Chapter 3

Extensibility and Usability of the Kenzo framework

The previous chapter was mainly devoted to present a framework which provides a mediated access to the Kenzo system, constraining its functionality, but providing guidance to the user in his navigation on the system. However, we cannot consider that the Kenzo framework is enough to fully cover our aims.

On the one hand, we want to be able to increase the capabilities of the system by means of new Kenzo functionalities or the connection with other systems such as Computer Algebra systems or Theorem Prover tools. On the other hand, the XML interface (to be more concrete, the OpenMath interface) is not desirable for a human user; then, a more suitable way of interacting with the Kenzo framework must be provided.

To cope with the extensibility challenge, we have deployed the Kenzo framework as a client of a *plug-in* framework that we have developed. This allows us to add new functionality to the Kenzo framework without accessing to the original source code. Moreover, as a client of both Kenzo and plug-in frameworks, an extensible friendly front-end has been developed. Combining the frameworks and the front-end we are able to improve the usability and the accessibility of the Kenzo system, increasing the number of users who can take profit of the Kenzo computation capabilities. The whole system, that is to say, the merger of the two frameworks and the front-end, is called *fKenzo*, an acronym for *f*riendly Kenzo [Her09].

The rest of this chapter is organized as follows. The *plug-in* framework, which the Kenzo framework is a client of, is explained in Section 3.1. The main client of the Kenzo framework, an extensible user interface, is introduced in Section 3.2.

3.1 A plug-in framework

Architectures providing plug-in support are used for building software systems which are extensible and customizable to the particular needs of an individual user. Several interesting plug-in approaches exist. One of the first plug-in platforms was Emacs ("Editor MACroS") [Sta81], whose extensions are written in *elisp* (a Lisp dialect) and can be added at runtime. The Eclipse platform [Ecl03] is certainly the most prominent representative of those plug-in platforms and has driven the idea to its extreme: "Everything is a plug-in". Plug-ins for Eclipse are written in Java. Other examples are OSGi [OSG03], a Java-based technology for developing components, Mozilla [Moz], a web browser with a great amount of extensions, or Gimp [Pec08], a famous image processing software which allows one the addition of new functionality by means of plug-ins.

Kenzo is also a plug-in system: to add new functionality to the Kenzo system, a file with the new functions must be included; remaining untouched the Kenzo core. In general, all the systems implemented in Common Lisp are extensible. Therefore, to extend the functionality of a Common Lisp program, we, usually, only need to load new functions by means of Common Lisp files.

One of the most important challenges that we faced in the development of the Kenzo framework was the management of its extensibility. On the one hand, we wanted that the Kenzo framework could evolve at the same time as Kenzo. On the other hand, the framework should be enough flexible to integrate new tools, such as other Computer Algebra systems and Theorem Provers tools. In addition, we wanted that the original source code of the Kenzo framework remained untouched when the system was extended. Therefore, we decided to include a *plug-in* support in our framework by means of a *plug-in* framework developed by us.

The rest of this section is organized as follows. Subsection 3.1.1 is devoted to present the architecture of the plug-in framework. Explanations about the way of adding new functionality to the Kenzo framework through the plug-in framework are provided in Subsection 3.1.2.

3.1.1 Plug-in framework architecture

Extensibility is very important because an application is never really finished. There are always improvements to include and new features to implement.

With the aim of being able to extend different systems we have developed a plug-in framework. This framework may have systems of very different nature as clients. Therefore, the way of extending a client can be different to the way of extending the rest of them. This issue has been taken into account in the design of the plug-in framework.

The implementation of the plug-in framework has been based on the *plug-in* pattern presented in [MMS03]. This pattern distinguishes two main components: the *plug-ins*

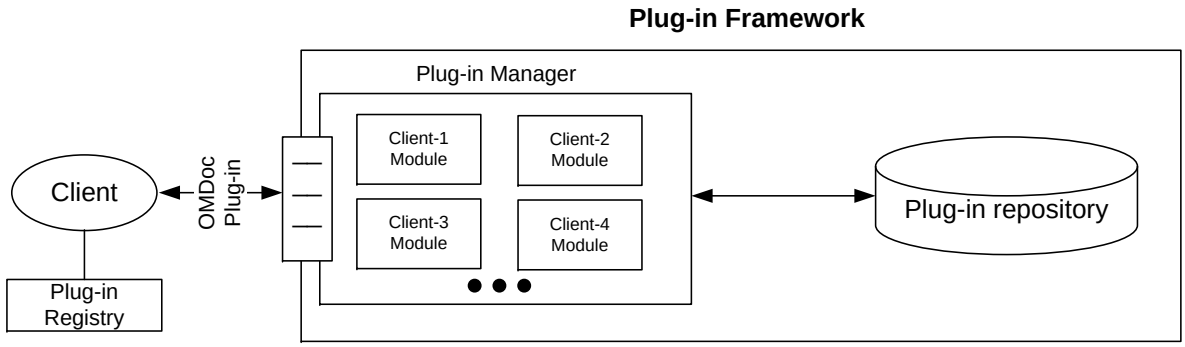

Figure 3.1: Plug-in framework

and the *plug-in manager*. Moreover, we have included two improvements of our own, the *plug-in repository* and the *plug-in registries* (each client of the *plug-in framework* will have associated a *plug-in registry*). A high level perspective of the plug-in framework is depicted in Figure 3.1. Let us explain each one of the constituents of our plug-in framework.

A *plug-in* in our system is an OMDoc document which consists of different resources used to extend a concrete client. Since the clients of the plug-in framework can be very different, the nature of the resources included in them are different, too. However, as it is desirable that the format of all the *plug-ins* (not the format of the resources which obviously depends on each client) was common for the different clients, we have chosen *OMDoc* [Koh06] as the format to encode our *plug-ins*. OMDoc is an open markup language for mathematical documents, and the knowledge encapsulate in them. The OMDoc plug-ins are documents that wrap the necessary resources to extend a concrete client of the plug-in framework.

Essentially, an OMDoc plug-in is an XML document, that stores the metadata about the plug-in (authorship, title, and so on) and wraps the resources (by means of references to other files that, of course, depend on the concrete client) which extend a client of the plug-in framework.

For instance, let us suppose that we have a client, called `client-1`, of our plug-in framework; and we want to extend that client by means of the resources stored in a file called `client-1-resources`. Then, the OMDoc plug-in, called `new-module-client-1`, will contain, in addition to the metadata about the *plug-in*, the following XML fragment (all the OMDoc plug-ins follow the same pattern).

```
<code id="new-module-client-1">
   <data format="client-1"> client-1-resources </data>
</code>
```

The above XML code must be read as follows. The `id` argument of the `code` tag indicates the name of the new module. Inside this tag the different resources to extend

`client-1` with the new plug-in are referenced by means of the `data` tag. This tag has as `format` attribute the name of the concrete client, in this case `client-1`, and the value of the `data` tag is the reference to the concrete resource. This information is very useful for the *plug-in manager*.

When the plug-in framework receives as input an OMDoc plug-in, the plug-in manager is invoked. The *plug-in manager* is a Common Lisp program that consists of several modules, one per client of the plug-in framework. As we have said, each one of these modules is related to a concrete client, and then, is implemented depending on the extensibility needs of each one of them. The plug-in manager, depending on the information stored in the plug-in (namely the value of the `format` attribute of the `data` tag), invokes the corresponding client module. Subsequently, this module extends the client with the indicated resources.

