Design of an UNDO Framework

Hernán Merlino, Oscar Dieste, Patricia Pesado, Ramón García-Martínez

Grupo de Ingeniería de Software Experimental. Facultad de Informática. UPM
Programa de Doctorado en Ciencias Informáticas. Facultad de Informática. UNLP
Instituto de Investigaciones en Informática LIDI. Facultad de Informática. UNLP - CIC
Laboratorio de Sistemas Inteligentes. Facultad de Ingeniería. UBA
Área Ing. del Software. Lic. en Sistemas. Dep. Desarrollo Productivo y Tecnológico. UNLa.
hmerlino@fi.uba.ar, odieste@fi.upm.es, ppesado@lidi.info.unlp.edu.ar, rgarciamar@fi.uba.ar

Abstract. This paper sets out to provide a highly automated mechanism for an undo process that can be easily built into a new or existing system. Our proposal is based on the observation that it is not necessary to store the state of objects in memory, or the commands executed by the system to perform an undo, but it is sufficient to store the input data. This greatly simplifies the design process and undo encapsulate most of the functionality into a framework similar to those existing in many object oriented programming.

Keywords. UNDO. Framework. Paterns.

1. Introduction

It is hard to build usability into a system. One of the main reasons is that usability is built into systems at an advanced stage of system development [1], when there is little time left and the key design decisions have already been taken. Usability patterns were conceived with the aim of making usable software development simpler and more predictable [2]. Usability patterns can be defined as mechanisms that could be used during system design to provide the software with a specific usability feature [1]. Some usability patterns defined in the literature are: Feedback, Undo/Cancel, Form/Field Validation, Wizard, User profile and Help [3]. The main stumbling block for applying these patterns is that there are no frameworks or even just architectural or design patterns associated with the usability patterns. This means that the pattern has to be implemented ad hoc in each system. Ultimately, this implies that (1) either the cost of system development will increase as a result of the heavier workload caused by the design and implementation of the usability features or, more likely, (2) many of these usability features (Undo, Wizard, etc.) will be left out in an attempt to cut the development effort.

The goal of this paper is to develop a framework for one of the above usability patterns, namely, the undo pattern. As its name suggests, the undo pattern aims to provide the functionality necessary to undo actions taken by system users.

Several authors have proposed alternative implementations of the undo pattern. However, these alternatives focus on particular applications remarkably: document editors [4] [5]. We aim; on the other hand, to set out a highly automated, generic solution that we believe is easy to build into both new and existing software systems. Essentially, our proposal is composed of a (partially implementation platform-dependent) mark-up language. Thanks to a suitable pre-processor, this language can relate user operations to system objects, so that it is very easy to store the current system state and return to an earlier (or later) state, if necessary.

This article is structured as follows. Section 2 describes the state of the art regarding the implementation of undo. Section 3 presents the undo infrastructure, whereas Section 4 describes the mark-up language. Section 5 shows a proof of concept of the proposed framework. Finally, Section 6 briefly discusses and presents the main contributions of our work.

2. Background

Undo is a very widespread feature, and is prominent across the whole range of graphical or textual editors, like, for example, word processors, spreadsheets, graphics editors, etc. Not unnaturally a lot of the undo-related work to date has focused on one or other of the above applications. For example, [5] and [6] have patented two methods for implementing undo in document editors within single environments.

The problems of undo in multi-user environments have also attracted significant attention. Both [4] and [7] have proposed mechanisms for using undo in distributed environments, and [8] propose a formal framework for this field.

The most likely reason for the boom of work on undo in the context of document editors is its relative simplicity. Conceptually speaking, an editor is a container accommodating objects with certain properties (shape, position, etc.). Consequently, undo is relatively easy to implement, as basically it involves storing the state of the container in time units $i, i+1, \ldots, i + n$. Then when the undo command is received, the container runs in reverse $i + n, i + n-1, i$. In distributed environments, the solution has to deal with the complexity of updates to shared data (basically, a serialization of changes).

