A Use Case-Oriented User Interface Framework

Eliezer Kantorowitz Alexander Lyakas Artur Myasqobsky
Computer Science Dept.
Technion—Israel Institute of Technology
32000 Haifa, Israel
kantor,lyakasal,artiom@cs.technion.ac.il
http://www.cs.technion.ac.il/~kantor/

Abstract

Interactive computer applications are sometimes specified by their use cases. These specifications are often formulated in a natural language to enable domain experts, that are not familiar with formal notations, to validate their usefulness. A use case-oriented application framework facilitates manual translation of the natural language specifications into high level code, whose equivalence with the specifications is easy to establish. The purpose is to reduce the costs of both verification and coding. A previous framework of this kind achieved its high level by providing most of the graphical user interface (GUI) code. However, the automatically produced GUI was not always satisfactory. This study reports on advances achieved by more intelligent user interface construction framework. The study introduces a new kind of user interface component, called use case displayer*, that enables an almost automatic generation of the user interface of the entire application. The framework was tested in a student laboratory, where it reduced the implementation effort, leaving most of the time (10 of the 15 available weeks) for requirements elicitation, specification development and validation. As expected, the designs produced had a higher level of usability than found in comparable student laboratories. At this stage the framework is useful for student laboratories. More research is required to assess its suitability for industrial use.*

1. Introduction

One possible way to reduce the high costs of software development is to employ a framework, which is a set of reusable software components and a methodology for construction of programs from these components. The components and methodology of the framework are designed to produce software architectures with some useful software engineering properties. The $\mathcal{SI}$ framework, introduced in [3, 4], was thus designed to produce easy to verify architectures. It was designed by looking at state of the art software development processes, such as the Unified Software Development Process (USDP) [1]. In the following we discuss only the USDP process, as it is considered as a representative for a number of similar software development processes.

The USDP process may be divided into two major phases. The first *requirements and specification* phase comprises of requirements elicitation and a detailed specification of the system to be constructed. In USDP the system is specified by all its *use cases*. A use case is a single application of a system. An example of a use-case is the registration of a book loan in a library information system. The use cases are specified in a natural language, often English, and are accompanied by detailed drawings of the graphical user interfaces (GUI). The use of natural language and GUI drawings enable domain experts and user interface experts, that may not be familiar with formal notations, to *validate* the specifications. The purpose of the validation is to ensure the *usability* of the system—that it meets the users' needs, i.e., that the requirements and specifications are what the users need. The validation checks also that the specified system's user interface is easy to learn and to use, and enables the users to accomplish their work in a pleasant way with a minimal human effort.

In the second phase of USDP process, the *construction* phase, a system is developed that implements the use case specifications of the first phase. This phase includes a *verification* that the constructed system, i.e., that its code, implements the use case specifications precisely. Since the use case specifications were validated for their usability, the verified code will retain this usability. The USDP process is thus designed for developing systems with good usability properties.

The verification of the produced code may be done in any a number of different ways. One way is a testing of all

Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)
0-7695-2047-2/03 $ 17.00 © 2003 IEEE

the specified use cases. In such a testing, one must identify for each use case, all the kinds of possible scenarios, and develop test data for each one of these scenarios. Such a testing involves a considerable amount of work. There is also a risk that some of the scenarios may not be discovered and that the testing will therefore not be complete. Formal verification is another verification avenue.

In the second phase of the USDP process (the construction phase) the natural language use-case specification from the first USDP phase (the requirements and specification phase) are translated into diagrams in the Unified Modelling Language (UML) [2]. These use case diagrams are analyzed and further UML diagrams, that describe the system from different points of view, are developed. These diagrams represent an abstract model of the system, that is expected to facilitate the development of a consistent design. It is necessary to verify that these UML diagrams are consistent with the verbal use case specifications. Furthermore, the consistency between the different UML views of the model must be verified. The equivalence of the produced code with model must also be verified. These verifications are facilitated by current UML CASE (Computer Assisted Software Engineering) tools. These tools assist in drawing UML diagrams and in checking the consistency between them. Some CASE tools can also produce a part of the code automatically from a model, such that only the equivalence between the model and the hand coded part must be verified. The bottom line is that the development of the system in the second phase of the USDP Process and its verification is labor intensive and involves various error possibilities.

