

From Usage Scenarios to User Interface Elements in a Few Steps

Hermann Kaindl

Siemens AG Austria | <mailto:hermann.kaindl@siemens.at>

Rudolf Jezek

Siemens AG Austria | <mailto:rudolf.j.jezek@siemens.at>

Abstract. In practice, designers often select user interface elements like widgets intuitively. So, important design decisions may never become conscious or explicit, and therefore also not traceable. In order to improve this situation, we propose a systematic process for selecting user interface elements (in the form of widgets) in a few explicitly defined steps, starting from usage scenarios. This process provides a seamless way of going from scenarios through (attached) subtask definitions and various task classifications and (de)compositions to widget classes. In this way, it makes an important part of user interface design more systematic and conscious. For an initial evaluation of the usefulness of this approach, we conducted a small experiment that compares the widgets of an industrial GUI that was developed as usual by experienced practitioners, with the outcome of an independent execution of the proposed process. Since the results of this experiment are encouraging, we suggest to investigate this approach further in real-world practice.

Keywords: task analysis, usage scenarios, user interface elements, widgets

Learn more about usage-centered design at <http://www.forUse.com>.

From C. Kolski and J. Vanderdonckt, editors, *Computer-Aided Design of User Interfaces III*, Proc. 4th International Conference on Computer-Aided Design of User Interfaces (CADUI'2002), Valenciennes, France, May 2002. ISBN 1-4020-0643-8. Dordrecht, Netherlands: Kluwer, 2002, pp. 91-102. Reprinted with permission of the authors and the publisher. Copyright © 2002, Kluwer Academic Publishers, <http://www.wkap.nl>.

Chapter 8

FROM USAGE SCENARIOS TO USER INTERFACE ELEMENTS IN A FEW STEPS

Hermann Kaindl and Rudolf Jezek

Siemens AG Österreich, Geusaug. 17, A-1030 Vienna (Austria)

E-mail: hermann.kaindl@siemens.at

Tel.: +43-51707-43288 - Fax: +43-51707-53270

Siemens AG Österreich, Siemensstrasse 92, A-1210 Vienna (Austria)

E-mail: rudolf.j.Jezek@siemens.at

Tel.: +43 51707

Abstract In practice, designers often select user interface elements like widgets intuitively. So, important design decisions may never become conscious or explicit, and therefore also not traceable. In order to improve this situation, we propose a systematic process for selecting user interface elements (in the form of widgets) in a few explicitly defined steps, starting from usage scenarios. This process provides a seamless way of going from scenarios through (attached) sub-task definitions and various task classifications and (de)compositions to widget classes. In this way, it makes an important part of user interface design more systematic and conscious. For an initial evaluation of the usefulness of this approach, we conducted a small experiment that compares the widgets of an industrial GUI that was developed as usual by experienced practitioners, with the outcome of an independent execution of the proposed process. Since the results of this experiment are encouraging, we suggest to investigate this approach further in real-world practice.

Keywords: Task analysis, Usage scenarios, User interface elements, Widgets.

1. INTRODUCTION

In current practice, user interface design is typically more an art rather than science or engineering and requires much experience. In particular, designers tend to select user interface elements in the form of widgets for productivity tools intuitively, based on their experience. So, we address the problem of selecting widgets for a GUI that will be built from those building

blocks. Striving for a systematic process to be followed in a step-by-step fashion, we propose a task-based approach to widget selection.

Our approach is based upon *task analysis* and *scenario-based design*, assuming that envisaged usage scenarios of reasonable quality are already available. We propose to explicitly assign subtask descriptions to the interactions documented in such a scenario, to the steps of both users and the proposed system to be built. Through combining those subtasks that together make up an interaction, *interaction tasks* are identified. For these, the right granularity needs to be found, which may require task composition or decomposition. The resulting interaction tasks can be classified according to the kind of interaction they require, using a class hierarchy of interaction tasks that we developed for this purpose. From this classification, it is possible to map the interaction tasks to a class hierarchy of widgets that we developed as well. Up to this point, our process description is seamless, while the subsequent selection of concrete widgets is beyond the scope of this paper.

