TOWARDS ADAPTIVE USER INTERFACES GENERATION
One Step Closer To People

Víctor López-Jaquero, Francisco Montero, Antonio Fernández-Caballero, María D. Lozano
Laboratory on User Interaction & Software Engineering (LoUISE), University of Castilla-La Mancha, Albacete, Spain
Email: victor@info-ab.uclm.es, fmontero@info-ab.uclm.es ,caballer@info-ab.uclm.es, mlozano@info-ab.uclm.es

Keywords: Human Computer Interaction, model-based design, connectors

Abstract: User interface generation has become a Software Engineering branch of increasing interest, probably due to the great amount of money, time and effort used to develop user interfaces and the increasing level of exigency of user requirements for usability (Nielsen, 1993) and accessibility (W3C, 2002) compliances. There are different kinds of users, and that is a fact we cannot ignore. Human society is full of diversity and that must be reflected in human-computer interaction design. Thus, we need to engage users in a new kind of interaction concept where user interfaces are tailored-made, and where user interfaces are intelligent and adaptive. A new generation of specification techniques is necessary to face these challenges successfully. Model-based design has proved to be a powerful tool to achieve these goals. A first step towards adaptive user interface generation is introduced by means of the concept of connector applied to model-based design of user interfaces.

1 INTRODUCTION

Many things about computers are not changing at all (Dourish, 2001). Our basic idea about what a computer is, what it does, and how it does it, for instance, has hardly changed for decades. The increase in computational power and the expanding context, in which we put that power on, suggest that we need new ways of interacting with computers, ways that are better tuned to our needs and abilities.

In the last few years, a new conceptualization of computational phenomena has placed the emphasis not on procedure but on interaction (Wegner, 1997). Human-computer interaction in traditional application development is focused on the interaction between tasks and a single user interface designed for a single kind of user. Application user mass is treated as a single entity, making no distinction between the different user stereotypes included in that user mass (figure 1a). A logical evolution should lead interaction to a development model where these stereotypes are taken into account. There are different kinds of users, and that is a fact we cannot ignore. Human society is full of diversity and that must be reflected in human-computer interaction design (figure 1b).

However, one step forward in interaction design is required in order to translate this diversity into application development. Adding support for different user profiles is, of course, more accurate than development for a single kind of user, but the real thing is that we are all a little bit different. We might match a user profile, but with our own particularities, leading to the concept of specialization (figure 1c). Thus, we need to engage users in a new kind of interaction concept where user interfaces are tailored-made for each user, and where user interfaces are intelligent and adaptive.

From business point of view, HCI is becoming more and more important, because of the high cost associated to user interface construction for applications. Different studies have shown that 48% of an application code is dedicated to user interface development, and that 50% of implementation stage time is dedicated to user interface construction (Myers, 1992).

These facts have motivated the creation of different research projects (Elwert, 1995; Vanderdonckt, 1996; Lozano, 2001) that face these problems from an automatic user interface generation point of view. These projects try to fill the gap in Software Engineering between functional modelling and user interface development.

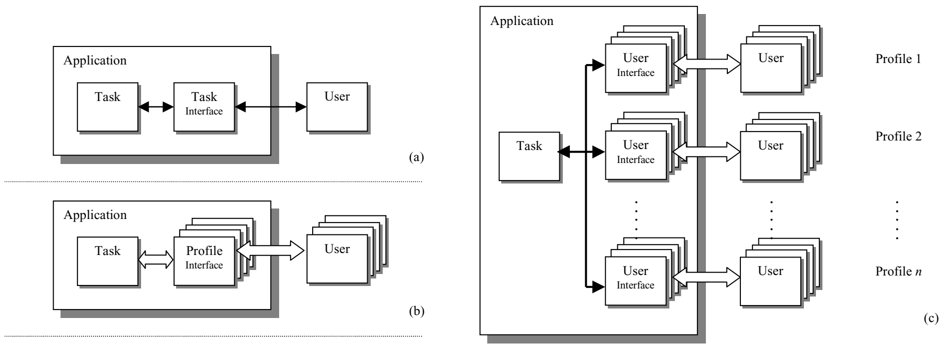


Figure 1: (a) Unity, (b) Diversity, and (c) Specialization in interaction design.