In order to reduce the high costs of the second phase of the USDP process, an attempt was made in [4] to develop a framework, that provides some of this code in ready made reusable components. The experimental $\mathcal{SI}$ framework was designed to enable a direct manual translation of the English use case specification into Java code employing $\mathcal{SI}$ classes. In one experiment each English statement in the use case specification was on average translated into 2.5 Java statements. The Java code produced with $\mathcal{SI}$ resembled the English language use case specification, such that it was quite easy to establish the equivalence between the specification and the code. This result was achieved partly by hiding the considerable amount of code required to implement the user interface.

$\mathcal{SI}$ was geared to the implementation of simple information systems. The result of gearing to information systems and hiding the user interface code was a *data-centered* abstraction, i.e., a considerable part of the code was for data input and output, as well as database manipulations using the concise SQL language. This enables $\mathcal{SI}$ developers to concentrate on the data without being too much distracted by user interface technicalities. The $\mathcal{SI}$ developer has only to select one of a number of offered *interaction styles*. An interaction style [9] is a ready made specification of the geometrical properties of the various kinds of GUI controls.

$\mathcal{SI}$ was tested by manufacturing a number of information systems. It was realized that its methods for GUI construction were in some cases insufficient flexible to produce optimal GUI's. A new $\mathcal{SI}+$ system was therefore developed to improve this and some other points. In this paper we report on the more powerful GUI facilities developed for $\mathcal{SI}+$.

2. The Structure of $\mathcal{SI}+$-based Applications

$\mathcal{SI}+$ was primarily designed for construction of information systems. The central principle is to implement each use case by a separate class that is an extension of the $\mathcal{SI}+$ `UseCase` class. This class contains the code obtained by the manual translation of the English language use case specification. This code activates objects belonging to classes that are either provided by the $\mathcal{SI}+$system or by the developer of the system.

The $\mathcal{SI}+$ system provides classes with GUI implementations, database access code and an online help system. The developer is expected to write a help file for each use case in accordance with the rules of the help system. Thus an application, constructed in this way is essentially a set of distinct use case classes and classes that serve them. The integration of the distinct use case implementations into one application is done in a typical information system fashion, i.e. through a logically single shared relational database (Fig. 1). The values stored in this database are considered as the state of the system. The database is only updated by SQL queries of the different use cases. As usual in information systems, the developer is expected to formulate the update transactions such that the database consistency is retained. Relational database management systems (RDBMS) usually provide a serialization-based concurrency mechanism. This means that the designer of a use case need not know anything about the other use cases, and in general about any other users, that operate the database at the same time.

The developer of an $\mathcal{SI}+$-based application specifies the GUI of a particular use case by a list of its controls and by the name of the interaction style to be employed. The geometrical shape of the controls, their color etc., as well as their precise location in the rectangular area (pane), allocated to the GUI of the use case, are all determined automatically by the employed interaction style. The GUI classes of $\mathcal{SI}+$ are based on the classes of the Java language Swing packages. The $\mathcal{SI}+$ classes add automatic computation of some of the details and enable the $\mathcal{SI}+$ programmer to work at a higher level of abstraction than the programmer, that employs only the facilities of the Java language.

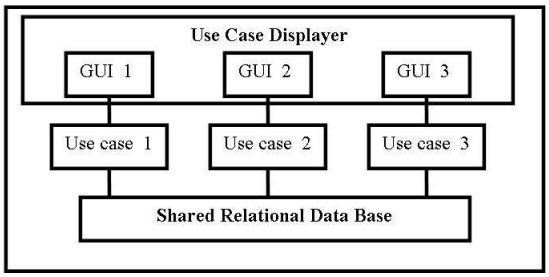

Figure 1. The structure of an $\mathcal{SI}+$-based application

The GUI's of the different use cases are integrated into the application's GUI, such that the users get an interface to the entire system (Fig. 2). To enable this, we developed for $\mathcal{SI}+$ a new kind of a GUI element, which we call *use case displayer* (often shortened to *displayer*). The displayer is an environment, that enables the user to execute the different use cases of an application and view their online help. Graphically a displayer appears as a kind of a window on the screen, containing the GUI's of all the use cases of the application. The geometrical layout and location of the different use case GUI's in the displayer are computed automatically by the interaction style specified for the displayer. $\mathcal{SI}+$ provides a number of ready made interaction styles that may be employed. If the appearance of the displayer is not satisfactory, it can be easily changed by specifying another interaction style.