We wanted to get an idea of whether and how strongly the kinds of widgets selected with this approach might correspond with those in the development of an industrial GUI. So, we conducted a small experiment with such a GUI. The results are encouraging and make us believe that this approach is promising. This paper is organized in the following manner. First, we summarize related work in the literature. Then, we present our process for widget selection, with its focus on the identification of interaction tasks and their mapping to widget classes, and illustrate it with a running example. Finally, we describe the experiment and present its results.

2. RELATED WORK

In the literature, many approaches to *task analysis* and *task modeling* can be found (see, e.g., [1, 4, 10]). A task can be viewed as a piece of work that a person or other agent has to (or wishes to) perform. The granularity of tasks can vary from whole work tasks [4] to minute detail of how a user interacts with a computer at the command and input/output level [1]. Since we propose an approach to identifying user interface elements in this paper, our focus here is on the latter, more detailed level. Still, work tasks are the original starting point for the design of a user interface in our approach.

Concrete ways of performing work tasks can be described in scenarios such as those in our approach. Similar constructs to scenarios have occasionally been called *methods* or *plans* in the context of task analysis. The author of TKS (Task Knowledge Structures) [4] also dealt with scenarios, which in this context describe human activity [5]. Another role of scenarios in this ap

proach is in depicting an account of a proposed course of action. Our approach fits in here as a proposal for systematically designing a *task model* based on such scenarios and developing an *abstract interface model* from it (these notions are taken from ADEPT [13]).

The approach most similar to ours for scenario-based GUI design is called *essential modeling* and can be found in [2]. While we stick with concrete scenarios, the basic idea behind essential modeling is *abstraction* from a concrete use case to an abstract use case and, to develop an abstract user interface with abstract interface elements from it and finally a concrete GUI with widgets. This approach is certainly useful and can possibly also be combined with ours, but there is no seamless process defined of how one should go from scenario descriptions to the user interface elements (widgets).

Even approaches to automatic generation of user interfaces can be found in the literature (see, e.g., [12]). They typically focus on input and output of data, based on data models (E/R models). In [12], there is a distinction between abstract and concrete interface objects and a focus on automatically mapping the former to the latter using selection rules. Other work has its focus on layout issues, proposing knowledge-based generation and evaluation [7] or creation and metric-based evaluation [11] of user interfaces. Rather than building on what is to be input and output, or elaborating on how the user interface should really look like, we focus on what is to be done with the resulting user interface once it will be available, and what its basic building blocks should be.

Our approach centers around (a taxonomy of) *interaction tasks*, a notion that can be found already in [9]. It is used there for abstracting from the details of window management, which is similar to our abstraction from concrete widgets. But we describe how to systematically determine such interaction tasks and to map them to widget classes.

3. A SYSTEMATIC PROCESS FOR WIDGET SELECTION

Our new process for widget selection is based on scenarios, more precisely envisaged interaction scenarios of the kind dealt with in [6], in the larger context of the method and tool named RETH (Requirements Engineering Through Hypertext). This approach combines scenarios both with functions and goals. Functions are required to make the desired behavior of some scenario happen in order to achieve one or more goals. Rather than giving elaborate theoretical explanations, we simply present the example scenario “Displaying a summary of changes” in Table 1 as an illustration,

which was developed according to this approach. This scenario serves also as a running example for the explanations of the new process below. It is one of the 20 scenarios actually defined in the case study at hand.

In order to facilitate a better understanding of this scenario and the widgets resulting from it, let us briefly sketch the context. This scenario is part of a requirements specification for a versioning facility of the RETH tool, where it helped to acquire and to define the functional requirements. One might think that there would be a complication introduced through having RETH both as the target system and the tool. However, it is a common *bootstrapping* approach to use an existing tool for developing a subsequent release of the same but enhanced tool. In addition, our focus here is not on the use of the RETH tool, but rather on a specific part of the RETH method.