Several papers have provided insight on internal aspects of undo, such us [9], who attempted to describe the undo process features. In [10] created an undo infrastructure and in [11] defined a selective undo.

Also, patents have been registered, like the method for building an undo and redo process into a system [12]. Remarkably, this paper presents the opposite of an undo process, namely redo, which does again what the undo previously reverted. Other author’s addresses the complexities of undo/redo as well. Thus, for example, [13] define a mechanism for managing a multi-level undo/redo system, [14] describe an Undo and Redo algorithm and [15] wrote about a method for graphically administering Undo and Redo, based primarily on the undo method graphical interface.

The biggest problem with the above works is that, again, they are hard to adopt in software development processes outside the document editor domain. The only

noteworthy exception to this is a design-level mechanism called Memento [16]. This pattern restores an object to a previous state and provides an implementation-independent mechanism that can be easily integrated into a system. The downside is that this pattern is not easy to build into an existing system. Additionally, Memento only restores an object to a previous state; it does not consider any of the other options that an undo pattern should include.

3. UNDO Pattern

Before introducing our proposal for the implementation of the undo pattern, we will describe it in detail, identifying all the undo's prominent characteristics (for example, when and how undo can be invoked, what we can expect from undo's application, etc.). This should provide a baseline for defining a set of requirements for the implementation of the undo, that is, for the *undo process*.

3.1 Common features

As viewed by an end user, the undo process has some fairly stereotype features, indisputably motivated by how commercial software implements this functionality. Essentially, the user can invoke undo at any time, and it has an immediate effect (for example, the contents of a particular field change from their value at time t+1 to their value at time t). Some applications, like MS Word, can undo not just the *last change,* but also the *latest k* or even *all changes* made.

3.2 Problem issues surrounding UNDO

Although at first glance this looks like a sound description of the undo process, a more thorough analysis shows that this description is incomplete. It fails to consider many aspects that are extremely important for defining a really useful implementation:

- First, the undo process cannot be invoked at any time. In actual fact, in none of the applications known to us is within-process undo possible invocation (that is, when the system is processing a user command); they offer only between-process invocation (that is, when the system is waiting for a user input). For example, suppose that an application is enacting a time-consuming process, such as increasing the prices of all items in a large database by 10%. The undo process cannot be invoked while the update process is executing. But, as most processes do not take too long, the user has the impression that undo is always available. Even so, in a competitive undo implementation we should be possible to stop the ongoing process and return to the state immediately before the operation was invoked (DB update, for example). Note that, in this case, undo is very similar to HCI's cancel pattern.
- Applications limit the possibility of undoing actions in long/complex processes. Database updates like the above are a very common case in point. Typically, once the DB transactions have been committed, undo is no longer available (all previous

states are deleted). Any implementation aiming for widespread use should take into account this type of system operations.

- Undo is mostly linear (that is, the actions taken are undone in strictly reverse order), even if the work completed by the user is not. A fairly common example is when a word processor user makes a change to paragraph X, goes on to modify paragraph Y and then tries to undo the changes in X. This is impossible unless you also undo the changes made to Y. A flexible undo implementation should account for this possibility.
- Finally, undo is often applied in single-user applications, which is the case, for example, of most text editors, spreadsheets, graphical editors, etc. However, undo should be equally applicable to multi-user applications, where any user can modify shared data at any time. Think, for example, of the Google Docs editor. It looks like this type of applications will be increasingly important in the near future.

3.3 UNDO process requirements

The above sections suggest a set of features that a comprehensive undo implementation should have. These features are listed in table 1.

