Modality Abstraction: Capturing Logical Interaction Design as Abstraction from „User Interfaces for All"

Hans-W. Gellersen
Telecooperation Office, University Of Karlsruhe,
Vincenz-Prießnitz-str. 1, D-76131 Karlsruhe, Germany.
Ph. +49 (721) 6902-49, hwg@teco.uni-karlsruhe.de

Abstract. Modality abstraction is a concept for capturing those parts of a user interface that describe logical interaction in abstraction from appearance. Thus, modality abstraction provides a common ground for user interfaces that may differ in used representational media, input modalities and dialogue styles. Building modality abstraction into interactive software contributes to portability, modifiability and integration of different modalities. To facilitate modality abstraction, we have developed the MAUI toolkit with building blocks for *Modality Abstracting User Interfaces* and the MEMFIS method for building modality abstraction into interactive software.

1 Introduction

User interface design comprises two sets of design decisions, one set determining *what* users can do through the interface and the other one determining *how* they can do it. The first set is concerned with the logical interaction between human and computer. Decisions in this set are concerned with the types of information that can be exchanged and the operations a user can invoke. The second set of design decisions is concerned with user interface appearance. The appearance is determined by the interaction modalities chosen for information exchange and invocation of operations. An interaction modality defines a distinguishable way of achieving a logical interaction. Modalities can differ in use of interface devices, human communication channels, representational media and interaction styles.

Commonly, decisions concerning logical interaction and interaction modality are not separated. Before introduction of graphical user interfaces (GUI), the design space for interaction modalities was limited to text I/O due to technological constraints. With lack of alternative modalities there was no serious need for separation of logical interaction from interaction modalities. With introduction of GUIs the modality design space grew, but in general it was felt that graphics as medium and direct manipulation as interaction style were defining a superior modality. Built on this assumption, most guidelines, methods and tools support a very restricted range of alternatives for a given logical interaction. Most of these methods and tools are appearance-oriented and do not capture the logical interaction design to be reused for alternative designs.

Recently, new interaction technologies are becoming available and affordable, thus enabling new interaction modalities. Also, it becomes apparent that diverse modalities are required to satisfy demand and expectation of different user communities (home users, elderly, disabled) in different usage environments (home, car, meeting,…). So, user interface design is challenged to support alternative modalities for an application under de-

velopment. A prerequisite is to capture the commonalities among such alternatives: the logical interaction design. In this paper, we present a toolkit and a method for capturing logical interaction design based on the concept of modality abstraction.

2 Related Work

We are not aware of any work explicitly aimed at support for building modality abstraction into interactive software. Yet, there is related work with respect to some goals underlying modality abstraction. First, there is work on capturing commonalities among various user interface platforms to reduce development effort. Secondly, there is work on shifting the user interface design focus from appearance to logical interaction; the approaches can be divided into toolkits and methodologies. Thirdly, there is work on integration of modalities for interaction flexibility.

Capturing commonalities among diverse platforms. There are many toolkits that capture commonalities among diverse platforms in platform-independent user interface objects. User interfaces constructed based on these toolkits can easily be deployed to range of platforms. These platform-independent toolkits are often referred to as *virtual toolkits* or *PIGUI toolkits* (platform-independent GUI). The term virtual does not seem appropriate, as these toolkits are simply generalisations from a range of fairly similar GUI toolkits, so we clearly prefer the term PIGUI [8]. PIGUI objects are platform-independent but fully committed to the GUI paradigm. They have attributes and operations explicitly related to the desktop metaphor and to direct manipulation. So, they can only be reused for a currently very popular but nonetheless fairly limited range of platforms.

Shifting design focus from appearance to logical interaction. In development of GUI the design focus is usually on appearance. Commonly, user interfaces are built before the functional core of an application, which naturally leads to focus on the looks rather than on logical interaction between functional core and user. Work addressing this issue can be divided into toolkit and methodological approaches. Toolkit approaches aim at provision of semantic-oriented rather than appearance-oriented user interface objects. The Garnet system provides a set of *interactors* as abstractions from low-level controls defined by behaviour rather than appearance [10]. Still, the interactors are clearly GUI-oriented, for example the *move-grow-interactor.* The ACE system provides *selectors* as abstract interaction objects supporting user choices of either commands or data values. Selectors are uncomprisingly defined by selection semantics in abstraction from modalities [9]. Still, they are not aimed at building modality abstraction into a system. Rather they are one of three interrelated collections of building blocks for construction of semantic-based concrete user interface objects. The other two collections comprise basic data types and concrete presentation objects. In contrast to toolkit approaches, methodological approaches aim at deriving required user interface objects from task models. One such approach is the Adept design process [13]. Adept starts with modelling the tasks to be performed by the computer system in abstraction from user tasks. These tasks are then mapped to abstract interaction objects. These are simply qualified by the type of input they accept from the user, so there are for example objects for text entry or numeric input. So, the abstract interface model is limited for capturing logical interaction as it only captures temporal relations among data entry actions. Another very interesting task-based approach is IMAP for information mapping to interaction modalities [3]. IMAP starts with task analysis for identification of the information to be exchanged between human and computer. A system of rules aids the designer in mapping information to suitable modalities for realising the information exchange.