Table 1. A tabular representation of the scenario Displaying a summary of changes.

Scenario <i>Displaying a summary of changes</i>	
1. The user initiates the displaying of a summary of changes between any two versions. <i>Task:</i> To select Display Change Summary	2. RETH displays a list of all available versions and asks the user to select the first version. <i>Task:</i> To display version list
3. The user selects a version from the list. <i>Task:</i> To pick a version	4. RETH displays a list of all available versions and asks the user to select the second version. <i>Task:</i> To display version list
5. The user selects a version from the list. <i>Task:</i> To pick a version	6. RETH displays a summary of changes that have occurred between the two versions <i>Task:</i> To display change summary

Since such a scenario describes interactions between a user and the system to be built, it pre-determines to some extent the kind of interface that will support it. The extent depends on how concrete or abstract these interactions are specified. Abstract use cases [2] leave more room for later GUI design decisions. In fact, scenario descriptions may actually vary within a spectrum from concrete to abstract [6]. In this paper, we assume the availability of more concrete scenarios.

The creation of more concrete scenarios already involves more such design work. In some sense, designing more concrete scenarios also has to assume the availability of certain user-interface elements, in our case widget classes. As long as the usual functionality is covered, this should not constrain the design much. We deal with widgets from the MS Windows style

guide [8] in the context of this paper. The process for widget selection as presented here selects appropriate widget classes from the ones available.

Since this process is seamless from concrete scenarios to widget classes through several steps of task manipulations, it facilitates traceability. Of course, some record linking the various tasks needs to be kept for this purpose. The difference to unguided GUI design in this respect is, however, that defined artifacts in the form of such task descriptions are available for such a record.

3.1 Interaction Scenarios with Attached Task Descriptions

As a unique feature of our scenario descriptions, the steps of the proposed system to be built have annotations that link to descriptions of functional requirements. Whenever this system performs such a step, it will need to accomplish these functions. In this sense, this relation between a step and a function is called *By-Function*. Since the scenarios are envisaged at the requirements stage, the functions are not yet available but they are still functional requirements.

Additionally, the steps of users of that system can have analogous attachments. Rather than talking about functions to be required from users, let us view them as attachments to task descriptions. In fact, we can also view the functional requirements as specifications of tasks that the proposed system to be built will have to perform. In the following, we assume that each and every step of an envisaged interaction scenario has an attachment of descriptions of one or more tasks that are associated with the step. In Table 1, only the names of these tasks are given. They are underlined in order to indicate that they link to the corresponding task descriptions (in the RETH tool itself or Web representations automatically generated from it, in the form of hypertext links). We assume that the actual task descriptions are not needed here for understanding this example.

In general, these are many-to-many relations between the steps of a scenario and such tasks. Therefore, a given set of scenarios is associated with a set of tasks which have to be performed for executing these scenarios. So, this set of tasks is related to the whole set of scenarios, rather than a single one.

3.2 From Task Descriptions to Widget Classes

The core of our new approach leads from the set of those task descriptions to widget classes in the following steps. It is systematic in that these steps can and should be performed in the given sequence.

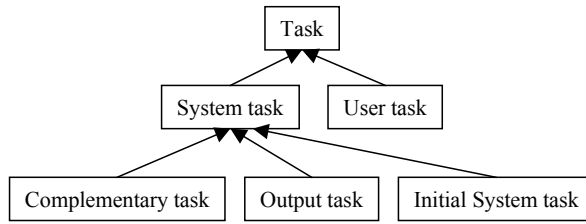


Figure 1. A hierarchy of task categories.

