the *Edit* menu, in the main toolbar, which is now inactivated.

These extra functionalities are rather a matter of standard programming, and it is foreseen that no research problem will appear when tackling them. The questions discussed in the next section, on the contrary, could imply important challenges.

7 Open Problems

The most important issue to be tackled in the next versions of the system is how organizing the decision on when (and how) a calculation should be derived to a remote server. To understand the nature of the problem it is necessary to consider that there are two kinds of *state* in our context. Starting from the most simple, the *state of a session* can be described by means of the spaces that have been constructed so far. Then, to encode (and recover) such a state, a session file as explained in the previous section would be enough: an XML document containing a sequence of calls to different constructors and methods. In this case, when a calculation is considered too hard to be computed in a local computer, the whole session file could be transmitted to the remote server. There, executing step-by-step the session file, the program will re-find the same state of the local session, proceeding to compute the desired result and sending it to the client. Of course, as mentioned previously, some kind of subscription tool should be enabled, in such a way that the client could stop its running, and then to receive the result (or a notification indicating the result is already available somewhere), after some time (perhaps some days or weeks of computation on the remote server).

Even if this approach can be considered reasonable as a first step, it has turned out to be too simplistic to deal with the richness of Kenzo. A space in Kenzo consists in a number of methods describing its behavior (explaining, for instance, how to compute the faces of its elements). Due to the high complexity of the algorithms involved in Kenzo, a strategy of *memoization* has been systematically implemented, as we already commented in Section 2.3. As a consequence, the *state of a space* evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a face, depends on the concrete states of every space involved in the calculation (in the more explicit case, to re-calculate a face on a space could be negligible in time, even if in the first occasion this was very time consuming). This notion of *state of a space* is transmitted to the notion of *state of a session*. We could speak of two *states of a session*: the one *sallow* evoked before, that is essentially *static* and can be recovered by simply re-executing the top-level constructor calls; and the other *deep state* which is dynamic and depends on the computations performed on the spaces.

To analyse the consequences of this Kenzo organization, we should play with some scenarios. Imagine during a local session a very time consuming calculation appears; then we could simply send the *sallow state of the session* to the remote server, because even if some intermediary calculations have been stored in local memory, they can be re-computed in the remote server (finally, if they are cheap enough to be computed on the local computer, the price of re-computing them in the powerful remote server would be low). Once the calculation is remotely finished, there is no possibility of sending back the *deep state* of the remote session to the local computer because, usually, the memory

used will exhaust the space in the local computer. Thus, it could seem that to transmit the *sallow state* would be enough. But, in this picture, we are losing the very reason why Kenzo uses the memoization (dynamic programming) style. Indeed, if after obtaining a difficult result (by means of the remote server) we resume the local session and ask for another related difficult calculation, then the remote server will initialize a new session from scratch, being obligated to recalculate every previous difficult result, perhaps making the continuation of the session impossible. Therefore, in order to take advantages of all the possibilities Kenzo is offering now on powerful scientific servers, we are faced with some kind of *state sharing* among different computers (the local computers and the server), a problem known as difficult in the field of distributed object-oriented programming.

In short, even if our initial goal was not related to distributed computing, we found that in order to enable our intermediary layer as an *intelligent assistant* with respect to the classification of calculations as simple (runnable on a standard local computer) or complicated (to be sent to a remote server), we should solve problems of distributed systems. Thus, a larger perspective is necessary, and we are working with the *Broker* architectural pattern, see [3], in order to find a natural organization of our intermediary layer.

8 Conclusions

The current state of our project can be considered solid enough to be a good point of continuation for all our objectives. We have showed how some *intelligent guidance* can be achieved in the field of Computational Algebraic Topology, without using standard Artificial Intelligence techniques. The idea is to build an intermediary layer, giving a mediated access to an already-written symbolic computation system. Putting together both Kenzo itself and the intermediary layer, we have produced a framework which is able to be connected to different clients (desktop GUIs, web applications and so on). In addition, with this framework, several profiles of interaction can be considered. In general, this can imply a restriction of the full capabilities of the kernel system, but the interaction with it is easier and enriched, contributing to the objective of increasing the number of users of the system.

References

[1] Andrés M., Pascual V., Romero A., Rubio J., *Remote Access to a Symbolic Computation System for Algebraic Topology: A Client-Server Approach*, Lecture Notes in Computer Science 3516 (2005) 635–642.

[2] Ausbrooks R. et al., *Mathematical Markup Language (MathML)* Version 2.0 (second edition), 2003. http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[3] Buschmann, F., Meunier, R., Rohnert H., Sommerland P., Stal M., *Pattern-oriented software architecture. A system of patterns, Volume 1*, Wiley, 1996.

[4] Buswell S., Caprotti O., Carlisle D.P., Dewar M.C., Gaëtano M., Kohlhase M. *OpenMath* Version 2.0, 2004. http://www.openmath.org/.

[5] Calmet, J., Homann, K., *Towards the Mathematics Software Bus*, Theoretical Computer Science 187 (1997) 221–230.

[6] Carlsson G., Milgram R. J., *Stable homotopy and iterated loop spaces.* in [14], pp 505-583.

[7] Cordero C. C., De Miguel A. , Domínguez E., Zapata Mł A., *Modelling Interactive Systems: an architecture guided by communication objects in HCI related papers of Interacción 2004*, Springer (2006) 345–357.

[8] Domínguez C., Lambán L., Rubio J., *Object oriented institutions to specify symbolic computation systems*, Rairo - Theoretical Informatics and Applications 41 (2007) 191-214.

[9] Domínguez E., Zapata M.A., *Noesis: Towards a situational method engineering technique*, Information Systems 32,2 (2007) 181-222.

[10] Dousson X., Rubio J., Sergeraert F., Siret Y., *The Kenzo program.* `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`

[11] Franz Inc. *Allegro Common Lisp.* http://www.franz.com/.

[12] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[13] González-López M. J., González-Vega L., Pascual A., Callejo E., Recio T., Rubio J., *TutorMates.* http://www.tutormates.es/.

[14] *Handbook of Algebraic Topology* (Edited by I.M. James). North-Holland (1995).

[15] *Internet Accessible Mathematical Computation (IAMC).* http://icm.mcs.kent.edu/research/iamc.html.

[16] Lambán L., Pascual V., Rubio J., *An object-oriented interpretation of the EAT system*, Applicable Algebra in Engineering, Communication and Computing 14 (2003) 187–215.

[17] Object Management Group. *Common Object Request Broker Architecture (CORBA).* http://www.omg.org.

[18] Rubio J., Sergeraert F., *Constructive Algebraic Topology*, Bulletin des Sciences Mathématiques 126 (2002) 389-412.

[19] Schelter W., *Maxima.* http://maxima.sourceforge.net/index.shtml.

[20] Sergeraert F., $\maltese_k$*, objet du* 3^e *type.* Gazette des Mathématiciens, 2000, vol. 86, pp 29-45.

[21] Stein W., *SAGE mathematical software system.* http://sage.scipy.org/sage/.

[22] Winer D., *Extensible Markup Language-Remote Procedure Call (XML-RPC).* http://www.xmlrpc.com.