Fig. 2 and Fig. 3 show two different displayers for the same single MP3 Manager application. Fig. 2 presents a displayer for the MP3 Manager application, having a desktop-like area (the right side of the figure) to visualize the use cases. The window in the top-left corner of the displayer presents all the use cases available in the MP3 Manager application: 'Manage MP3 List', 'Play MP3 Files', 'Export Playlist', 'Rename MP3 Files' and 'Edit ID3 Tag'. The latter four use cases are grouped into a package named 'Actions'. The displayer presents this package in the top-left corner window as a subtree, whose nodes are the package's use cases.

Double-clicking on the node of a use case displays the GUI of this use case in the displayer's desktop. In Fig. 2 the user works with two use cases: 'Manage MP3 List' and 'Edit ID3 Tag'.

The tree in the top-left corner also has a subtree named 'Help'. The help subtree contains nodes for all the use cases of the application. Double-clicking on a node of a use case presents the online help for this use case in the bottom-left window of the displayer. Currently the user has chosen to read the help for the 'Edit ID3 Tag' use case.

Fig. 3 presents the same MP3 Manager application using a different interaction style, i.e., a different use case displayer has been picked. The use cases of the 'Actions' package are now visualized using a tabbed pane. No change to the code implementing the use cases was required. The online help in this displayer (Fig. 4) employs the JavaHelp [8] packages.

The code, presented in Fig. 5, is typical for starting $\mathcal{SI}+$ applications. The programmer picks one of the available displayers and instantiates it, registers the application use cases with the displayer (grouping some of them in packages), instructs the displayer to build online help and, finally, makes the displayer visible. The high level of abstraction of the code, i.e., lack of details, should be noted.

3. $\mathcal{SI}+$ Database Management Support

$\mathcal{SI}+$ facilities for database access are based on the Java language classes in the Swing and JDBC packages. The $\mathcal{SI}+$ classes add some intelligence and solve automatically a number of problems such as correct support of the data types of the SQL language and facilitating of performing database transactions. Using the high level $\mathcal{SI}+$ classes instead of the underlying Swing and JDBC classes saves programming effort and reduces the possibilities for programming errors.

One such useful area regards the $\mathcal{SI}+$ facilities for automatically displaying the tables obtained by SQL queries. When using the facilities of Java, the results of such a query are stored in a `ResultSet` object. The programmer, that wishes to display the results in a table must write a piece of code for displaying a table with data from the `ResultSet`. Using $\mathcal{SI}+$ table models, this is done automatically.

Fig. 6 presents a code that executes an SQL query and presents its results in a table shown in Fig. 7. The code employs the $\mathcal{SI}+$ `SIMutableRSTableModel`. The `db_proxy` variable is an instance of $\mathcal{SI}+$ `DBProxy` class, that hides most JDBC details for accessing databases. The SQL query in this example retrieves the details of overdue books returned on a particular date. The conciseness and high level of abstraction of this code is remarkable.

$\mathcal{SI}+$ provides a further table model named `SIUpdateableRSTableModel`, which enables the application user to edit values of a table or a view from a database. In addition, when presenting a table to the user, some columns can be specified as non-editable or invisible. The rationale for setting some columns invisible is that these columns are not needed by the user and their presence can be distracting.

The idea of direct presentation of `ResultSets` in a table is not new. For example, [5] suggests a factory creating `TableModels`, that encapsulate a single `ResultSet`.

These table models of [5] are, however, not type-aware, as the class of all columns is assumed to be `String`, and provide no control over cells' editability. These problems are mended in $\mathcal{SI}+$.