The plug-ins and the resources referenced by them are stored in a folder included in the plug-in framework called the *plug-in repository*.

Finally, each client of the plug-in framework has associated a file called *plug-in registry*. Each one of these files stores a list of the *plug-ins* that were added to the corresponding client. When a new plug-in is included in a client, the information about that plug-in is stored in its plug-in registry. Moreover, when a client is started its first task consists in sending the information of the *plug-in registry* to the *plug-in manager* in order to achieve the same state of the last configuration of the client.

If some problem appears during the loading of a plug-in the plug-in framework informs of the error and aborts the loading in order to avoid inconsistencies.

Once we have presented the plug-in framework, let us explain how one of its clients, that is the Kenzo framework, uses it. Another client of this framework is a customizable front-end that will be presented in Section 3.2.

3.1.2 The Kenzo framework as client of the plug-in framework

As it was discussed earlier, one of the most important issues tackled in the design of the Kenzo framework was the deployment of an extensible architecture. A good approach to solve this question consists in having a component-based architecture as base-system, and then equipping it with different components. This is the approach followed in the Kenzo framework. As was explained in the previous chapter, the base-system of the Kenzo framework is based on the *Microkernel* architectural pattern, therefore, we have a component-based architecture. Moreover, to be able to include new improvements and features, the Kenzo framework has been implemented as a client of the plug-in framework.

As we explained previously, all the Kenzo framework components are Common Lisp modules, see Section 2.2, and since Common Lisp programs are designed to be extensible, we only need to load new functions in each component by means of Common Lisp files.

This is the same method followed in Kenzo to extend its functionality.

The plug-in manager of the plug-in framework contains a component that is devoted to load new functionality in the Kenzo framework. This module extends the functionality of the Kenzo framework components with two different aims. On the one hand, to provide access to new Kenzo functionality through the Kenzo framework. On the other hand, to interact with other systems, such as Computer Algebra systems or Theorem Provers tools, by means of the Kenzo framework. Let us present the integration of new functionality in the Kenzo framework.

Let us suppose that we have developed a new module for Kenzo that allows us to construct a kind of spaces that were not included in the Kenzo system, we will see concrete examples in Chapter 5, and, of course, we want to include the new functionality in our framework. Then, we need to extend all the components of the framework by means of the following files:

- a file (let us called it `new-constructor.lisp`) with the functionality to include the new constructor developed for the Kenzo system in the Kenzo component of the internal server,

- a file (let us called it `new-constructor-is.lisp`) with the functionality to expand the interface of the internal server to support the access to the new Kenzo functionality,

- a file (let us called it `new-constructor-m.lisp`) with the functionality for the microkernel to include a new construction module. This functionality follows the pattern explained in Subsubsection 2.2.3.3 for constructions modules. Moreover, this file also expands the microkernel interface to provide access to the functionality of the new construction module,

- the new specification of the XML-Kenzo language including the new constructor. As we have explained previously in Subsection 2.2.4, this will extend the capabilities of the external server. The file containing the XML-Kenzo specification is the `XML-Kenzo.xsd` file, and

- a file (let us called it `new-constructor-a.lisp`) with the functionality which increases the adapter functionality to transform from OpenMath requests to XML-Kenzo requests.

The `new-constructor` plug-in is an OMDoc document which contains some metadata about the document and references the different files in the following way.

```
<code id="new-constructor">
   <data format="Kf/internal-server"> new-constructor.lisp </data>
   <data format="Kf/internal-server"> new-constructor-is.lisp </data>
   <data format="Kf/microkernel"> new-constructor-m.lisp </data>
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/adapter"> new-constructor-a.lisp </data>
</code>
```

Each `data` element has specified in the `format` attribute one of the Kenzo framework components (internal server, microkernel, external server and adapter). The value of a `data` element is a file which extends the Kenzo framework. The indicated component in the `format` attribute of a `data` element is extended by means of the file indicated with the value of the `data` element. For instance in the case of the first `data` element, the functionality of the internal server is extended by means of the `new-constructor.lisp` file.

The plug-in framework receives as input the `new-constructor` plug-in. Subsequently, the Kenzo framework module of the plug-in framework is invoked. This module is split in four constituents, one per Kenzo framework component. The behavior of the constituents in charge of the internal server, the microkernel and the adapter consists in loading the new functionality in the corresponding Kenzo framework component. The constituent for the external server overwrites the XML-Kenzo specification with the new one (no additional interaction is needed to extend the functionality of the external server, since when the XML-Kenzo specification is changed the external server is automatically upgraded).

Finally, the information of the new plug-in is stored in the plug-in registry of the Kenzo framework to store the configuration for further uses.

This was related to the inclusion of new Kenzo functionality in the Kenzo framework; the case of widening the Kenzo framework through the addition of an internal server (a Computer Algebra system or a Theorem Prover tool) is practically analogous. The main difference is the file related to the internal server which, instead of adding new Kenzo functionality to the current internal server, shows how to connect with the new system. Examples of the addition of new internal servers will be presented in Chapter 4.

The features associated to the Kenzo framework owing to its implementation as a client of the plug-in framework are listed below.

- Extensibility: the Kenzo framework can be extended by plug-ins. Each new functionality can be realized as an independent plug-in.
- Flexibility: each unnecessary plug-in can be removed and each necessary plug-in can be loaded at run-time. Therefore the Kenzo framework can be configured in such a way that it has only the needed functionality.
- Storage of configuration: thanks to the registry associated to the Kenzo framework,

the configuration of a session is stored for the future.

- Easy to install: the installation of plug-ins is friendly from the plug-in folder.

3.2 Increasing the usability of the Kenzo framework

The design of a client for our framework was one of the most important issues tackled in our development. The client should not only take advantage of all the enhancements included in the Kenzo framework but also be designed to increase the usability and accessibility of the Kenzo system. In this context, the program designers in Symbolic Computation always meet the same decision problem: two possible organizations[1].

1. Provide a package of procedures in the programming language L, allowing a user of this language to load this package in the standard L-environment, and to use the various functions and procedures provided in this package. The interaction with the procedures is by means of a command line interface.

2. Provide a graphical user interface (GUI) with the usual buttons, menus and other widgets to give to an inexperienced user a direct access to the most simple desired calculations, without having to learn the language L.

The main advantage of the former alternative is that the total freedom given by the language L remains available; however, the technicalities of the language L remain present as well. Moreover, the *prompt* of a command line interface does not usually provide adequate information to the user about the correct command syntax. Besides, since the user has to memorise the syntax and options of each command, it often takes considerable investment in time and effort to become proficient with the program. Therefore, command line interfaces may not be appealing to new or casual users of a software system. The vast majority of Computer Algebra systems fall in this category, including GAP, CoCoA, Macaulay, and Kenzo. On the contrary, graphical interfaces (the second alternative) are easier to use, since instead of relying on commands, the GUI communicates with the user through objects such as menus, dialog boxes and so on. These objects are a means to provide a more intuitive user interface, and supply more information than a simple prompt. However, no reward comes without its corresponding price and GUIs can slow down expert users.