Integration of modalities for interaction flexibility. The common approach for supporting alternative modalities is to extend a given technology to support an additional technology. For example, speech interaction facilities are currently being added to computer systems by extending GUI technology. This leads to inappropriate notions such as *display* for referencing a speaker as output device in audio extensions for X. Speech-based interaction objects are unintuitively cast in terms of widgets, for example a speech-based command would be modelled as *button*. Savidis et.al. presented a more suitable approach to integration of modalities in the context of developing integrated solutions for blind and sighted users [12]. They acknowledge the need for abstraction from modalities and argue that specific modalities can only be appropriately supported when they are defined with respect to an abstraction rather than with respect to each other. In their HOMER system they support construction of *virtual objects*, though it is not clear how proper abstraction is ensured in this process.

3 MAUI: AToolkit for Modality Abstraction

3.1 Modality-Abstracting User Interface Objects

The MAUI (*Modality-Abstracting User Interface*) toolkit provides abstract interaction objects as building blocks for modality abstraction. Each MAUI object is a mechanism for information exchange between human and computer, defined by the kind of information exchange it enables. MAUI objects are organized in an inheritance tree. The root object `Interaction` is the generalmost interaction object capturing attributes and operations shared by all interaction objects. For instance, it has an attribute `Perceptibility` (an abstraction from *visibility*) and operations `make_perceptible` and `make_imperceptible`. Each specialization in the hierarchy is defined by a discriminator cleanly separating the different specializations of an interaction object. Hence, the structure can also serve as decision tree for selection of objects in the design process (cf. below). The root object is specialized into `Input` and `Output` with the direction of information exchange as discriminator. `Output` is in fact identical with `Interaction` but has been added for reasons of symmetry. `Input` is the abstraction of interaction objects that are responsive to user events. It inherits attributes and operations from `Interaction` and adds attributes and operations common to all objects that are responsive to user events, for instance an attribute `Sensitivity` determining whether or not an object is currently responsive.

Both `Input` and `Output` are further specialized based on discrimination of control flow and data flow. `View` is the abstraction of all output objects that realize data flow from computer to human by making application concepts perceptible at the user interface. `Message` is the abstraction of those output objects that can make control information perceptible. These two output concepts have important differences, for instance, `View` as opposed to `Message` has to be associated with a conceptual object of the application domain; further it requires an update operation and update behaviour, which `Message` does not need as it delivers static information, such as error messages. Input objects are grouped into `Control` and `Entry`. Control objects have a `callback` operation in common for invocation of an application function in response to a specified user event. The simplest specialization is `Signal`, which can sense only one predefined user event and reacts with unparametrized invocation of an application function. `Command` enables editing of control information and a parametrized callback. Data entry objects have in common that they inherit the behaviour of view objects. From `View` they inherit an association with a conceptual object. It is this object whose value can be manipulated by the data entry object. Inheritance from `View` allows data entry objects to be used for