1. **Put tasks into predefined categories.** First, we propose to put each task into one of several categories that we define here with the intention to work out which of the tasks relate to the user interface and, in which way. As indicated above, we distinguish between tasks to be performed by the user (User Tasks) from tasks to be performed by the proposed system to be built (System Tasks). Fig. 1 illustrates this distinction in a UML class diagram, where the triangles denote a relationship between object classes that we can intuitively view as a specialization. That is, a User Task is a special case of a Task in general. In our running example, the following tasks are obviously put into the category User Task: “To select Display Change Summary” and “To pick a version”. The latter occurs actually twice, which is a specialty of this scenario. A System Task is also a special case of a Task in general. In this diagram, we can even see a further specialization of System Tasks.
 - *Complementary Tasks* – These System Tasks are complementary to User Tasks in the sense that they together need the user interface for being performed. For each User Task, something needs to be prepared in the user interface so that it can be performed on it. That is what a Complementary Task of the system to be built does. In our running example, the task “To display version list” occurs twice, and we put it into the category Complementary Task, corresponding to the User Task “To pick a version” given above.
 - *Output Tasks* – These are tasks of the system to be built that output something without a User Task to complement with. That is, an Output Task does not prepare anything in the user interface that would be needed by a complementary task as defined above. In our running example, we understand the task “To display change summary” in such a way that the change summary output is not to be processed further in the user interface by a User Task. So, we put it into the category Output Task.
 - *Internal System Tasks* – These are System Tasks that do not have any effect on the user interface, since they are performed with computations inside the program only. Of course, Internal System Tasks are not directly involved in our process of widget selection, but we in

clude them for the purpose of completeness. In our running example, there is no Internal System Task.

2. **Aggregate complementary tasks.** Then, the complementary tasks of users and the proposed system can be aggregated. For this purpose, we define an aggregate of such tasks and call it Interaction Task, since the complementary tasks together make up an interaction through the user interface. Fig. 2 illustrates this relationship in a UML class diagram, where the diamonds denote the aggregation relationship. A concrete Interaction Task (an object-oriented instance of this class) is the whole and, concrete User Tasks and Complementary Tasks are the parts in such an aggregation. In our scenario descriptions, not for every User Task a Complementary Task is explicitly given and vice versa. Since the point of this step is the identification of Interaction Tasks, this should not be much of a problem, as long as they can be identified from either a User Task or a Complementary Task. Output Tasks are special, since they ave no complementary User Tasks in our approach. Still, we think it is appropriate to view them as a specialization of Interaction Tasks (more precisely a specialization of I/O Tasks, see below). In our running example, we aggregate the tasks “To display version list” and “To pick a version” into the Interaction Task “To interact on selecting a version”. For the User Task “To select Display Change Summary”, no Complementary Task is explicitly given in this scenario. Its (implicit) precondition is that such a selection is possible before the scenario can be executed, i.e., a Complementary Task has been performed before. Still, it is possible to identify a corresponding Interaction Task “To interact on selecting Display Change Summary”.

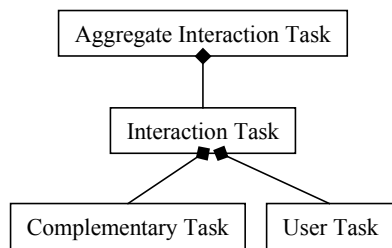


Figure 2. Task aggregations.

3. **Compose and decompose tasks.** This step is optional, since those Interaction Tasks may or may not yet have the right granularity. Either they might need to be composed with others to an Aggregate Interaction Task or they might need to be decomposed into “atomic” tasks, where the decomposed task is modeled as an Aggregate Interaction Task. This is again an aggregation relationship, which is illustrated in the top part of Fig. 2. In our running example, the Interaction Task “To interact on se

lecting a version” occurs twice, once for the old and once for the new version to be compared. But according to the scenario description these selections are to be executed in sequence. So, no Aggregate Interaction Task is needed here and also none for the other Interaction Tasks in this scenario. Since all of them are “atomic”, also no decomposition is needed here. In general, however, even recursive composition may occur, according to the quasi-recursion in UML between Interaction Task and Aggregate Interaction Task in Fig. 3. That is, an Aggregate Interaction Task aggregates Interaction Tasks. It is an Interaction Task itself and can, therefore, be aggregated by another Aggregate Interaction Task. This case actually occurred in the scenario “Displaying the List of Document Versions”, where first the Interaction Tasks on selecting all versions and on major versions only are aggregated. The resulting Aggregate Interaction Task is itself aggregated with the Interaction Task on viewing a version list. It is clear that such task composition and decomposition is due to human judgement. Still, it occurs embedded in the systematic process and is done consciously.