Table 1. Undo features

No.	Feature	Detail
1	**Invocation time**	*Within-processes*
		Between-processes
2	**Invocation source**	*User*
		System
3	**Scope of the action to be undone**	*Total*
		Partial (limited)
		Last change
4	**Application mode**	*Linear*
		Non-linear (selective)
5	**Complexity of undoing the change**	*Low (idempotent)*
		High (command)
6	**Environment**	*Single-user*
		Multi-user

1. **Invocation time**: this feature refers to the process time at which the UNDO can be invoked (executed). Ideally, undo should be able to be executed *at any time*, although it could be best to reduce this flexibility under some circumstances (for example, to reduce the undo framework's system overhead).
2. **Invocation source**: this feature refers to where the undo process can be invoked. This source can be a user operating the application, or another system requesting the undo process invocation.
3. **Scope of the action to be undone**: this feature defines the extent to which the executed action will be undone. Actions can be fully or partially reverted. Partial undo is the reversal of the last or latest x executed steps of the action.
4. **Application mode**: this refers to how the changes will be reverted. If they are executed in the opposite order to which they were made, undo is referred to as linear, whereas if it is possible to select which change to apply, it is known as non-linear undo.

5. **Complexity of undoing the change**: this feature is designed to catalogue the undo processes according to their complexity or their features. Undoing a DB change is a complex process which requires a completely different strategy than take back, for example, the assignment of a new value to an object's attribute.
6. **Environment**: this feature catalogues the undo process according to whether it is applied in a single- or multi-user environment.

4. UNDO framework

In this section we describe our proposal on the design of the undo. Specifically, we created a framework (set of classes) that implements the six characteristics described in the previous section. In what follows, I will describe the structure and functioning of the framework, but for simplicity associated with the extension of the paper, it will develop features, 1 and 6, which discusses only briefly in the conclusions, and only part of the face feature 5.

4.1 Context of the UNDO framework

The most common alternative to develop a process to undo is to save the states of objects that are likely to be applied to undo a process before they undergo any operation that alters the value of any of its attributes. This method has one obvious advantage: you can go back the system without the need for a particular process for this; it is only necessary to evict the objects that are now in memory and replace it with objects that have been previously saved.

This approach represents a simple mechanism for the implementation of the undo, but it brings some disadvantages: Firstly, to save all objects that can be used to undo a process could generate an excessive workload for the system. On the other hand if the system already exists and you want to undo facility, you should have a very detailed knowledge of the application to find out what objects are to preserve and what not.

A second alternative to implement a process by storing undo the operations performed by the system rather than object's changes. In this case, the undo would run in reverse order of the inverse operations.

The approach we propose is based on the latter strategy, but with a particularity that greatly simplifies the design. The key is: the only command processed by an arbitrary system software and that are relevant to undo the process, are the updates that cause the data model (e.g., the introduction of an entry in a field of a form that causes the update of an attribute of an object, the introduction of a backspace character as to cause the removal of one letter in a document object, etc.). In the vast majority of cases, these updates are idempotent, e.g., the effects of the entry do not depend on the history of states. This is the example of the form above (though not, for example, the word processor). When the updates are idempotent, it is not necessary to store the state of the object model or operations, but only the list of entries in the system. In cases where the characteristics are not idempotent (as the example of the word processor), this strategy is not valid and must be used in the original strategy (store the command and apply the inverse in the case of undo). However, it seems

clear that the vast majority of the commands are executed by a system of the first type, while the second is rather the exception.

Therefore the approach we propose has several advantages: (1) The simple data entry can be treated in a completely automatic and transparent (2) is not necessary to handle the complexity of objects in memory (3) is only necessary to know the logic of the system for the command and (4) Finally, our approach allows the design of an undo framework as described in the next section, quite independent of the application and therefore highly reusable.

4.2 Undo framework

In this paragraph describe the process raised by the implementation of undo. For simplicity, we will focus on idempotent operations, but also discuss how to undo commands. Our intention is to create a redistributable package and friendly, like the existing frameworks so often used in object oriented development.

4.2.1 Application's Architecture

This application is based on a set of layers. We try to build a common application, with the use of a chain of filters to perform common operations of the user interface and a controller to unify the user requests a single point. In figure 1 shows a sequence diagram that represents the application which will be added to the Undo Framework.