4. $\mathcal{SI}+$ Evaluation

The evaluation in our software laboratory considers both the value of $\mathcal{SI}+$ for training computer science students and its value as a software engineering tool. The evaluation is done through a number of student projects, which included information systems for football game scheduling and for flight travel planning. We employed in addition a face recognition-based security system project and a chip controller project. Both latter projects are not information systems, but have a considerable amount of GUI.

We expected $\mathcal{SI}+$ to be useful for training in system development, because the very high level $\mathcal{SI}+$ classes considerably reduce the amount of programming required. The students are therefore expected to use most of their time on requirements elicitation and careful design of use case specifications. Out of the 15 weeks of the semester 10 weeks were allocated to requirements elicitation, specification development and validation. For the purpose of validation the students used $\mathcal{SI}+$ to implement a prototype of the system to be validated with potential users. Usually the students employed their friends for this validation. The preparation of the use case specifications included the design of the scenarios to be employed in final testing of the code. The preparation of the test scenarios at this early stage helped to reveal problems in the use cases.

The remaining 5 weeks of the semester are for the implementation, testing, verification, evaluation and reporting. The verification involved comparing the high level $\mathcal{SI}+$-based Java code to the verbal use case specification. This code verification can reveal serious problems and it is therefore profitable to do it before the testing. The documentation of the project is developed throughout the process. The students have to submit three reports during the semester: requirements and the validated use case specification, database design and the final system. The students were also asked to validate and evaluate their system. Each project is done by only 2–3 mostly 3rd year Computer Science students, who have already taken an introductory Software Engineering course. In such introductory courses students develop small software projects, employing software developing processes similar to USDP [1]. Developing projects in these courses requires more analysis, design and coding than in our framework-based laboratory. Therefore, the USDP courses leave less time for requirements elicitation and specification development. The instructors of our laboratory felt, that the students produced designs with better usability than we have observed in the USDP laboratories. A further reason for emphasizing the requirements elicitation and validation is the observation, that requirements faults can be extremely damaging for the software development [6].

It seems, that the ease of implementation with $\mathcal{SI}+$ is due not only to the high level of abstractions provided by the $\mathcal{SI}+$ software components, but also by the relatively simple structure of $\mathcal{SI}+$-based applications (Fig. 1). After that the schema of the relational database has been established, the different use cases may be implemented independently of each other, since they almost always communicate with the database only and not with other use cases. The implementation thus involves the writing of a use case class for each use case. These classes are serviced by the ready made $\mathcal{SI}+$ classes for GUI and database access. In few cases further classes had to be written. In the relatively simple student information systems a considerable part of the processing could be accomplished by SQL queries. The distinct use cases are combined into one application through the database and through the use case displayer. Due to the quite simple structure of $\mathcal{SI}+$-based applications, the students didn't have to invest much time into software design. It seems, in conclusion, that $\mathcal{SI}+$ comes quite close to the framework ideal of [7], where the programmer has only to write scripts that put together the ready-made components.

The student projects demonstrated that $\mathcal{SI}+$ enabled many different kinds of GUI. One reason for this power is the construction of GUI by the nesting of panes having different interaction styles. Another area, where $\mathcal{SI}+$ has better facilities than the Java's Swing GUI packages, is in the automatic layout facilities of the components of the GUI (Fig. 2). This saves developer time and the results seen in the laboratory are in our opinion satisfactory. A further possible advantage of our approach is that using the same interaction styles in the different parts of the application in similar roles may give the user the impression of a coherently designed system.

4.1. Open Problems

We do not know how the experiences gained with small student projects scale up to large systems. Experimentation with large systems is planned.

$\mathcal{SI}+$ was designed primarily for independent use cases, where the programmer, that writes or modifies them, need not know anything about other use cases. $\mathcal{SI}+$ enables, however, also the implementation of use cases that are dependent on each other by employing the Mediator design pattern [10]. The resulting code in that case is usually quite complex and difficult to verify. Use case dependency should be avoided when possible.

The experiments showed, that $\mathcal{SI}+$ met to a considerable extent the goal of being able to produce GUI's of nearly

any desired kind. Unfortunately, this required in some cases more detailed coding than we liked. We believe, however, to improve on this issue in the next version.