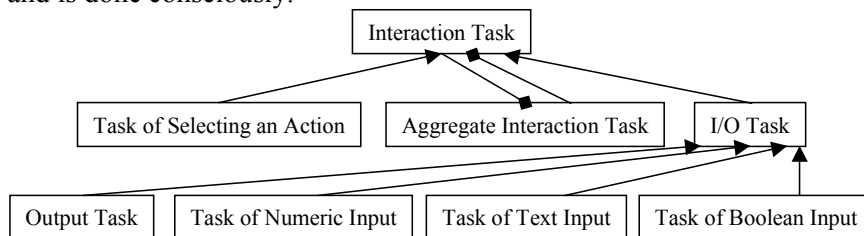


Figure 3. A classification hierarchy of Interaction Tasks.

4. **Classify tasks according to the kind of interaction required.** After that, we propose to classify the resulting Interaction Tasks according to the kind of interaction they require. Fig. 3 shows the classification hierarchy that we have defined for this step. (In fact, this is another possibility of specializing tasks than that illustrated above in Fig. 1.) Apart from the Aggregate Interaction Tasks, we distinguish here Tasks of Selecting an Action from I/O Tasks, which are further specialized according to what is input and output, e.g., text or Boolean values. In our running example, we obviously classify “To interact on selecting Display Change Summary” as a Task of Selecting an Action. Since “To interact on selecting a version” means I/O on the textual representation of the version name, we classify it as a Task of Text Input.
5. **Map to widget classes.** Finally, it is possible to map from the domain of classes of Interaction Tasks (see Fig. 3) to the domain of Widget classes (see Fig. 4) in such a way that a widget from the resulting class can support the corresponding Interaction Task in a GUI. More precisely, we

map I/O Task to Control, Task of Selecting an Action to Widget for Selecting an Action, and Aggregate Interaction Task to Container Widget. In addition, we map the specializations of I/O Task to their obvious counterpart in the specializations of Control in the widget classification hierarchy. Due to space restrictions, we show here only the top-most part of the hierarchy of widget classes, but the full and fine-grained hierarchy can be found in [3]. This object-oriented hierarchy reflects a breakdown according to the purpose of the widgets in the GUI, rather than a structural view of widgets. Structural composition, however, is modeled through aggregations of widgets. While we cannot show our full hierarchy of Interaction Tasks in this paper either, note that it is equally detailed as the hierarchy of widget classes. In fact, it is complete with regard to the functionality of the widgets of the MSWindows style guide [8]. In our running example, for the task “To interact on selecting Display Change Summary” a Widget for Selecting an Action should be appropriate and, for the task “To interact on selecting a version” a Widget for Text Input. We view text input not only in the sense of typing text here, but more generally, so that it includes selection of text from a list. For the Output Task “To display change summary”, an Output Widget should be appropriate. So, this mapping looks rather straight-forward. In fact, this is partly due to the way we defined both hierarchies, so that there are more or less obvious one-to-one correspondences. However, it seems clear that for any reasonable and complete hierarchy of user interface elements, there will exist some mapping from our classification hierarchy of Interaction Tasks to it.

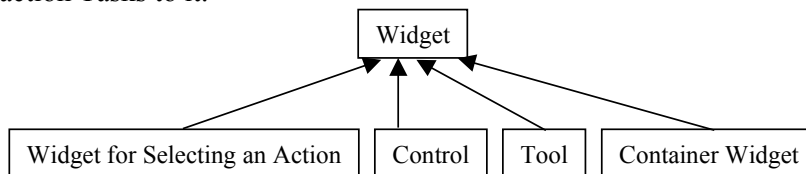


Figure 4. Our classification hierarchy of widgets (its top-most level).