4.2.2 Undo Framework Architecture.

The structure of the Undo Framework is based on the implementation of the Intercepting Filter pattern and Command. The first of these patterns can create a chain of filters to perform common independent tasks for pre and post-processing and provides a simple mechanism through a filter to recognize when a user requests an undo. Command pattern avoids coupling between the UI and the pattern, allowing you to work with multiple objects that share the same command and facilitating future modifications and extensions of the system. In figures 2 and 3 is described the framework of Undo can be viewed as patterns are used Intercepting Filter and Command. Here you can see how different filters have been added for different types of actions that can be treated by the markup meta-language. This design decision has been taken for the following reasons: (a) to add a filter for each new way to go back an action and, (b) allow optimize the performance of the system through the choice of filter to apply. When the filters are evaluated, control passes to the standard command for execution. The pattern ends when the command adds a record in the temporary persistence mechanism for handling Undo. For reasons of simplicity has avoided detailed mechanism of persistence. Figure 4 details as it is presented to the user requesting an Undo process, the information available to restore.

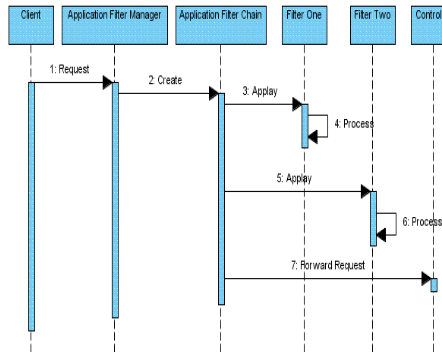

Fig. 1. Sequence Diagram. Common Application

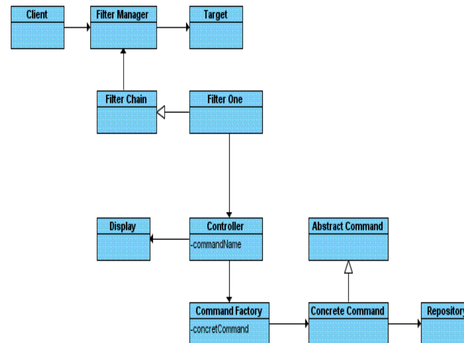

Fig. 2. Class Diagram Undo Framework

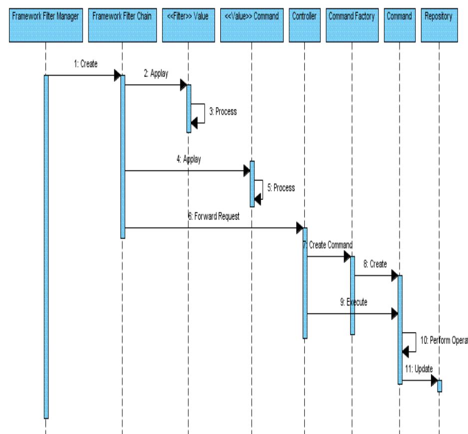

Fig. 3. Sequence diagram. Undo Framework

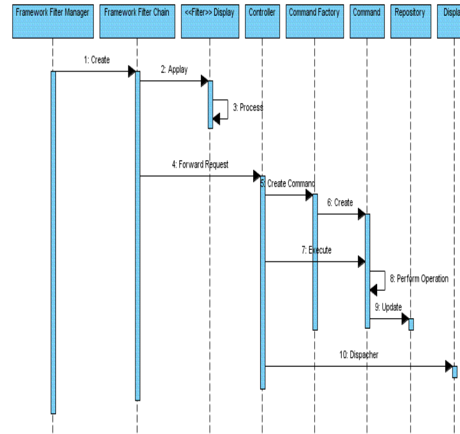

Fig. 4. Sequence Diagram. Undo Request.