Since the final users of our framework are Algebraic Topology students, teachers or researchers that usually do not have a background in Common Lisp, we opted for implementing a user-friendly front-end allowing a topologist to use the Kenzo program guiding his interaction by means of the Kenzo framework, without being disconcerted by the Lisp technicalities which are unavoidable when using the Kenzo system. This GUI is communicated with the Kenzo framework through the OpenMath interface of the

[1] These are, of course, two extreme positions: many other possibilities can be explored between them.

adapter and can be customized by means of the plug-in framework. The whole system, that is to say the two frameworks and the GUI, is called *fKenzo*, an acronym for friendly Kenzo [Her09]. We have talked at length about the Kenzo framework in Chapter 2 and also about the plug-in framework in Section 3.1, so, let us present now the GUI of the *fKenzo* system.

The *fKenzo* GUI was implemented with the *IDE* of Allegro Common Lisp [Incc]. Various features have been implemented in this GUI. They improve the interaction with the Kenzo system from the user point of view. The main features of the *fKenzo* GUI are listed below.

1. *Easy to install*: the installation process is based on a typical Windows installation.
2. *No external dependencies*: the *fKenzo* GUI does not need any additional installation to work.
3. *Functionality*: the *fKenzo* GUI allows the user to construct topological spaces of regular usage and compute homology and (some) homotopy groups of these spaces.
4. *Error handling*: this GUI is a client of the Kenzo framework; and all the enhancements included in the framework are inherited by the GUI. In this way, the user is guided in his interaction with the system and some errors are avoided.
5. *Consistent metaphors*: an advanced user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured in the GUI.
6. *Interaction styles*: the user can interact with the GUI by means of the mouse and also using keyboard shortcuts.
7. *Mathematical rendering*: the *fKenzo* GUI shows results using standard mathematical notation.
8. *Storage of sessions*: session files include the spaces constructed during a session, and can be saved and loaded in the future. Moreover, these session files can be exported, used to communicate them to other users and rendered in browsers.
9. *Storage of results*: result files storing the results obtained during a session, as in the case of session files, can be saved, exported, used to communicate them to other users and rendered in browsers.
10. *Customizable*: the *fKenzo* GUI can be configured to the needs of its users.
11. *Extensibility*: the *fKenzo* GUI can evolve with the Kenzo framework using the plug-in framework.

The rest of this section is devoted to present the GUI of the *fKenzo* program, trying to cover both the user (Subsections 3.2.1 and 3.2.2) and the developer (Subsections 3.2.3 and 3.2.4) perspectives.

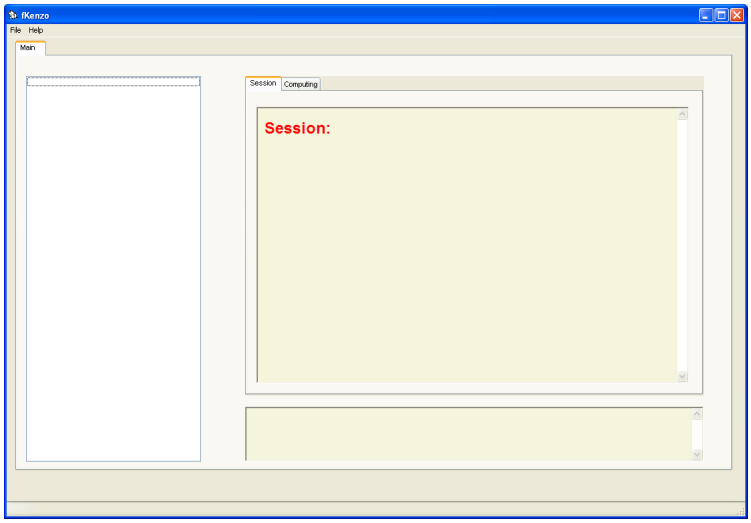

Figure 3.2: Initial screen of *fKenzo*

3.2.1 *fKenzo* GUI: a customizable user interface for the Kenzo framework

To use *fKenzo*, one can go to [Her09] and download the installer. After the installation process, the user can click on the *fKenzo* icon, accessing to an "empty" interface, which is shown in Figure 3.2.

Then, the first task of the user consists in loading some functionality in the *fKenzo* GUI through the different *modules* included in the distribution of *fKenzo*. From the user point of view, a module is a file which loads functionality in the *fKenzo* GUI.

The main toolbar of the *fKenzo* GUI is organized into two menus: *File* and *Help*. The *File* menu has the following options: *Add Module*, *Delete Module*, and *Exit*. The aim of the *Add Module* option consists in loading the functionality of a module. The user can configure the interface by means of five modules: *Chain Complexes*, *Simplicial Sets*, *Simplicial Groups*, *Abelian Simplicial Groups* and *Computing*. Each one of these modules corresponds with one of the XML-Kenzo groups explained in Subsection 2.2.1. In addition, some other *experimental* modules can be installed, such as a possibility of interfacing the GAP Computer Algebra system or the ACL2 Theorem Proving tool; these modules will be presented in Chapter 4.

When one of the "construction modules" (*Chain Complexes*, *Simplicial Sets*, *Simplicial Groups* or *Abelian Simplicial Groups*) is loaded, a new menu appears in the toolbar allowing the user to construct the spaces specified in the XML-Kenzo schema for that type, see Figure 2.5 of Subsection 2.2.1. Moreover, two new options called *Save Session* and *Load Session* are added to the *File* menu. When saving a session a file is produced containing the spaces which have been built in that session. These session files are saved using the OpenMath format and can be rendered in different browsers. These session files can be loaded, from the *Load Session* option, and allow the user to resume a saved

session.

When the *Computing* module is loaded, a new menu appears in the toolbar allowing the user to compute homology and homotopy groups of the constructed spaces. Besides, a new option called *Save Computing* appears in the *File* menu. This option works in a similar way to *Save Session* but instead of saving the spaces built during the current session, it saves the computations also using the OpenMath format. However, these results cannot be reloaded into the *fKenzo* GUI, since they cannot be re-used in further computations. It is worth noting that computations depend on the state of the system, and this state cannot be exported, so we cannot re-use computations performed in a different session. When the user exits *fKenzo*, its configuration is saved for future sessions.

The visual aspect of the *fKenzo* panel is as follows. The "Main" tab contains, at its left side, a list of the spaces constructed in the current session, identified by its type (CC = Chain Complex, SS = Simplicial Set, SG = Simplicial Group, AG = Abelian simplicial Group) and its internal identification number (the `idnm` slot of the `mk-object` instance of the microkernel). When selecting one of the spaces in this list, its standard notation appears at the bottom part of the right side. At the upper part, there are two tabs "Session" (containing a textual description of the constructions made, see an example of session in Figure 3.3) and "Computing" (containing the homology and homotopy groups computed in the session; see an example in Figure 3.4). In both "Session" and "Computing" tabs the results are rendered using again standard mathematical notation.

In the *fKenzo* GUI, focus concentrates on the object (space) of interest, as in Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it (in particular, with respect to computing results). It is separated by means of tabs not only because on the division between spaces and computed results (the system moves from one to the other dynamically, putting the focus on the last user action), but also due to the extensibility of *fKenzo*. To be more precise, our system is capable of evolving to integrate with other systems (computational algebra systems or theorem proving tools), so playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns. Examples of the integration of a Computer Algebra system and a Theorem Prover tool in *fKenzo* will be presented in Chapter 4.