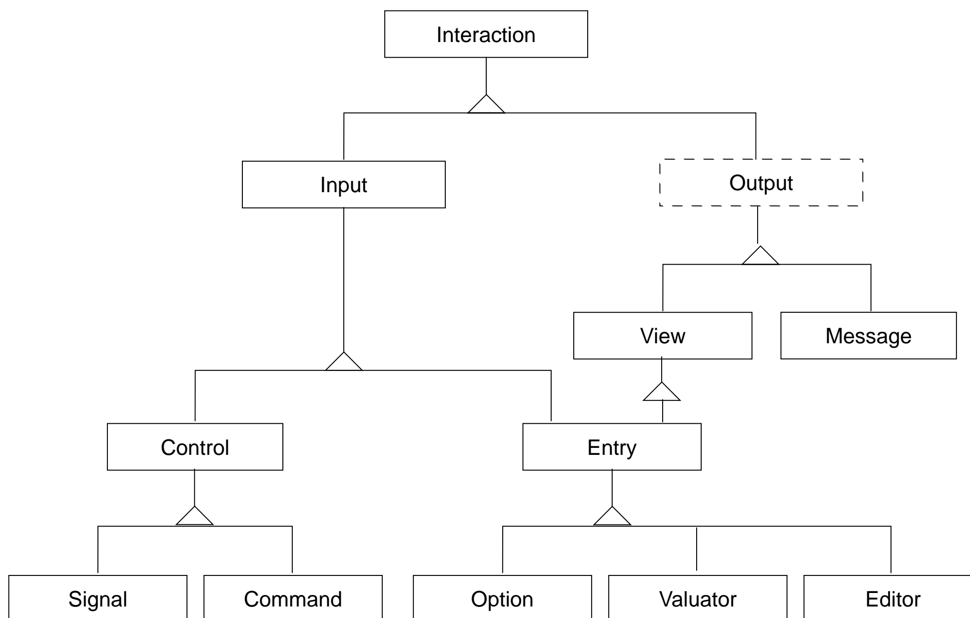

Fig. 1. Inheritance tree of MAUI objects

two purposes at once: display and manipulation of application state. Display and manipulation are commonly combined in state-of-the-art user interfaces. Specializations of Entry are `Option` which enables value selection from a number of objects, `Valuator` which supports value selection from a continuous range of values, and `Editor` for editing of input data.

Figure 1 shows the inheritance tree of MAUI objects in OMT notation. We do not consider the hierarchy to be complete, in particular we investigate further specialization of output objects. Work of the AI community on automated presentation design has produced characterizations for information to be presented, for example dimensionality and urgency [1]. We take these characterizations as well as characterizations of communication intent as starting point for identification of `View` and `Message` discriminators.

3.2 From Abstract to Concrete User Interface Objects

Concrete user interface objects can be derived from MAUI objects by adding modality-dependent attributes and operations. An idealistic vision would be to further refine the MAUI object hierarchy top-down across a border of modality-dependence to embody objects of diverse modalities in a single inheritance tree. Alas, a prerequisite would be isolation and ordering of discriminatory modality decisions. Recent work in modality theory identified very useful discriminators such as linguistic vs. non-linguistic, dynamic vs. static, analogueness, specificity and arbitrarity for characterizing modalities [2]. Yet these features are largely independent of each other, have no suitable ordering and thus do not lend themselves for hierarchical organization.

Still, any toolkit of concrete user interface objects, for example GUI widgets, can be realized by specialization of MAUI objects. New toolkits to be developed can inherit logical interaction capabilities from MAUI objects and add modality-specific attributes and operations. Existing toolkits can be adapted to MAUI object interfaces by using the *adapter* design pattern (cf. Gamma et.al.'s catalogue of object-oriented design patterns [6]).

4 MEMFIS: AMethod for Building Modality Abstraction into interactive Software

4.1 Overview

A toolkit, be it well designed, does not ensure good design practice. Hence, we have developed MEMFIS, a method that systematizes the task of building modality abstraction into interactive software. MEMFIS (*Method for Engineering Multimodal flexible Interactive Software*) is based on a object-oriented and a task-oriented model of interactive applications. MEMFIS defines of a set of activities for development of a logical interaction design leading to the construction of an abstract user interface based on MAUI objects. An important aspect of the method, discussed in [7], is integration of dedicated user interface design support in object-oriented software engineering practice.

The MEMFIS method is based on two complementary models for interactive applications: SAM (*Static Architecture Model*) and TOM (*Task-Oriented Model*). SAM models the static architecture of an application in terms of objects that are interrelated in associations, aggregations and generalizations/specializations. All existing object-oriented methods (OOM) for software development contain suitable models for static architecture. For our purpose we chose Rumbaugh's OMT (*Object Modeling Technique [11]*) notation as it is in wide-spread use and easy to understand in its resemblance to entity-relationship-models. To complement static architecture modelling we developed TOM, a new model for dynamic and functional aspects of an application. These aspects are rather poorly supported in existing object-oriented methods [4]. TOM embodies powerful modelling concepts which will be explained below in the course of describing the development activities of our method.

4.2 MEMFIS Activities

MEMFIS consists of a number of activities for development of interactive software with built-in modality abstraction. All these activities contribute to the development of the SAM and TOM models by extending rather than transforming them. All activities are coordinated via these two models which renders the development process seamless. So the various activities do not have to be performed in any specific sequence and can also be performed iteratively.