$\mathcal{SI}+$ enables only the construction of stand-alone applications working on a single machine. Many current real life systems are, however, Internet based multi-user applications. The nature of such Internet systems is different than of a stand-alone system. To get a complete evaluation of our approach, an Internet system version of $\mathcal{SI}+$ must be developed and tested.

The evaluation of the student experiments was based on the experience and feelings of the laboratory instructors. Controlled behavioral experiments may provide deeper insight on how the $\mathcal{SI}+$ package influences the ability of the programmers to produce high usability systems.

5. Conclusions

The study achieved to a considerable extent its goal of being able to produce many different kinds of GUI in a nearly automatic fashion. This was achieved by introducing a number of innovations. The most interesting is the use case displayer, which provides an automatic layout of the GUI of the entire application. To the best of our knowledge this level of automation has not been achieved in previous systems. Further GUI innovations were the ability to nest panes with different interaction styles and new layout algorithms beyond those provided in Java's Swing packages. Some of the power and possibilities of these facilities are illustrated in Fig. 2 and 3.

In addition to nearly automatic GUI generation $\mathcal{SI}+$ also provided some high level database access facilities, especially its automatic table display components, that considerably reduced the implementation effort and the possibilities for implementation errors. The reduction of the implementation and verification effort by the $\mathcal{SI}+$ framework enabled the students of the software laboratory to invest most of their time, i.e., 10 of the 15 weeks of the semester, on requirements elicitation, specification development and validation. The result was that the students designed applications with a higher level of usability than we have observed in comparable laboratories. More research is, however, required in order to clarify the degree to which the methods employed in this research are applicable for the development of larger systems.

References

[1] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[3] E. Kantorowitz, and S. Tadmor, "A Specification Oriented Framework for Information System User Interface", Workshop of Object Oriented Information Systems 2002 (OOIS'02), Springer LNCS 2426, 2002.

[4] S. Tadmor, *A Framework for Interactive Information Systems*, M.Sc Thesis, Technion—Israel Institute of Technology, 2002.

[5] D. Flanagan, "Making SQL Queries with JDBC and Displaying Results with Swing", O'REILLY JAVA, 2000. `http://www.oreillynet.com/pub/a/oreilly/java/news/javaex_1000.html`

[6] T. Korson, "Constructing Useful Use Cases (Managing RequirementsPart 3)". `http://www.korson-mcgregor.com/publications/korson/usecase3/`

[7] D. Roberts, and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", presented at PLoP '96. `http://st-www.cs.uiuc.edu/~droberts/evolving.pdf`

[8] JavaHelp product information. `http://java.sun.com/products/javahelp/`

[9] E. Kantorowitz, and O. Sudarsky, "The Adaptable User Interface", Communication of the ACM, 32, (Nov 1989), 1352-1352.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