3.2.2 *fKenzo* in action

To illustrate the performance of the *fKenzo* program, let us consider a hypothetical scenario where a graduate course is devoted to *fibrations*, in particular introducing the functors *loop space* Ω and *classifying space* B. In the simplicial framework, [May67] is a good reference for these subjects. In this framework, if X is a connected *space*, its loop space ΩX is a simplicial group, the structural group of a universal fibration $\Omega X \hookrightarrow PX \to X$. Conversely, if G is a simplicial group, the classifying space BG is the base space also of a universal fibration $G \hookrightarrow EG \to BG$. The obvious symmetry

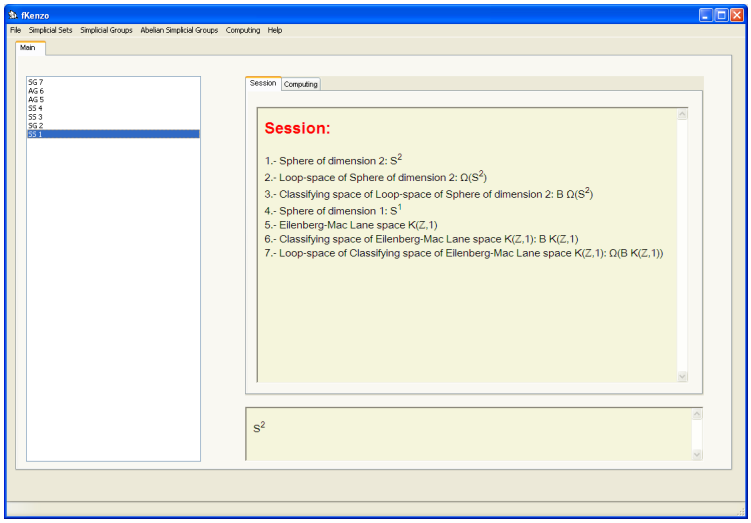

Figure 3.3: Example of session in *fKenzo*

between both situations naturally leads to the question: in an appropriate context, are the functors Ω and B inverse of each other?

For the composition $B\Omega$, let us compare for example the first homology groups of S^2 with the homology groups of $B\Omega S^2$, and the homology groups of ΩS^2 and $\Omega B\Omega S^2$.

Since we are planning to work with simplicial sets and simplicial groups, we can load the necessary functionality using *File* $\rightarrow$ *Add Module*, and then choosing `Simplicial-Groups.omdoc`. The interface changes including now a new menu called *Simplicial Sets* and another one called *Simplicial Groups*. When selecting *Simplicial Sets* $\rightarrow$ *sphere*, *fKenzo* asks for a natural number, as we saw in the definition of XML-Kenzo, limited to 14. Then (see Figure 3.3) the space denoted by `SS 1` appears in the left side of the screen; when selecting it, the mathematical notation of the space appears in the bottom part of the right side of the panel. The history of the constructed spaces is shown in the "Session" tab.

If we try to construct a classifying space from the *Simplicial Group* menu, *fKenzo* informs us that it needs a simplicial group (thus likely an error is avoided). We can then construct the space ΩS^2. Since ΩS^2 is the only simplicial group in this session, when using *Simplicial Groups* $\rightarrow$ *classifying space*, ΩS^2 is the only available space appearing in the list which *fKenzo* shows. Finally, we can construct the space $\Omega B\Omega S^2$.

When loading the `Computing.omdoc` file, the menu *Computing*, where we can select "homology", becomes available. In this manner we can compute the homology groups of the spaces as can be seen in Figure 3.4.

These results give some plausibility to the relations $B\Omega = id$ and $\Omega B = id$.

The reader could wonder why the simpler case of S^1 has not been considered. Using again *fKenzo* this time a warning is produced; yet the result $B\Omega S^1 \sim S^1$ is true. But the Eilenberg-Moore spectral sequence cannot be used in this case to compute $H_*\Omega S^1$,

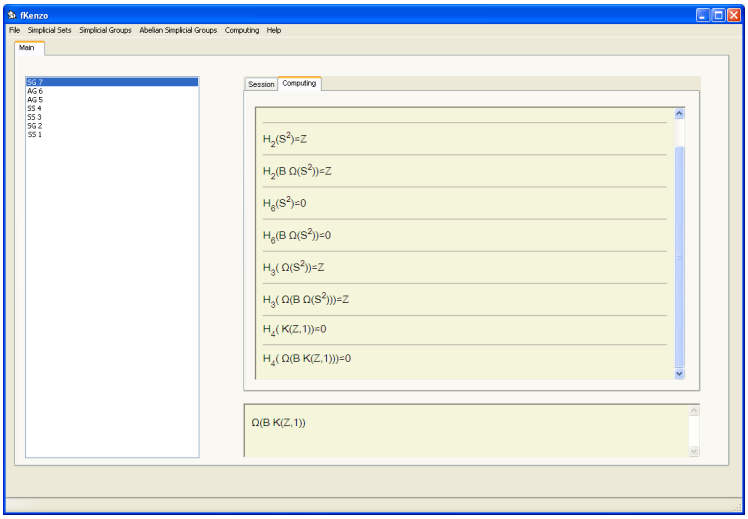

Figure 3.4: Example of computations in *fKenzo*

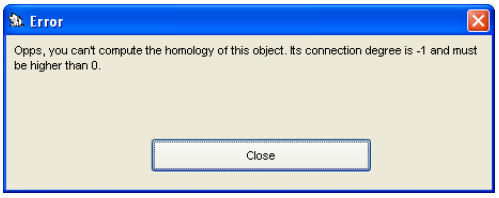

Figure 3.5: Reduction degree error in *fKenzo*

because S^1 is not simply connected, and *fKenzo* checks this point (see Figure 3.5). The symmetry comparison between S^1 and ΩBS^1 fails too, for another reason: the "standard" S^1 is a topological group, but the standard simplicial representation of S^1 cannot be endowed with a structure of simplicial group. This situation is reflected in the construction of classifying spaces in *fKenzo* since the sphere S^1 does not appear in the list which *fKenzo* shows.

This is a good opportunity to introduce the Eilenberg MacLane space $K(\mathbb{Z}, 1)$, the "minimal" Kan model of the circle S^1, a simplicial group. In order to work with Eilenberg MacLane spaces in *fKenzo*, the *Abelian Simplicial Group* module should also be loaded, in this way the space $K(\mathbb{Z}, 1)$ can be built, as can be seen in Figure 3.3. The *fKenzo* comparison between the first homology groups of $K(\mathbb{Z}, 1)$ and $\Omega BK(\mathbb{Z}, 1)$ does give the expected result.

We think this illustrates how *fKenzo* can be used as a research tool, precisely a specialized computer tool, for Algebraic Topology.

Up to now, we have presented the user point of view. In the following subsections, some explanations on the development of the *fKenzo* GUI are given.

3.2.3 Customization of the *fKenzo* GUI

The most important challenge that we have faced in the development of the *fKenzo* GUI was the deployment of an extensible and modular user interface. Modularity has two aims in *fKenzo*. One of them is related to the separation of concerns in the user interface. The second one allows us to design a dynamically extensible GUI, where modules are plugged in.