Essential Modelling. A fundamental analysis activity is the identification of concepts and tasks in an application. Modern analysis methods use scenarios or use cases to this end. Scenarios commonly contain modality commitments, for example concepts such as *screen* and *button*, or event traces committed to particular dialogue styles. Of course, in order to achieve modality abstraction, such commitments have to be avoided. Therefore we adopt Constantine's *Essential Use Case Modelling* [5] for identification of the essential tasks and concepts of an interactive application. Part of the philosophy behind this variation of scenario-based modelling is to assume a perfect world with unlimited technological possibilities so to keep use case descriptions technologically unconstrained.

Task Decomposition. This activity develops a task hierarchy as core of the TOM model. The fundamental purpose of an application as identified by essential modelling is the root of the hierarchy. Tasks to be performed or goals to be achieved in order to fulfil the fundamental purpose are recursively modelled as subtasks. Figure 2 shows the task decomposition of a simple application that we will use subsequently for illustration of our method. The TOM notation uses ellipses to represent tasks, and nested boxes to describe hierarchical task structure. Nested boxes are used instead of typical tree representations

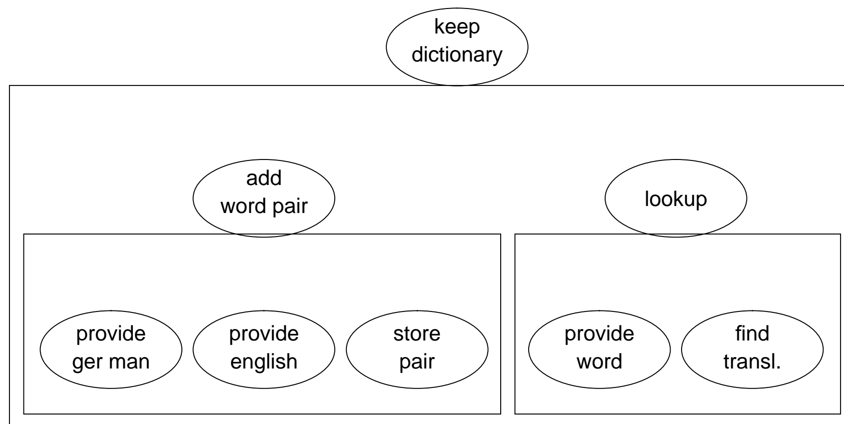

Fig. 2. Basic TOM model of the dictionary application

to reserve the use of arcs for representation of data- and controlflow within the same diagram. The application is a simple german-english dictionary that supports adding of word pairs and lookup of translations. The task of adding a word pair requires a german word to be provided, an english word to be provided and the pair to be stored in the dictionary. Subtasks of the lookup task are provision of a word as key for the lookup and the actual lookup of the translation. In the task description, we deliberately use the notion of "provide data" instead of "enter data". The latter notion would carry the assumption that the required data has to be provided by the user. Even if this is obviously the case, we prefer to defer such a decision to a dedicated task allocation activity.

Static Architecture Modelling. Concepts identified in essential use cases are candidates for objects in SAM. The standard OOM literature describes a repertoire of static architecture modelling activities. Here, we will only describe the SAM model of the dictionary application, so we can reference it in description of other development activities. The central object is `WORD_LIST`, which is an *aggregation* (diamond shape) of *many* (ball shape) objects `WORD_PAIR` which in turn is an aggregation of two objects `WORD`. `WORD_LIST` is a generic object with attributes and operations for management of a list of word pairs, such as list operations and lookup operations. `DICTIONARY` is a *specialization* (triangle shape) of `WORD_LIST` and adds application specific details, for example that the two elements of a word pair are referenced as german or english word, respectively. In contrast to common static architecture modelling approaches, we explicitly exclude user interface objects or objects representing the user from this modelling activity. The rationale for this is that the user interface or in general the human-computer relationship is developed in dedicated modelling activities described below.

Specification of Temporal Relations. This activity refines the TOM model by specification of temporal relations among tasks. For each composite task a regular expression specifies the possible sequences of subtasks. In the TOM notation, the expression is included at the top of each box associated with a composite task (cf. fig. 4). The regular expressions contain operators for sequential (*comma*), interleaved (*and*), and alternative (*or*) task execution. For example, the regular expression assigned to the task `add_word_pair` of our sample application specifies that the two subtasks for provision of a german and an english word can be executed concurrently but before the subtask

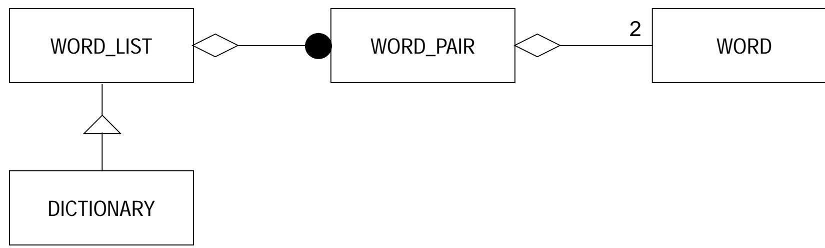

Fig. 3. SAM model of the dictionary application

of storing the new word pair. It further specifies that this sequence has to be executed at least once.