The diagram shows the addition of the filter "" <<filter>> Display "to the chain of filters. This filter is responsible for recognizing the meta-language associated with the ruling request Undo from the user. Once acknowledged this request, control passes to the standard command that is responsible for creating the command to get information stored in it temporarily persistence mechanism and sends the user the defamation in the corresponding view.

4.2.3 Connection between Undo Framework and Application

Figure 5 shows the aggregate of the existing application (see Figure 1) by including a specialized filter ("<<filter>> Undo Framework") in the existing chain of filters in implementation.

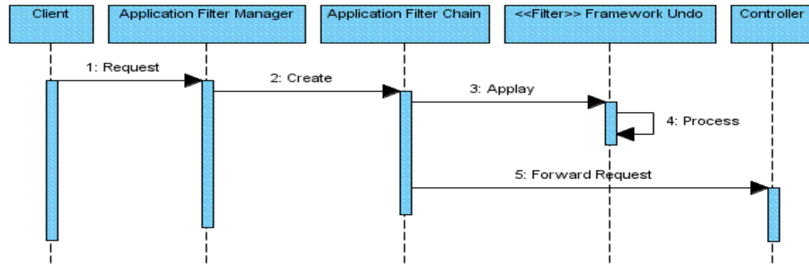

Fig. 5. Sequence Diagram. Undo Framework in Application.

4.3 Identification of input data

The design proposed in the previous section has one major deficiency: no way to determine what input data correspond to which input field and then to re-enter data to undo. We must implement a mechanism by which the framework of undo can automatically recognize the source of the data entering the system to store them properly indexed. The proposed design, which is undo implemented in a filter added to the chain of input filters, provides a fairly simple strategy to achieve this goal. Simply add to the input of any sign that identifies its origin. That label would be recognized by undo the filter, which would store the data associated in the mechanism of persistent undo. Subsequently, the label would be removed from the input data, so all the subsequent process will run as usual without the business layer of the application.

5. Proof of Concept

To perform a proof of concept has been implemented Undo process in a grid of data (very simplified) which can be shared between users. Figure 6 shows a snapshot of this application. The grid lets you enter numbers and perform simple calculations, following the usual conventions of a spreadsheet. This application has been made on the Google App Engine platform. In this application, communication between the interface layer and the business is conducted by exchanging data in XML format. The interface has a function “sendData” implemented in JavaScript on the client side, which takes all the values are loaded into the grid, add in an XML document and sends to the server. The XML document can have any format. In this proof of concept, we used a very simple format that is shown in Figure 7.

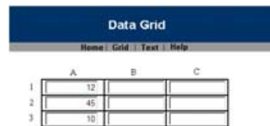

Fig. 6. Application’s screen.

```
<spreadsheet>
        <column>
        <row >...<row/>
        ...
        <column/>
<spreadsheet/>
```

Fig. 7. XML Document

In the application server all the requirements (XML documents) are served by the function “webapp.WSGIApplication” (see Figure 8) that acts as a controller of an MVC pattern. This function is responsible for distributing the notice to the appropriate business layer (in this case, GridExample). Then, this function takes control and performs the necessary operations (see Figure 9) to meet the requirement of the client.

```
application = webapp.WSGIApplication(
[('/', MainPage),
 ('/grid', GridExample),
debug=True)
```

Fig. 8. Function webapp.WSGIApplication

```
class
GridExample(webapp.RequestHandler):
  def __init__(self):
   webapp.RequestHandler.__init__(self)
    self.methods = RPCMethods()
  def post(self):
   args =
simplexml.loads(self.request.body)
   . . .
```