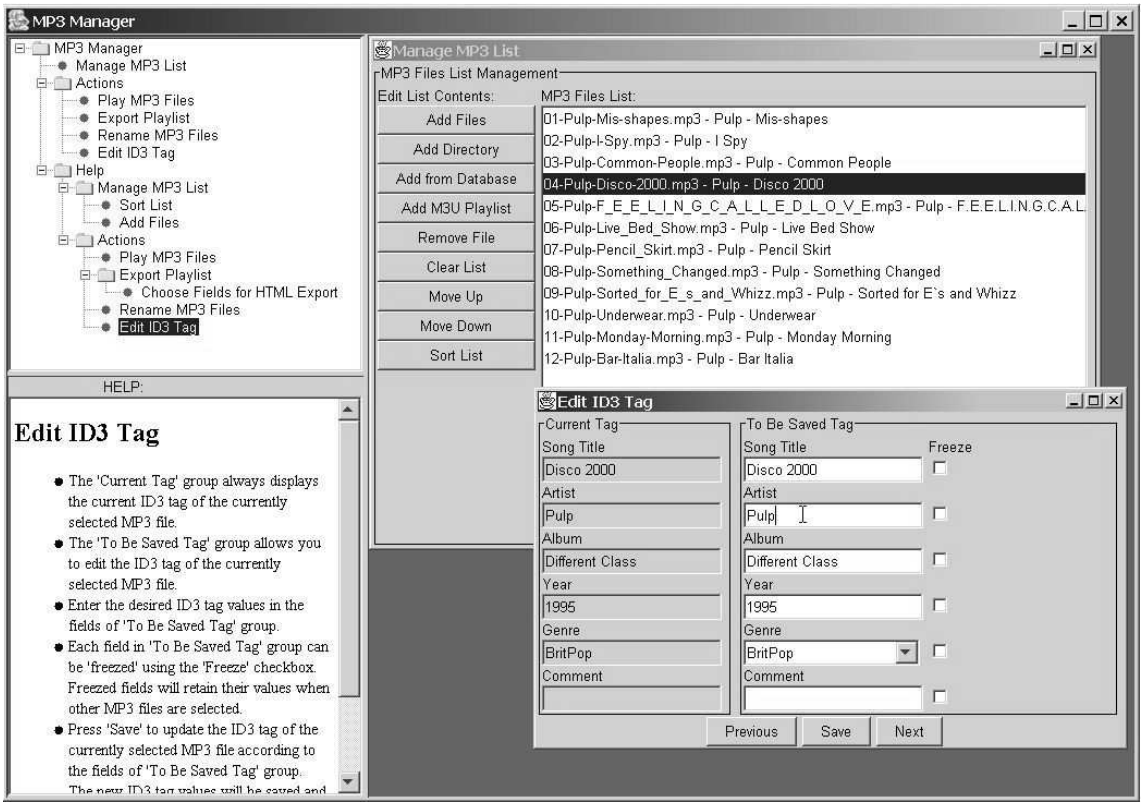

Figure 2. The MP3 Manager Application presented via *TreeDesktop* Displayer

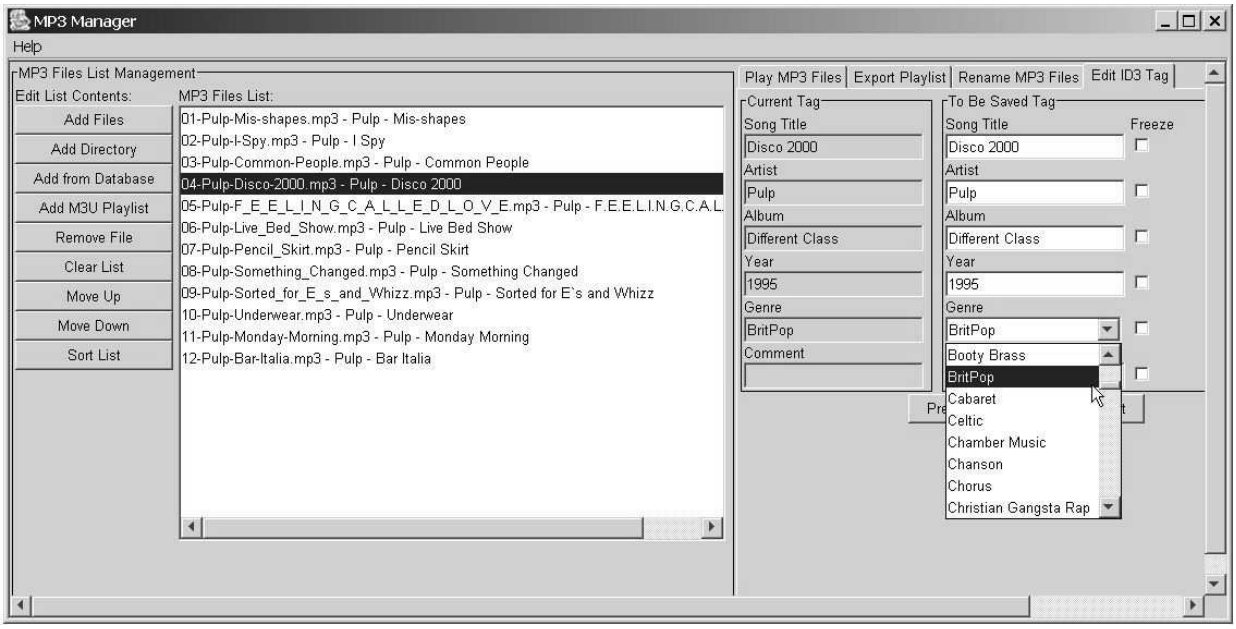

Figure 3. The MP3 Manager Application presented via *OneRowTabbedGroups* Displayer