Dataflow Modelling. This activity identifies dataflow between tasks. In a first step, the required input data is established for a task. Then a task has to be identified as provider of this input. This may involve adding of a new task. The relation between provider and consumer tasks is modelled by a dataflow arc (cf. fig. 4, solid lined arcs). Dataflow arcs can be associated with objects modelled in SAM to specify the type of data flowing from provider to consumer task (cf. fig. 5). When the input requirements of all tasks are satisfied, the tasks have to be checked for „dangling output". For each output a consumer task has to be found. Again, this may involve adding of a new task. In our example development, the task `use_translation` was added as destination of the output generated by `find_translation`. The rationale for insisting on a closed system of tasks with a provider task for each input and a consumer task for each output is to avoid external data sources and sinks in the model. Traditional dataflow models have the concept of actors for such external sources and sinks. We avoid actors for two reasons. First, dataflow models of interactive applications tend to use a single actor to model the user. This leaves a very important part of an interactive systems, the user, as completely unstructured source and sink of data. Human tasks in an interactive system remain hidden. Secondly, the introduction of actors commits a task allocation in the process of dataflow modelling. Da-

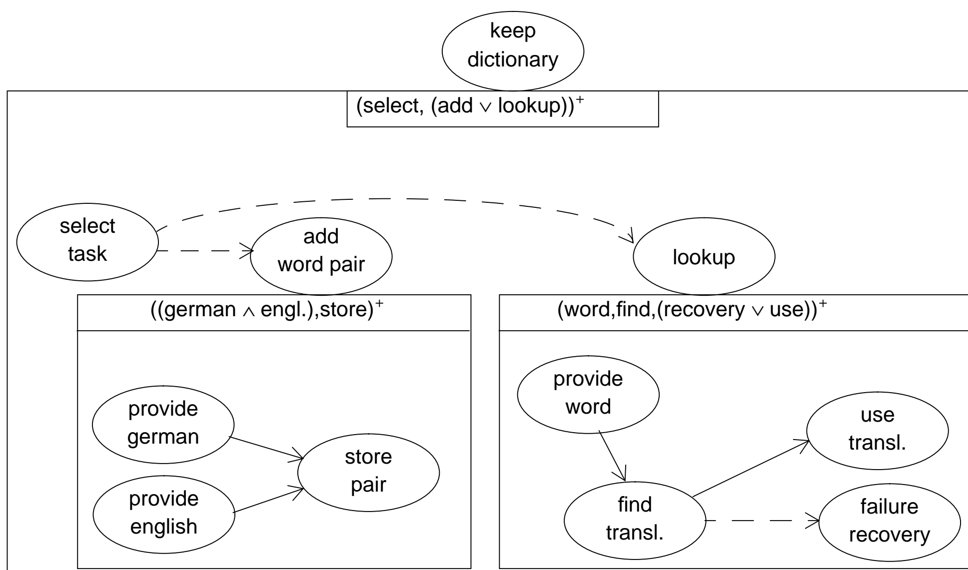

Fig. 4. TOM model with temporal relations, control- and dataflow

taflow modelling and task allocation are orthogonal concepts and ought to be treated separately.

Controlflow Modelling. Specification of temporal relations and dataflow dependencies determine part of the controlflow among tasks. The controlflow modelling activity aims at completing the picture. For all sets of alternative tasks, a task has to be identified that holds the decision about which alternative to take. Possibly, a new task has to be added to the model. For instance, in our sample application `select_task` has to be added for control of the alternative tasks of adding a word pair and performing a lookup. In addition to the handling of alternatives the handling of exceptions such as error conditions has to be modelled. Each task has to be checked for exceptional conditions that might occur. If the exception can not be handled by the task itself, another task has to be identified for handling it. In our sample application, we identify the error condition "word not found" in the task `find_translation`. A new task `failure_recovery` is added to the model to handle this error. The alternative between `failure_recovery` and `use_translation` can obviously be controlled by `find_translation`, so no additional task is required. In the TOM model, the actual controlflow is annotated with dotted lined arcs (cf. fig. 4).