Among these projects, model based approaches (Paternò, 1999) arise as a useful and powerful tool to develop user interfaces. These approaches take as input a requirements specification that is converted into different declarative models. The most widely used are the task, the user, the domain, the dialogue and the presentation models. These declarative models are used to generate automatically a user interface compliant with the requirements captured in these models. Within these methodologies, user-centred design must be taken into account, so we are able to build *usable* (Nielsen, 1993) and *accessible* (W3C, 2002) user interfaces.

User-centred design implies studying the final user that will use the application that it is being created and make user take part in a interactive manner through all development stages.

In the next sections the connector concept is introduced applied to user interface generation and we will show how it actually makes easier adaptive and portable user interface generation.

2 USER INTERFACE DESIGN IN *IDEAS*

There are different proposals for model-based user interfaces design, *IDEAS* is one of those proposals (Lozano, Ramos & González, 2001). *IDEAS* is a methodology for user interfaces development within the framework of an automatic software production environment. This environment is supported by the object-oriented model *OASIS* (Letelier, 1998).

Abstraction is one of the basic principles needed to understand and model the reality. The object oriented paradigm favours this principle as it conceives the object oriented development process as an iterative and incremental approach that progressively allows a detailed specification of the system to be obtained.

The user interface development process within *IDEAS* is tackled following this principle. This process is not flat, but it is structured in multiple levels and multiple perspectives. The vertical structuring shows the reification processes followed from the first and most abstract level passing through the following levels to finally reach the system implementation, which constitutes the last level. On the other hand, the horizontal structuring shows the different perspectives offered by the different models developed in every one of the vertical levels. Thus, different models are used at the same abstract level to describe the different aspects of the graphical user interface.

Following these ideas, we propose the user interface development process depicted in figure 2. Due to space constraints, we cannot detail the different models proposed, so we will briefly describe this process showing some examples of the implemented tool which, interactively and automatically, supports this methodological approach.

At requirements level three models are created: the *Use Case Model*, the *Task Model* and the *User Model*. The *Use Case Model* captures the use cases identified within the information system. Then, for every one of the use cases there will be one or more tasks which the user may perform to accomplish the functionality defined by the use case.

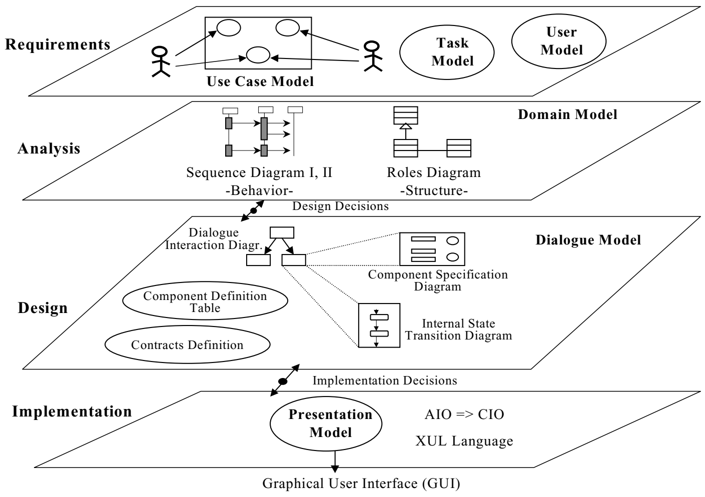


Figure 2: *IDEAS* user interface development methodology.

These tasks will be modelled in the task model. The *Task Model* defines the ordered set of activities and actions the user has to perform to achieve a concrete purpose or goal. We propose a template based on the one proposed by Cockburn (Cockburn, 2001) to describe in natural language all these issues. The *User Model* describes the characteristics of the different types of users. The purpose of this model is to support the creation of individual and personalized user interfaces. At analysis level the *Domain Model* is generated. This model consists of two diagrams. The first one is the Sequence Diagram, which defines the system behaviour. The second one is the *Role Model*, which defines the structure of the classes that take part in the associated sequence diagram together with the relationships among these classes, specifying the role of each one of them.

At design level the *Dialogue Model* is developed. All the models that have been generated up to now do not contain any graphical aspect of the final user interface. It is from now on that these issues start to be addressed and the way in which the user-system interaction will be performed is especially important.

The purpose is to describe the syntactic structure of the user-computer interaction. It establishes when the user can invoke commands, select or specify the input data and when the computer can require data from the user or display the output data. These items are modelled by means of Abstract Interaction Objects (AIO) (Vanderdonckt & Bodart, 1993).