3.2.3.1 Declarative programming of User Interfaces

In all the graphical user interfaces exist a separation of concerns, hence if we want to extend a GUI we need to extend it at all its levels. With respect to this aspect, our inspiration comes from [HK09], where a proposal for declarative programming of user interfaces was presented. In [HK09], the authors distinguished three constituents in any user interface: *structure*, *functionality* and *layout*.

Structure: Each user interface (*UI*) has a specific hierarchical *structure* which typically consists of basic elements (like text input fields or selection boxes) and composed elements (like dialogs).

Functionality: When a user interacts with a UI, some events are produced and the UI must respond to them. The event handlers are functions associated with events of some widget and that are called whenever such event occurs (for instance clicking over a button).

Layout: The elements of the structure are put in a layout to achieve a visually appealing appearance of the UI. In some approaches layout and structural information were mixed, however in order to obtain clearer and reusable implementations these issues should be distinguished.

The approach presented in [HK09] used the *Curry* language [Han06] to declare all the ingredients. Instead of doing a similar work, but using the Common Lisp language (recall that our GUI is implemented in Common Lisp) to all the ingredients, we have preferred to employ different technologies devoted to each one of the constituents. Let us present each one of these ingredients in our context using a concrete example, that is the window used to construct a sphere in the *fKenzo* GUI, see Figure 3.6.

The *structure* of our GUI is provided by *XUL* [H$^+$00]. XUL, XML User Interface, is Mozilla's XML-based user interface language which lets us build feature rich cross-platform applications defining the structure of all the elements of a UI. Then, a XUL description must be provided in order to define the structure of the elements of our GUI. The main reason to encode the structure of our GUI by means of XUL is the reusability of this language. The XUL code can be used in order to build forms in different environments for different applications without designing new interfaces.

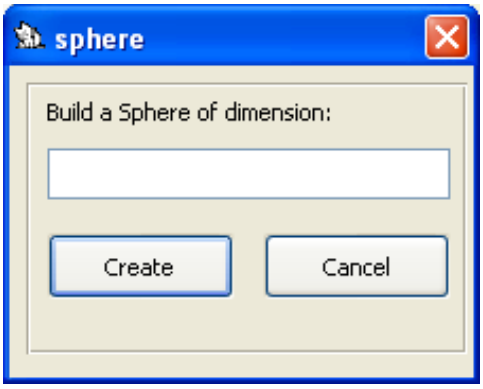

Figure 3.6: Sphere dialog

Let us examine the structure of the window of Figure 3.6. That window is called "sphere" and gathers in a groupbox the following elements: the text "Build a Sphere of dimension:", a textbox to introduce a natural number between 1 and 14, and a row that contains both "Create" and "Cancel" buttons. Moreover, each button has associated an event when its state is changed, namely when the button is clicked. We can specify that structure in the following XUL code:

```
<window name="sphere">
 <groupbox>
     <label value="Build a Sphere of dimension:"/>
     <textbox id="n" type="number" min="1" max="14"/>
     <hbox>
       <button label="Create" name="create" event="on-change"/>
       <button label="Cancel" name="cancel" event="on-change"/>
     </hbox>
 </groupbox>
</window>
```

As we have said previously, we can use the above XUL code in different clients. For instance, the previous XUL is presented in a Mozilla browser [Moz] as shown in Figure 3.7.

As can be thought, providing the XUL description of an element of a GUI can be a tedious task due to the XML nature of XUL. To make this task easier, an interpreter which is able to convert from the Allegro Common Lisp *IDE* forms to their XUL representation, and viceversa, has been developed. This is a more comfortable way of working because we define the forms in the Allegro *IDE* using a graphical interface, and then, the interpreter automatically generates the XUL code.

The *functionality* of the elements of our GUI, that is the set of *event handlers*, has been programmed in Common Lisp keeping the following convention in order to define the names of the *event handlers* functions:

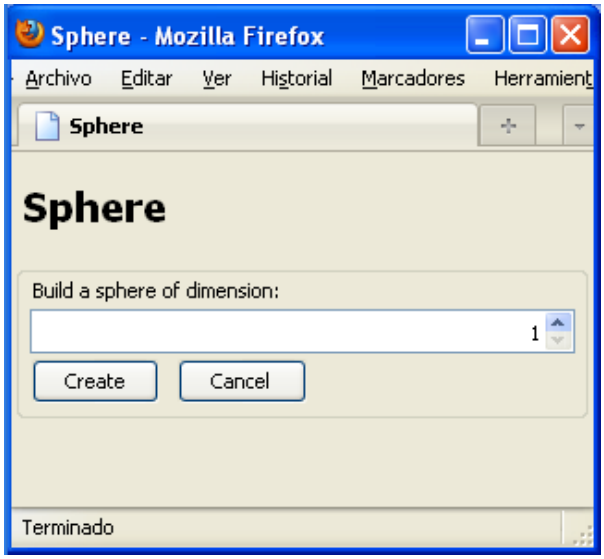

Figure 3.7: Sphere form in Firefox browser

```
(defun <window-name>-<element-name>-<event> (params)
  ;; event handler code)
```

where `<window-name>`, `<element-name>` and `<event>` must be replaced with the name of the dialog, the name of the element and the event, respectively. For instance, the event associated with the "Create" button of the sphere dialog, see Figure 3.6, is codified as follows:

```
(defun sphere-create-on-change (params)
  ;; event handler code)
```

Finally, the *layout* of the elements of our GUI, that is the visual appearance of the GUI, can be configured by means of a *stylesheet* [K+07]. For instance, if we define the following (fragment of a) stylesheet:

```
<xsl:template name="window">
     <xsl:param name="color">blue</xsl:param>
</xsl:template>

<xsl:template name="groupbox">
     <xsl:param name="color">orange</xsl:param>
</xsl:template>

<xsl:template name="label">
     <xsl:param name="background-color">yellow</xsl:param>
     <xsl:param name="font-color">red</xsl:param>
</xsl:template>
```

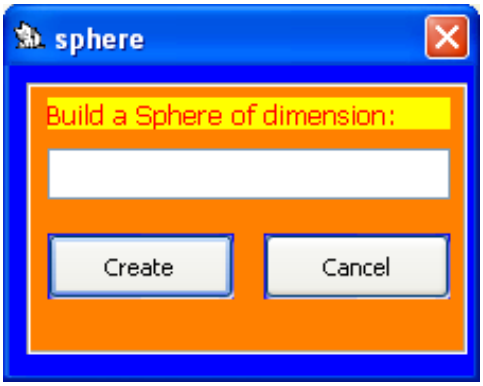

Figure 3.8: Sphere dialog with a stylesheet

the sphere dialog would have the aspect shown in Figure 3.8. That is to say, the stylesheet is used to modify the visual attributes of the elements of the GUI. If we do not provide a stylesheet the default values of the visual aspect attributes are used.

In this way, all the graphical constituents of the interface can be defined.

The next subsubsection is devoted to present how the information related to the different constituents is stored in our modules.