Task Allocation. In MEMFIS, task allocation is performed at two levels. At a more general level, tasks are allocated to either human or computer. At a more detailed level, tasks are allocated to SAM objects. A technique for the more general allocation is to classify tasks according to the kind of processing they require. Tasks that would typically be allocated to the computer are those that require data transformation or algorithmic processing. Tasks that typically would be allocated to the human are those that involve data creation, negotiation or judgement. A technique for the more detailed level of task allocation is to establish a relationship *is-responsible-for* between SAM objects and TOM tasks. Fig. 5 shows our sample application after task allocation. The object `WORD_LIST` was identified to be responsible for the tasks `store_pair` and `find_translation`, as these are rather generic and not dependent on our particular dictionary. The tasks `add_pair` and `lookup` are more application specific as they hold decisions that do not apply to word lists in general, for example the decision that subtask sequences can be executed once or repeatedly. Thus, these tasks are allocated to the application specific object `DICTIONARY`. Tasks for which no responsible SAM object can be identified are candidates for human tasks, as the SAM model explicitly excludes user or user interface objects. So, the two levels of task allocation are integrated smoothly and do not have to be performed in any specific sequence. The TOM notation reflects task allocation in two ways. First, identifiers of SAM objects are included in the ellipses representing tasks for which the objects are responsible. Secondly, a separation line is included to separate human tasks from computer tasks. The nested box notation allows the task structure to be reorganized so human tasks are placed below the line and computer tasks above the line. With this notational convention, the separation line which in fact represents the human-computer interface, becomes a focal point of the TOM diagram (cf. fig. 5). In a CASE tool currently being developed for support of our notation, task allocation is actually performed by dragging and dropping of task ellipses over a given separation line rather than by drawing a complex separation line.

Construction of Modality Abstraction. After task allocation, the dataflow and controlflow arcs crossing the task separation line establish the required interaction between human and computer. For each arc crossing the separation line, an interaction object enabling the required information exchange has to be identified. The MAUI object inheritance tree eases this identification process, as the discriminators for specializations are based

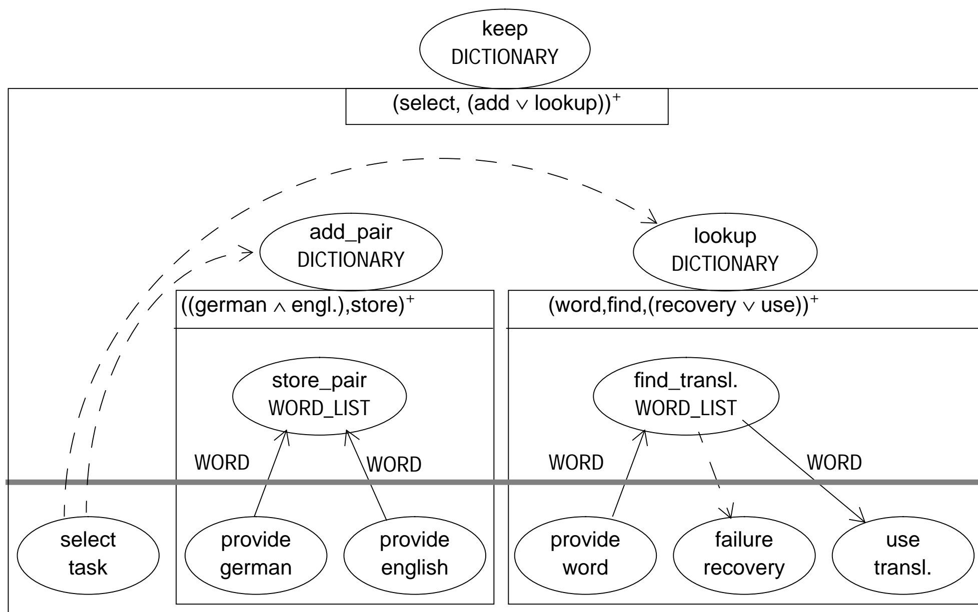

Fig. 5. TOM model after task allocation

on logical interaction capability. The direction of an arc and the distinction of data- and controlflow directly lead to identification of either `Control`, `Entry`, `View` or `Message` as required interaction object. Additional information from the model such as annotation of dataflow arcs with SAM objects can be used for further specialization of these objects. Fig. 6 shows the MAUI objects identified for our sample application. The controlflow arcs from `select_task` to `add_pair` and `lookup` are mapped to `Signal` objects, as the controlflow is unparametrized. All dataflow arcs from the user to the computer are mapped to `Editor` objects as the range of possible values for the data in question can neither be enumerated (then `Option` would have been appropriate) nor be mapped to a representation required for a `Valuator` object. Further dataflow and controlflow from `find_translation` to the user are mapped to a `View` object and a `Message` object. The identified objects are leaves of a hierarchy constituting the modality abstracting user interface. The hierarchy is established by grouping objects recursively.