At implementation level the *Presentation Model* is created. The Presentation Model describes the concrete interaction objects (CIO) composing the final graphical user interface, its design characteristics and visual dependencies among the objects. This model leads to the visualization of the final graphical user interface according to the final platform style guides. The final graphical user interface generation is performed by using XUL (Boswell, 2002), an XML based language, in order to make it as independent as possible from the final platform where the application is going to run.

The starting point for generating the graphical user interface in XUL is the Dialog Model developed at design level, which, as stated before, models the structure and the behaviour of the graphical user interface by means of AIOs. These AIOs are translated into the CIOs offered by XUL. Therefore, the graphical user interface structure is generated automatically from the Component Specification Diagram created at design level.

As a result of *IDEAS* methodology applied to an application three different societies of objects will appear: (1) the functional domain object society, that represents the objects that perform the functionality required in order to achieve the identified tasks, (2) the abstract interaction object society, that includes the objects that represent graphical user interface in an abstract manner, and finally (3) the concrete interaction object society, that will contain the objects that represent the graphical user interface in a specific platform.

Therefore, user interface operation will consist on the interaction between the objects included in the same society (intra-society interaction), the interaction between functional domain objects and abstract interaction objects, and the interaction between abstract interaction objects and concrete interaction objects (intersociety interaction) (see figure 3a).

Originally in *IDEAS* methodology interaction between functional domain objects and abstract interaction objects was specified using a modification of the principles about *contracts* between objects proposed by Andrade and Fiadeiro (Andrade & Fiadeiro, 1999).

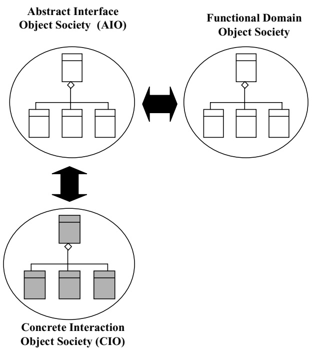
3 INTERCONNECTING OBJECTS IN IDEAS: CONTRACTS

A contract describes interaction between objects creating an association between the objects involved in the interaction act. Business rules determine the policy that will rule the communication process and the coordination between objects, where that coordination will not be included in the definition of the interacting objects, but into contracts, because of the nature inherent to business rules. Business rules are associated to the tasks performed using the user interface, and not to the interacting objects.

Including this communication process into the contract definition will allow us to adjust business rules according to the possible changes in system requirements in a transparent manner for the objects involved in the interaction act.

Figure 3b shows an excerpt of a contract specification using the template in *OASIS* language proposed in *IDEAS* methodology for this purpose.

To make easier understanding the mechanism described we have chosen a well-known interaction act: the interaction between a *Main Window* (belonging to abstract interaction object society - AIO), and a functional domain object that represents the global configuration for an application. The objects interacting will be described in the *partners* section of the specification. This interaction is included into a task we have called *Window_Management*. When *Main Window* tries to close - maybe because the user has ask the system to do so – the request will be captured by the contract according to the guard conditions specified in *when* clauses. The contract will check whether pre-condition is satisfied or not for that action. Pre-condition is specified in *with* clause. In our example, coordination event *CW* models this interaction as specified in *when* clause. First, it will check pre-condition. In this case it checks whether configuration has changed or not. If it has not changed it is not necessary to save data. Second, if the pre-condition is satisfied the actions in *do* clause will be executed. If the pre-condition is not satisfied the coordination event ends.


Abstract Interface Object Society (AIO)
Functional Domain Object Society
Concrete Interaction Object Society (CIO)

```
  ContractClass CWindow_Management
    identification
      code:(code)
    task
      aWindow:Window_Management
    partners
    GUI objects // (AIO)
      aMainWindow:Window;
    domain objects
      aConfig:Configuration;
    constant atrributes
      code:nat;
    variable attributes
      ConfigChanged:bool(FALSE);
    coordination
    CW: when aMainWindow.calls(aWindow.Close)
        do aConfig.SaveWindow(aMainWindow)
        with ConfigChanged = TRUE
    …
    valuations
      [SavedConfig] ConfigChanged = FALSE
    …
end class CWindow_Management
```

(a) (b)

Figure 3: (a) User interface operation. (b) Contract class according to the template proposed in *IDEAS*.