4. AN EXPERIMENT WITH THIS APPROACH

Ideally, the usability of a user interface resulting from our process should be evaluated. However, it systematically covers the selection of widget classes only, while the selection of concrete widgets and their placement are out of its scope and require much human skill and experience. So, evaluating concrete widgets or even a complete user interface built by using our process would involve too much bias, because even more human judgement would be involved. Since a usability test requires a real user interface or at least a

reasonable mockup, a selection of widget classes cannot be evaluated in this way. Even usability inspections cannot be applied to a set of widget classes.

Still, in order to get an idea about the validity of this new approach, we conducted a small experiment. The second author of this paper worked completely through the process described above for all 20 scenarios in the given requirements specification of the versioning facility of RETH (which was the only information available to him about this facility). This approach had the advantage that he knew our process already, of course, while we would have had to teach someone else first. Note, that he is less experienced in user-interface design than the key interface developer in the regular development.

For the various (re)classifications, he used the RETH tool itself. Primarily, he utilized the object-oriented features of the tool, which helped to avoid inconsistency. Of course, this was the RETH tool without the versioning facility, which the second author has never seen until this exercise was completely finished. In fact, the versioning facility was developed concurrently and independently. The real GUI of this facility was designed and implemented by experienced GUI developers. After both this exercise and the development of the real GUI was finished completely, we compared the results of executing our proposed process with the part of the RETH user interface for versioning of the tool itself.

Before this experiment was conducted, we formulated the following hypothesis: The kinds of widgets selected in both processes would correspond to a high degree.

A major issue was to define how to measure the proportion of the widget classes selected through execution of the new proposed process, from the widgets in the GUI developed as usual (which served as a kind of “oracle”). We defined the following way of measuring this degree of correspondence: How frequent is a widget of the real versioning GUI in the widget class that is the direct result of the mapping from the corresponding task in step 5 of our process as described above?

Table 2. Experimental results.

	No. of widgets	No. of corresp. Widget classes	Frequency of correspondence
Widget class mapped	75	64	85%
Widget class at level 1	75	65	87%

Table 2 shows the results. 85.3 percent of the widgets of the real versioning GUI were in the widget class that is the direct result of the mapping. Since the mapping is partly done on two levels of the hierarchies (1 and 2), we were also interested in how the results would differ between them. With 86.7 percent for the level 1 only, this result is more or less the same as that

of the mapping overall. This means that the widgets of the real versioning GUI were also in the more specific classes most of the time when they were in the less specific class at level 1. This suggests that most of the different selections actually occurred close to the root of the widget class hierarchy. An analysis of these differences confirmed this and revealed that they came from different strategic design decisions made. The second author of this paper had rather straightforward enhancements, e.g., of the already existing menus in mind, while the designers of the real GUI decided to introduce an extra subpane for versioning, that provides a permanent display of versioning information whenever a versioned document is loaded. This implies that in the real GUI there is a permanent selection of versioning-related information, where an action can be immediately performed on. In essence, this amounts to a trade-off between less “real-estate” used on the screen and fewer interactions required. Based on this result, we tend to accept the hypothesis above. While the outcome is clearly not the same, the results are encouraging. The high frequency of the same widget classes on a higher level of classification after the mapping from the systematically determined Interaction Tasks suggests preliminary evidence that the proposed process delivers results that correspond to a high degree with what designers elaborate intuitively and based on their experience.

5. CONCLUSION

In this paper, we propose a new and systematic process for selecting user interface elements in the form of widgets. From given interaction scenarios, it leads in a few steps to classes of widgets, which specify the kinds of building blocks for a GUI that shall support the execution of these scenarios.