Fig. 9. Function `GridExample`

To add functionality to undo this application only two changes are necessary. The first is to add labels to identify the input fields, in line with the specifications in section 4.3. In this proof of concept, these signs (which we implemented as named parameters item) can be added easily, since the interface-business communication is done via XML documents and these documents are easily adaptable, as shown in figure 10. Since the XML document is composed on the basis sendData function, only be necessary to slightly modify the code for this function. The second change is to link the framework to undo the application, put the filter in the corresponding data path. To do it is necessary to make a simple addition to the GridExample function, as shown in figure 11. FilterUNDO function takes the XML document, store the values of the entries and fields and rebuilt document to original version, so the application can continue its normal course.

```
<spreadsheet>
        <column>
        <row item=”cell_1”>...<row/>
        ...
        <column/>
<spreadsheet/>
```

Fig. 10. XML Document with label

```
class
GridExample(webapp.RequestHandler):
  def __init__(self):
   webapp.RequestHandler.__init__(self)
    self.methods = RPCMethods()
  def post(self):
    rawData                                =
simplexml.loads(self.request.body)
   args = filterUNDO(rawData)
   . . .
```

Fig. 11. GridExample function modified

The above modifications allow the marking of input data, so as to allow their recognition and storage of the undo filter. Commands may be marked in a similar way. Finally, it is only necessary to re-enter information previously entered by the user in the application at the instant t-2, as described above.

6. Conclusions

In this paper we proposed the design of a framework for incorporating functionality into applications software undo arbitrary. One of the salient features of

this framework is the type of information stored to be able to undo the operations performed by users. Instead of storing the state of objects in memory, or the commands executed by the system, store data, which simplifies greatly the impact that the incorporation of exercise in the application framework.

There are two aspects that we have not addressed in this article, which features are the 1 and 6 of Table 1. The reason is the complexity of these features, which requires a long explanation. However, both are perfectly implemented within the framework exposed. The features 1 require only the possibility of stopping a process during execution and reverse the changes made. This can be achieved easily using modern control structures such as catch and try-threaded processes; although depending on the architecture used complications arise that must be solved case by case basis.

7. Reference

1. Ferre, X., Juristo, N., Moreno, A., Sanchez, I. 2003. *A Software Architectural View of Usability Patterns*. 2nd Workshop on Software and Usability Cross-Pollination (at INTERACT'03) Zurich (Switzerland)
2. Ferre, X; Juristo, N; and Moreno, A. 2004. Framework for Integrating Usability Practices into the Software Process. Universidad Politécnica de Madrid.
3. Juristo, N; Moreno, A; Sanchez-Segura, M; Davis, A. 2005. Gathering Usability Information through Elicitation Patterns.
4. Qin, X. y Sun, C. 2001. *Efficient Recovery algorithm in Real-Time and Fault-Tolerant Collaborative Editing Systems*. School of computing and Information Technology Griffith Univesity Australia.
5. Bates, C. y Ryan, M. 2000. *Method and system for Undoing edits with selected portion of electronic documents*. PN: 6.108.668 US.
6. Baker, B. y Storisteanu, A. 2001. *Text edit system with enhanced Undo user interface*. PN: 6.185.591 US.
7. Abrams, S. y Oppenheim, D. 2001. *Method and apparatus for combining Undo and redo contexts in a distributed access environment*. PN: 6.192.378 US.
8. Abowd, G.; Dix, A. 1991. *Giving UNDO attention*. University of York.
9. Mancini, R., Dix, A., Levialdi, S. 1996. *Reflections on Undo*. Universidad de Roma.
10. Burke, S. 2007. *UNDO infrastructure*. PN: 7.207.034 US.
11. Korenshtein, R. 2003. *Selective UNDO*. PN: 6.523.134 US.
12. Keane, P. y Mitchell, K. 1996. *Method of and system for providing application programs with an UNDO/redo function*. PN:5.481.710 US.
13. Nakajima, S. y Wash, B. 1997. *Multiple level Undo/Redo mechanism*. PN: 5.659.747 US.
14. Li, C. 2006. *UNDO/redo algorithm for a computer program*. PN: 7.003.695 US.
15. Martinez, A. y Rhan, M. 2000. *Graphical Undo/Redo manager and method*. PN: 6.111.575 US.
16. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison- Wesley.