3.2.3.2 *fKenzo* GUI modules

A *fKenzo* GUI module is an OMDoc document that references at least two resources: a file that contains the structure of the graphical elements of the module and another one containing the functionality of those elements. In addition, a *fKenzo* GUI module can reference a file with the layout. Besides, a *fKenzo* GUI module provides some metadata (authorship, title and so on). The following conventions have been followed in the four *fKenzo* GUI construction modules and also in the computation one.

A *fKenzo* GUI module follows the schema presented for the rest of plug-ins of the plug-in framework, see Subsection 3.1.1. For instance, in the case of the `Simplicial Groups` module:

```
<code id="Simplicial Groups">
 <data format="fKenzo/GUI/structure"> simplicial-groups-structure </data>
 <data format="fKenzo/GUI/functionality"> simplicial-groups-functionality </data>
</code>
```

The first reference corresponds to the structure of the graphical constituents of that module. This document is called "<Module>-structure" where "<Module>" is the name of the correspondent module. This file is an OMDoc document. To introduce XUL

code in these documents we have used an OMDoc feature called *OpenMath foreign objects* (the `<OMForeign>` tag) which allows us to introduce non-OpenMath XML in OMDoc files. For instance, the module of Simplicial Groups includes a new menu with two menu items: one to construct Loop spaces (which has associated the shortcut "Ctrl+L" and the event `show-loop-space`) and another one to Classifying spaces (which has associated the shortcut "Ctrl+Y" and the event `show-classifying`). In this case the structure of the new menu is stored in the document `simplicial-groups-structure` with the following XUL code.

```
<OMForeign>
  <toolbarbutton type="menu" label="Simplicial Groups">
    <menupopup>
      <menuitem label="Loop Space" acceltext="Ctrl" accesskey="L"
                command="show-loop-space"/>
      <menuitem label="Classifying Space" acceltext="Ctrl" accesskey="Y"
                command="show-classifying"/>
    </menupopup>
  </toolbarbutton>
</OMForeign>
```

The functionality related to a concrete module is encoded in an OMDoc document called "<Module>-functionality" where "<Module>" is the name of the corresponding module. To this aim, we use an OMDoc feature which allows us to introduce code (in our case Common Lisp functions) in OMDoc files by means of the `code` tag. For instance, the functionality of the event called `show-loop-space` associated to the *Loop Space* menu item of the *Simplicial Groups* menu is encoded in the `simplicial-groups-functionality` document as follows.

```
<code id="show-loop-space">
    <metadata>
        <description> The event associated to the Loop Space menuitem </description>
    </metadata>
    <data format="application/fKenzo">
        <![CDATA[ (defun show-loop-space ()
                    ;; code ) ]]>
    </data>
</code>
```

Finally, the resource related to the layout is optional, and in particular the *fKenzo* GUI modules use the default layout, so they never reference any layout file. Anyway, if we want to provide a different appearance for the graphical elements of a module we can define an OMDoc document where we encode the stylesheet, which customizes the visual aspect of the elements of the module, using the same feature employed in the case of structure documents, that is the `<OMForeign>` tag.

The organization presented here allows us to deal with the design of a dynamically extensible GUI, where modules are plugged in. Our front-end becomes *extensible* thanks

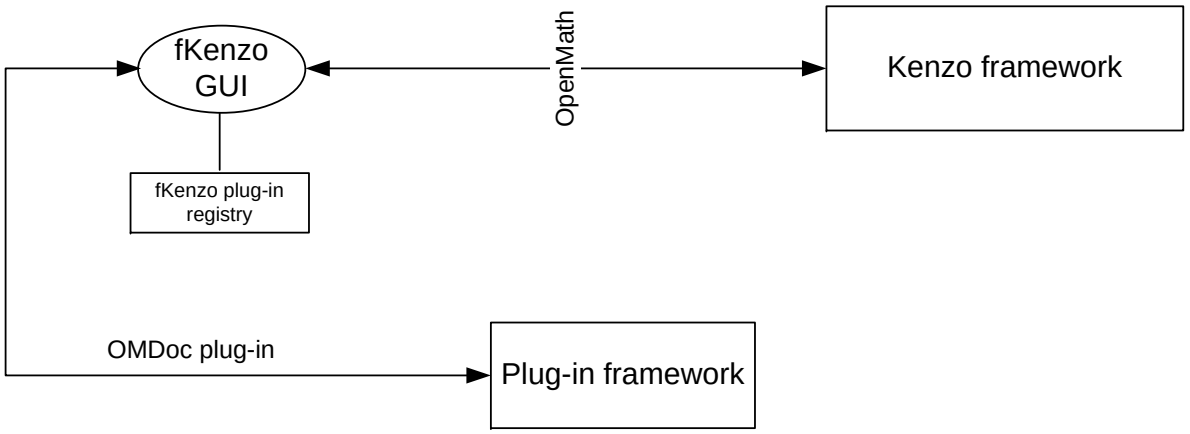

Figure 3.9: Plug-in framework and *fKenzo* GUI

to the plug-in framework since each user interface unit is encoded in a unique OMDoc file, with its inner modular organization: structure, functionality and (optionally) layout.

3.2.3.3 *fKenzo* GUI as client of the plug-in framework

To tackle the extensibility question in our user interface, the *fKenzo* GUI has been designed not only as a client of the Kenzo framework, but also as a client of the plug-in framework presented in Section 3.1. In this way, the *fKenzo* GUI can be extended in an easy way by means of modules, described in the previous subsubsection. In this context, instead of using the term *plug-in* we prefer the term *module* which is more appropriate (an application which is extended by means of modules does not have any functionality, apart from the one which allows us to load modules, if we have not added any module, as the *fKenzo* GUI; on the contrary, an application which is extended by means of plug-ins can work without adding them, as the Kenzo framework). The relations among the *fKenzo* GUI, the Kenzo framework, and the plug-in framework are depicted in Figure 3.9.

The plug-in manager (see Subsection 3.1.1) of the plug-in framework contains a module in charge of extending the *fKenzo* GUI. This module extends the GUI providing access to a concrete part of the functionality of the Kenzo framework by means of the five modules explained for the *fKenzo* GUI (the four construction modules and the computing one).

In particular, the plug-in manager of the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. This module is split in two constituents. The first one is an interpreter in charge of converting from XUL code to Common Lisp code the different graphical components. In addition if a layout file is specified, then the layout properties are applied to the graphical components. The second one associates the functionality of the event handlers to the elements defined in the structure document.

Then, it is enough to produce an OMDoc file with the suitable components, and then it can be interpreted and added in our GUI. It is exactly what happened when in Subsection 3.2.2 we described the way of working with *fKenzo*: using *File* → *Add Module* with the `Simplicial-Groups.omdoc` module sends this module to the plug-in framework. Subsequently, the plug-in manager invokes the *fKenzo* module (one of the subcomponents of the plug-in manager) that extends the user interface.

The implementation of the *fKenzo* GUI as a client of the plug-in framework shows the feasibility and usefulness of this framework.

The *fKenzo* GUI features partly owing to the implementation of the user interface as a client of the plug-in framework are listed below.