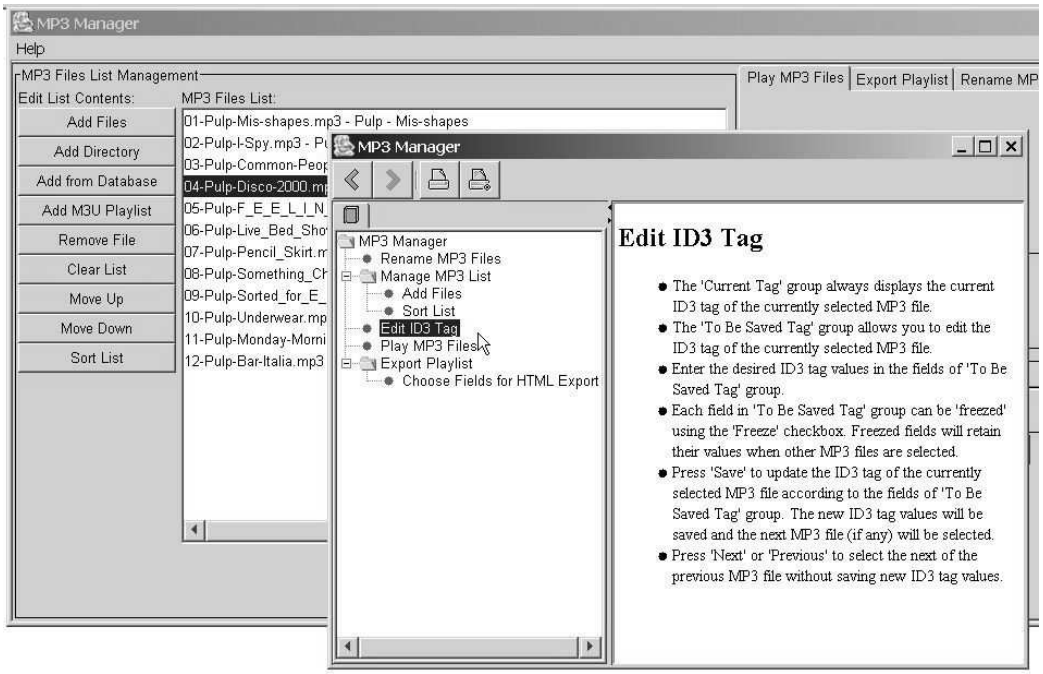

Figure 4. The online help as presented by *OneRowTabbedGroups* Displayer

```
//This is the displayer from Figure 2
UseCaseDisplayer displayer = new UseCaseDisplayerTreeDesktop("MP3 Manager");

//This is the displayer from Figures 3 and 4, it can be used instead
//UseCaseDisplayer displayer = new UseCaseDisplayerOneRowTabbedGroups("MP3 Manager");

displayer.registerUseCase( ManageMP3List.getInstance() );
displayer.startPackage("Actions");
displayer.registerUseCase( PlayMP3Files.getInstance() );
displayer.registerUseCase( ExportPlaylist.getInstance() );
displayer.registerUseCase( RenameMP3Files.getInstance() );
displayer.registerUseCase( EditID3Tag.getInstance() );
displayer.endPackage();

//The name of the directory, where HTML help files reside is passed
displayer.createHelp(".");

displayer.appear();
```

Figure 5. Typical code that starts $\mathcal{SI}+$ applications

```
java.sql.ResultSet query_results =
    db_proxy.executeQuery("SELECT BOOKNAME, AUTHOR, RET_TIME FROM RETURNED
                           WHERE RET_DATE='2003-05-23' AND IS_LATE=TRUE");
table_model.setDataFromResultSet(  query_results );
```

Figure 6. Direct presentation of a `ResultSet` in a table

BOOKNAME	AUTHOR	RET_TIME
Anna Karenina	L. Tolstoy	**10:14:00**
The C Programming Language	B. Kernighan, D. Ritchie	**15:30:00**

Figure 7. The results of a query from Fig. 6