A default for grouping of interaction objects is to mirror the hierarchical task structure. That means, for each higher-level task a `Group` object is modelled that contains the interaction objects mediating between its subtasks. `Group` objects control their component interaction objects much like parent widgets but of course with respect to logical interaction behaviour. All interaction objects are modelled in SAM notation. Note that the identified interaction objects are instances while the objects modelled in fig. 2 are classes. In contrast to object classes, object instances are represented by rounded boxes. Of course, the identification of user interface objects has to be reflected in the SAM model, for example subclasses of `View` and `Editor` have to be added and associated with the `Word` object.

Mapping to Concrete User Interface. The mapping from modality abstracting user interface to a concrete user interface is performed on a per object basis. For each MAUI object a concrete counterpart has to be identified. This concrete object has to support the abstract objects' interface (its attributes and operations). For any given user interface

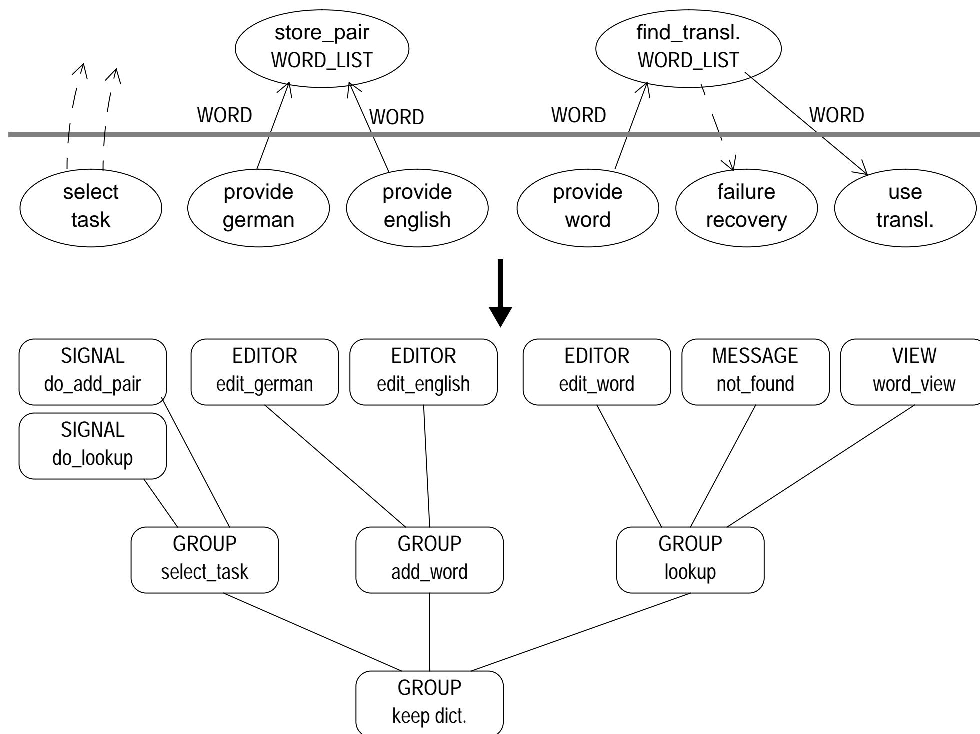

Fig. 6. Mapping from TOM model to MAUI objects

technology, possible counterparts of MAUI object as well as guidelines for mapping can be specified. We have developed such guidelines for mapping to GUI and for mapping to speech-only interaction. The guidelines consist of simple rules, for GUI for example “map signal to button” and “map group of signals to menu”. Such guidelines are aimed at mapping a modality abstraction to a concrete user interface in the context of a particular technology. With new technologies becoming available as true design options, there is increasingly a need for more general guidelines for mapping from abstract objects to objects associated with different modalities. A first step in this direction is Bernsen’s IMAP method [2].