- Modularity: the GUI is organized in different modules, each one devoted to a concrete concern.
- Extensibility: the GUI can be extended by the modules. Each new functionality can be realized as an independent module.
- Flexibility: each unnecessary module can be removed and each necessary module can be loaded at run-time. Therefore the GUI can be configured in such a way that it has only the needed functionality.
- Easy to install: the installation of modules is friendly (only select a file with the option `Add Module`) from the plug-in folder.
- Internet based update: the GUI supports an update mechanism from the `Help` menu. This allows the GUI to download new modules or updates.
- Storage of configuration: the GUI configuration is automatically saved for further sessions.

3.2.3.4 Extending the Kenzo framework from the *fKenzo* GUI

The previous subsubsections have been devoted to explain how the *fKenzo* GUI can be customized by means of the plug-in framework. Moreover, the plug-in framework can also be employed to increase the functionality of the Kenzo framework as we presented in Subsubsection 3.1.2. Besides, in the same way that we wanted that the Kenzo framework could evolve at the same time as Kenzo, we also hope that the *fKenzo* GUI will be able to evolve at the same time that the Kenzo framework.

Up to now, the *fKenzo* modules that we have presented (the four construction modules and the computation one) to customize the *fKenzo* GUI do not suppose any improvement in the Kenzo framework. However, it is worth noting that the modules for the GUI not only can extend the GUI but also the Kenzo framework.

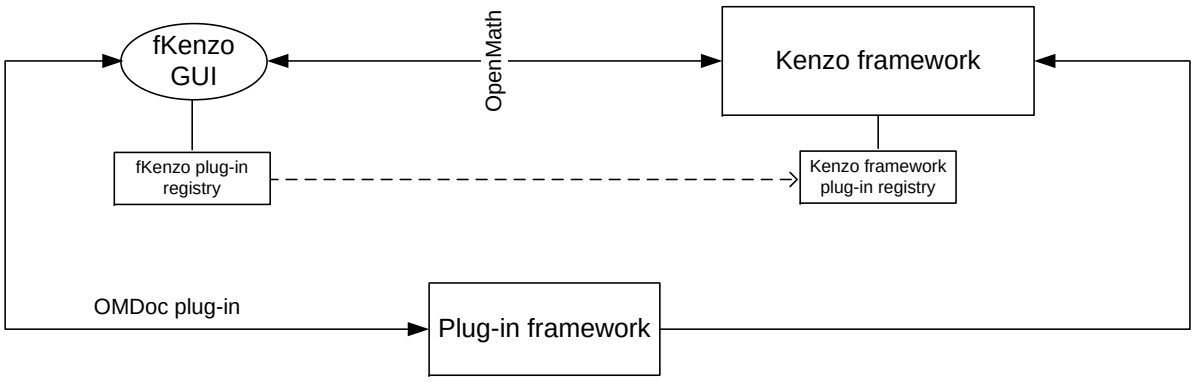

Figure 3.10: Plug-in framework, *fKenzo* GUI and Kenzo framework

Let us retake the example presented in Subsubsection 3.1.2 where a plug-in called `new-constructor` was defined to increase the functionality of the Kenzo framework by means of a new constructor. Now, following the guidelines of Subsubsection 3.2.3.2, we can define three files (structure, functionality and layout) to customize the GUI to interact with the new constructor. Finally, we define a *fKenzo* GUI module which not only references the three files (structure, functionality and layout) to customize the GUI but also the plug-in which adds new functionality to the Kenzo framework.

```
<code id="new-constructor">
 <data format="fKenzo/GUI/structure"> new-constructor-structure </data>
 <data format="fKenzo/GUI/functionality"> new-constructor-functionality </data>
 <data format="fKenzo/GUI/layout"> new-constructor-layout </data>
 <data format="fKenzo/GUI/Kf"> new-constructor-plug-in </data>
</code>
```

When the user selects this new module from the *Add Module* option, the plug-in framework will extend both the *fKenzo* GUI and the Kenzo framework. In Subsubsection 3.2.3.3, we explained that the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. We said that this module is split in two constituents but we have included a new constituent devoted to invoke the Kenzo framework module of the plug-in framework for the cases explained in this subsubsection. This last constituent is only invoked if the *fKenzo* GUI module references a Kenzo framework plug-in, in that case the Kenzo framework module of the plug-in manager is also called.

In addition, the *fKenzo GUI plug-in registry* and the *Kenzo framework plug-in registry* must be coherent in order to avoid inconsistencies in the whole system.

A high level perspective of the interaction between the two frameworks and the GUI can be seen in Figure 3.10.

This extensibility principle makes very easy to us the incorporation of experimental features to the system, without interfering with the already running modules, as we will

see in Chapters 4 and 5.

3.2.4 Interaction design

In the previous subsection we have explained the design decisions that we took to develop an extensible and modular interface, probably the most important feature of the *fKenzo* GUI from the developer perspective. However, other decisions were taken on the *fKenzo* user interface design.

3.2.4.1 Task model

The first idea guiding the construction of a user interface must be the objectives of the interaction. In *fKenzo* there is only one higher-level objective: to compute groups of spaces. This main objective is later on broken in several subobjectives, trying to emulate the way of thinking of a typical Kenzo user. Once this first objective analysis is done, the next step is to design a task model. That is to say, a hierarchical planning of the main actions the user should undertake to get his objectives. This is a previous step before devising the navigation of the user, which will give the concrete guidelines needed to implement the interface.

In our case, the two main actions of the system are: (1) computing groups, and (2) constructing spaces. Note that the second task is necessary to carry out the first one. In turn, the task of constructing spaces can be separated into: (a) constructing new fresh spaces and (b) loading spaces from a previous session. Thus, the notion of session comes on the scene. With respect to the construction of fresh spaces, once the user has decided to go for it, he should decide which type of space he wants to build: simplicial set, simplicial group, and so on. Note that the construction of a space of a type can involve the construction of other space of whether its same type or a different type. This third layer of tasks gives us the module organization of the interface, while computing produces a separated module.

The task design is organized hierarchically by diagrammatic means. See in Figure 3.11 a first decomposition layer depicted by means of a package diagram [Gro09]. Each task (each frame) is linked to auxiliary tasks (giving a horizontal dependency structure). Then, each frame is described in more detail (vertical structure) by making explicit its subtasks graph.

Task modeling provides us with both the high level modular structure and the different steps needed to reach a user subobjective. The concrete actions a user should perform to accomplish the tasks are devised in the control and navigation models.

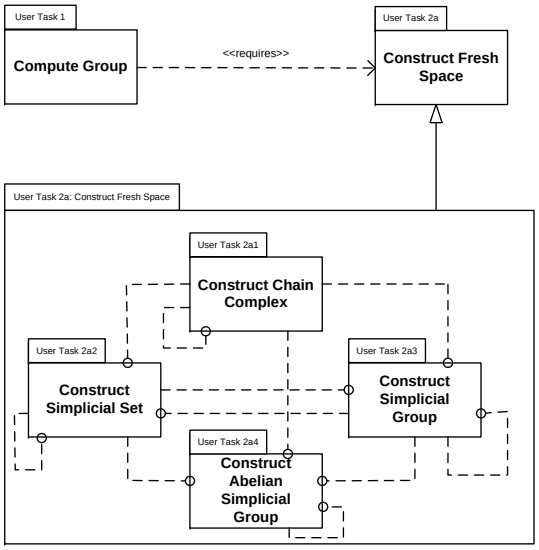

Figure 3.11: Hierarchical decomposition of the "Construct Fresh Space" user task

3.2.4.2 Control and navigation model

The design of the interaction between a user and a computer program involves well-known challenges (use of convenient metaphors, consistency of the control through the whole application, and so on). In order to avoid some frequent drawbacks we have followed the guidelines of the Noesis method (see [DZ07] for the general theory, and [CMDZ06] for the design of reactive systems). In particular, our development has been supported by the Noesis models for control and navigation in user interfaces. These graph-supported models enable an exhaustive traversal study of the interfaces, allowing the analyst to detect errors, disconnected areas, lack of uniformity, and so on, before the programming phase. Figure 3.12 shows the control and navigation submodel describing the construction of a loop space $\Omega^n X$. Different kinds of interactions are graphically represented in Figure 3.12 by different icons. For instance, selecting from a list is depicted with a form icon; directly writing an input is depicted with a pen, and so on. These pictures help the programmer to get a quick overall view of the different controls to be implemented.