This process involves various manipulations of tasks, based upon task categories that we have defined for it. As a result of these task manipulations, *interaction tasks* are identified and categorized, which seems to be a new approach. These interaction tasks are finally mapped to widget classes. Preliminary evidence suggests that these are more or less the right kinds of building blocks.

For this mapping, we defined a hierarchy of widget classes according to a functional rather than the usual structural view of classifying widgets. This hierarchy covers the MSWindows style guide [8]. We also defined a hierarchy of interaction tasks that reflects this widget hierarchy. We conjecture, however, that it is more generally applicable in the sense that it could also be mapped to other classes of user-interface-elements (e.g., built in hardware). These classification hierarchies are new contributions of this paper, in addition to the process that builds upon them.

There should also be a certain completeness with respect to the given scenarios in the sense that no functionality in the GUI is missing for their execution based on the determined building blocks. These user interface elements should even be consistent with respect to the given scenarios, in the sense that the required functionality is provided by them. So, all of the given scenarios should be executable at least somehow with a user interface that contains widgets of the kind the derived interaction tasks are mapped to.

We hope that this systematic process contributes to a better understanding of the relationship between usage scenarios and user interface elements through the various tasks involved. More generally, it should help to better understand a certain part of user interface design.

REFERENCES

- [1] Card, S.K., Moran, T.P., and Newell, A., *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum, Hillsdale, 1983.
- [2] Constantine, L. and Lockwood, L.A.D., *Software for Use*. ACM Press, New York, 1999.
- [3] Jezek, R., *Methodischer Designprozeß von GUIs unter Verwendung von Szenarios*. Diplomarbeit, Universität Wien, Vienna, 2000.
- [4] Johnson, P., *Human-Computer Interaction: Psychology, Task Analysis and Software Engineering*. McGraw-Hill, London, 1992.
- [5] Johnson, P., Johnson, H., and Wilson, S., *Rapid prototyping of user interfaces driven by task models*, in Carroll, J.M. (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development*, John Wiley & Sons, New York, 1995, pp. 209-246.
- [6] Kaindl, H., *A design process based on a model combining scenarios with goals and functions*, IEEE Transactions on Systems, Man, and Cybernetics (SMC) Part A, Vol. 30, No. 5, 2000, pp. 537-551.
- [7] Kim, W.C. and Foley, J.D., *Providing High-Level Control and Expert Assistance in the User Interface Presentation Design*, in Proceedings of ACM Conference on Human Factors in computing systems InterCHI'93 (Amsterdam, 24-29 April 1993), ACM Press, New York, 1993, pp. 430-437.
- [8] Microsoft, *The Windows Interface Guidelines for Software Design: An Application Design Guide*, Microsoft Press, Redmond, 1995.
- [9] Myers, B.A., *A new Model for Handling Input*, ACM Transactions of Information Systems, Vol. 8, No. 3, 1990, pp. 289-320.
- [10] Paternò, F., *Model-Based Design and Evaluation of Interactive Applications*, Springer Verlag, Berlin, 1999.
- [11] Sears, A., *AIDE: A Step Toward Metric-Based Interface Development Tools*, in Proceedings of the 8th ACM Symposium on User Interface Software and Technology UIST'95 (Pittsburgh, 15-17 November 1995), ACM Press, New York, 1995, pp. 101-110.
- [12] Vanderdonckt, J.M. and Bodart, F., *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*, in Proceedings of ACM Conference on Human Factors in computing systems InterCHI'93 (Amsterdam, 24-29 April 1993), ACM Press, New York, 1993, pp. 424-429.
- [13] Wilson, S., Johnson, P., Kelly, C., Cunningham, J., and Markopoulos, P., *Beyond Hacking: A Model Based Approach to User Interface Design*, in Proc. of the BCS Conference on Human Computer Interaction HCI'93 People and Computers VIII (Loughborough, 7-10 September 1993), Cambridge University Press, Cambridge, 1993, pp. 217-231.