5 Current status

We have implemented the MAUI toolkit in the object-oriented language Eiffel. Based on this toolkit, we have developed a speech-only user interface toolkit whose objects inherit the MAUI interface. Further, we have applied the adapter pattern to integrate Vision, Eiffel’s platform-independent GUI toolkit, with MAUI. The adapter objects inherit their interfaces from MAUI objects and delegate operations to Vision objects. For support of the MEMFIS method we currently develop a CASE tool. The CASE tool contains graphical editors for the SAM and TOM models and is integrated with the EiffelBench software development environment. SAM and TOM models can be mapped to Eiffel code templates. SAM objects can be mapped directly to Eiffel objects. The hierarchical tasks in TOM are mapped to a delegation tree of operations. Both toolkit and method have been applied and iteratively refined in a lab environment, and are currently being validated in a small real world project.

6 Conclusion

We believe, modality abstraction is a very important concept for future development of interactive software. While the design space of interaction modalities is rapidly growing there is an urgent need for capturing the modality-independent parts of a user interface in a modality abstraction. Such an abstraction is the basis for development of interactive software against platforms that differ in supported interaction technologies. Further, modality abstraction is the basis for evolvability of interactive software: emerging technologies can easily be integrated into interactive software if modality-independent and modality-dependent parts are separated. Also, modality abstraction supports integration of multiple modalities as it captures the common ground for different modalities.

In this paper we have addressed the problem of building modality abstraction into interactive software. As a solution we have presented a toolkit and a method. The MAUI toolkit provides generic building blocks for construction of modality abstracting user interfaces. The toolkit was developed by recursive specialization of interaction capabilities rather than by generalisation from existing user interface toolkits. While the toolkit uncompromisingly abstracts from modalities it still provides enough structure to support straightforward mapping to concrete user interface toolkits. In addition to the toolkit we presented the MEMFIS method for systematic development of interactive software with built-in modality abstraction. MEMFIS consists of a number of activities that are integrated via two complementary models of interactive applications. From a user interface design perspective, MEMFIS is an approach that shifts focus on logical interaction rather than appearance. From a software engineering perspective, MEMFIS is an evolution of object-oriented methods towards integration of dedicated activities for user interface development.

7 REFERENCES

1. Arens, Y., Hovy, E., Vossers, M. On the Knowledge Underlying Multimedia Presentations. In Maybury, M.T. (Ed.) *Intelligent Multimedia Interfaces*, AAAI Press/MIT Press, Menlo Park, 1993, pp. 280-306.

2. Bernsen, N.O. Modality Theory: Supporting Multimodal Interface Design. In Proc. of ERCIM Workshop Multimodal Human-Computer Interaction, Nancy, France, 2-4 Nov. 1993, pp. 13-23.

3. Bernsen, N. O. Information mapping in practice: Rule-based multi-modal interface design.In Proc. Of 1st Intl. Workshop On Intelligence And Multimodality In Multimedia Interfaces (IMMI-1), Edinburgh, July 1995.

4. Boger, M. and Gellersen, H.-W. Dynamic and Functional Modelling in Object-Oriented Methods: a Critique and a new Approach. Technical Report, Department of Computer Science, University of Karlsruhe, Sept. 1995. Submitted for publication.

5. Constantine, L.L. Essential Modeling. Interactions, Vol. 2, No. 2, April 1995, pp. 34-46.

6. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns. Elements of Reusable Object-oriented Software. Addison-Wesley 1994.

7. Gellersen, H.-W. Software Engineering meets Human-Computer Interaction: Integrating User Interface Design in an Object-Oriented Method. In Proc. of SOFSEM '95, Milovy, Czech Republic, Nov. 1995, Springer Verlag.

8. Guthrie, W. An Overview of Portable GUI Software, SIGCHI bulletin, Vol. 27, No. 1, Jan. 1995, pp. 55-69.

9. Johnson J. Selectors: Going Beyond User-Interface Widgets. In Proc. CHI'92 Human Factors in Computing Systems, May 1992, Monterey, CA, pp. 273-279.

10. Myers, B.A. Ideas from Garnet for Future User Interface Programming Languages. In Myers, B.A. (Ed.) Languages for Developing User Interfaces. Jones and Bartlett, Boston, MA, 1992, pp. 261-277

11. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, 1991.

12. Savidis, A. and Stephanidis, C. Developing Dual User Interfaces for Integrating Blind and sighted Users: the HOMER UIMS. In Proc. CHI'95 Human Factors in Computing Systems, Denver, May 1995, pp. 106-113.

13. Wilson, S., Johnson, P., Kelly, C. Cunningham, J. and Markopoulos, P. Beyond hacking: a Model Based Approach to User Interface Design. Proc. of HCI'93, Loughborough, UK, University Press Cambridge.