Let us observe that this diagrammatic control model is *abstract*, in the sense that nothing is said about the concrete way the transitions should be translated into the user interface. In fact, in the *fKenzo* GUI this model is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through control-keys. The second style has been included thinking of advanced users, who want to use shortcuts to access the facilities of the interface. The adaptation to different kinds of users is one of the principles for design usability in [Nie94], and has been considered, as the rest of principles, in our development.

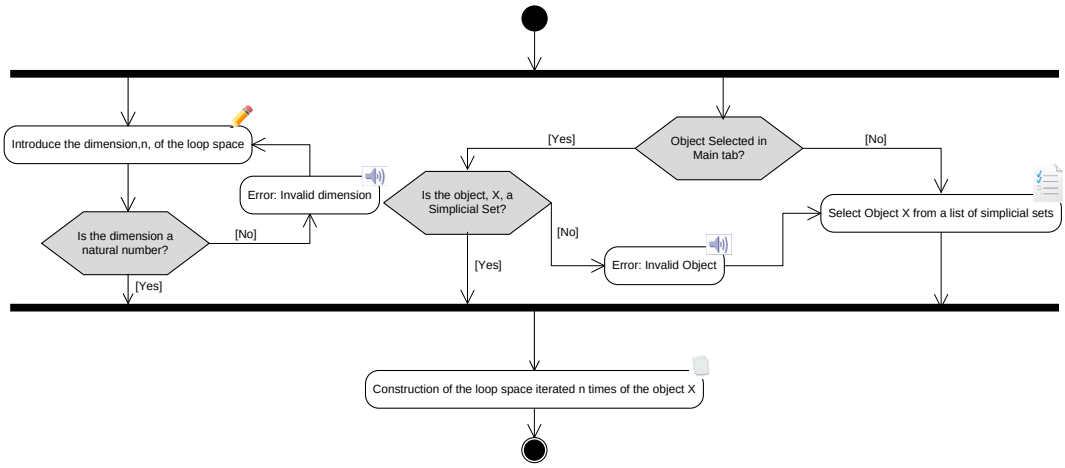

Figure 3.12: Control graph for the construction of $\Omega^n X$

3.2.4.3 Challenges in the design of the *fKenzo* GUI

User interface design is a central issue for the usability of a software system. Ideally, the design of a user interface should be done following certain rules, such as those listed in guidelines documents, see for instance [KBN04]. However these guidelines have hundreds of rules, then instead of strictly following those rules the design of a user interface abides by heuristics rules based on common sense. A small set of heuristic principles more suited as the basis for practical design of user interfaces was given in [Nie94]. We have used the nine principles given in [Nie94] for guiding the design decisions of the *fKenzo* GUI.

1. *Visibility of system status.* the *fKenzo* GUI should always keep users informed about what is going on. As we have said previously, the *fKenzo* GUI can be used to construct spaces and compute groups. In the case of the spaces constructed in the *fKenzo* GUI this first principle is achieved, since the *fKenzo* GUI shows a list with the spaces constructed in the current session to the user. Related to the computation of groups, some calculations in Algebraic Topology may need several hours, then to dealt with the *visibility of the fKenzo status*, a message informs the user of this situation when a computation is performed. Besides, the *fKenzo* GUI allows the user to interrupt the current computation and keep on working with his session.

2. *Match between system and the real world.* the *fKenzo* GUI should show results using well-known Algebraic Topology mathematical notation. This second aspect has been solved by means of combining OpenMath and stylesheets. When selecting one of the spaces of the left list of the main *fKenzo* GUI window, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. A *stylesheet* has been defined using *XSLT* [K+07]. This *stylesheet* is in charge of rendering using mathematical notation the object represented with an OpenMath instruction. In both "Session" and "Computing" tabs the results are also rendered using mathematical notation thanks to the same *stylesheet*.

3. *Consistency and standards.* A user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured. Then, the *fKenzo* GUI is *consistent* with respect to Kenzo. It is worthwhile noting that this is the most influential requirement with respect to the visual aspect of our interface. In addition to the menu bar, there are three main parts in the screen: a left part, with a listing of the objects already constructed in the current session, a right panel with several tabs, and a bottom part with the standard mathematical representation of the object selected. Thus, focus concentrates on the object (space) of interest, as in Common Lisp/Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it. It is separated by means of tabs not only because on the division among spaces and computing results (the system moves dynamically from one to the other), but also due to the capability of integrating other systems, playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns.

4. *Error prevention.* The *fKenzo* GUI should forbid the user the manipulations raising errors. The most important design decision related to this point is the use of the GUI as client of the Kenzo framework. In this way, all the enhancements included in the framework are inherited by the GUI forbidding the user some manipulations raising errors and guiding his interaction with the system.

5. *Recognition rather than recall.* The *fKenzo* GUI should minimize the user's memory load. This design principle is fulfilled thanks to the combination of stylesheets and OpenMath that are used to inform the user about the selected space. Moreover, this principle was important for the design of the dialogs used to construct spaces from other spaces and compute groups. For example, if a space is selected in the screen of the *fKenzo* GUI the dialogs used to construct spaces from other spaces and compute groups take that space by default as input, then the user does not need to select the space in the dialog.

6. *Flexibility and efficiency of use.* The *fKenzo* GUI should provide shortcuts that speed up the interaction for the expert user and also suit the needs of each user. To handle this question, the interaction with the GUI is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through control-keys used as accelerators. Moreover, thanks to the organization of the system by means of modules, a user can load the functionality that he needs.

7. *Minimalist design.* The *fKenzo* GUI should not contain irrelevant information; to that aim, the dialogs showed to the user only contain the key information.

8. *Good error messages.* the *fKenzo* GUI should indicate precisely the problems. Thanks to the use of the GUI as client of the Kenzo framework, the warnings obtained from the framework are used in the GUI. These warnings express in plain language the problem, and constructively suggest a solution.

9. *Help and documentation.* The *fKenzo* GUI should include a good documentation. Even though the *fKenzo* GUI can be used without documentation, help and documentation are provided in the *Help* menu. This help is always available, is focused on the user's tasks, lists concrete steps to be carried out, and contains both information about the use of the *fKenzo* GUI and the underlying mathematical theory.

In summary, design principles have been followed in the *fKenzo* GUI obtaining in this way a usable interface for the Kenzo system through the Kenzo framework.