Contracts provide a mechanism for object coordination where interacting objects are treated as black boxes.

Although this artefact greatly improves system flexibility, as long as it supports the modification of changes in business rules quite easily, it still introduces some drawbacks for adaptive and portable user interfaces generation. In contracts coordination between objects is specified explicitly inside the contract, so it makes hard to use "plug and play" coordination components according to different users (maybe profiles) or target platforms. It takes one step beyond so that we are allowed to switch between different components easily, even at run time. Connectors (Allen, 1997) provide a powerful tool to support this software coordination "plug and play" component paradigm.

4 ONE STEP BEYOND: CONNECTORS

A connector consists of a set of roles and the specification of glue to keep them together. Roles model the behaviour for each part involved in interaction. Glue, on the other hand, provides the coordination between instances for each role (Wermelinger, 2000).

Connectors where originally proposed for software architecture specification to provide a mechanism for software components interconnection. To use connectors in the construction process of a specific system, roles will be instantiated. Nevertheless, a component will not be able to instantiate the role if it doesn't comply with the specified service that role should play.

A connector is specified describing: (1) input variables that will be used as input ports, (2) output variables that will be used as output ports, and (3) a set of actions, which will be fired according to a guard condition. Both, variables and actions can be declared as public or private items. Private items are only available to the connector where they have been declared.

Communication between components is achieved in two different ways. On one hand, input and output variables from different components are interconnected, and on the other hand methods from several components may be synchronized.

When applying connectors to our object societies (Functional domain object society, Abstract interaction object society and Concrete interaction object society) we will need to encapsulate interacting objects within component interfaces, interconnected using connector paradigm. We will exemplify how to use connectors in user interface design by specifying the same scenario we described for contracts before.

Now we have a CIO for the AIO that represents the window. When the CIO wants to close, it will notify to the AIO component its intentions so it can react and perform any required action before it actually closes. In the example, *AIOWindow* should notify *Config* component. Then *Config* component will check whether the configuration for that window has changed or not, and if so it will ask the right object (*oConfig*) to save window information.

The communication protocol between the components and the objects involved in this coordination process is depicted in figure 4. Notice interconnection between input and output variables is shown too, where little white squares are input variables and grey little squares are output ones.

As shown in figure 4, three components are involved:

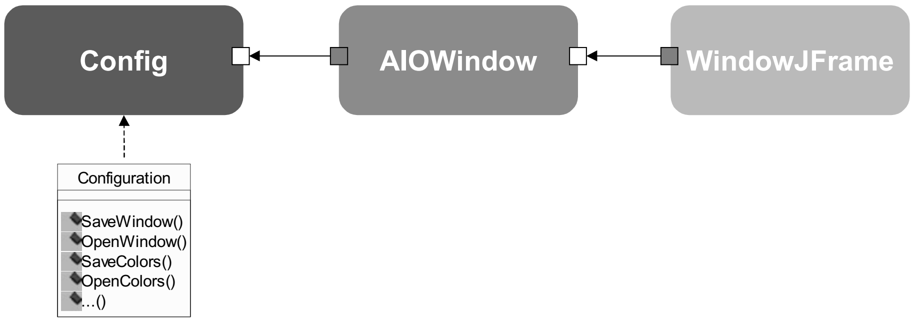


Figure 4: Connector and classes involved in *Window_Management* scenario.

```
Component Config
  IN Cstatus:{open, close, …)
  PRV ConfigChanged:boolean
  PRV oConfig: Configuration
do eCLOSE: if (Cstatus = Close) and
                 (ConfigChanged = TRUE) then
                   oConfig.SaveWindow(AIOWindow);
                   ConfigChanged := FALSE;
              end if ;
 …
End Component
```

```
Component AIOWindow
  IN status:{open, close, …)
  OUT AIOstatus:{open, close, …)
…
End Component
```

```
Component WindowJMainFrame
  IN status:{open, close, …)
  OUT AIOstatus:{open, close, …)
…
End Component
```

Figure 5: Connector components specification (excerpt).

(1) *WindowJFrame*, that models the CIO – a Java language frame -, (2) *AIOWindow*, which models the AIO that represents an application window, and finally (3) *Config*, which represents application configuration. This component makes use of an instance *oConfig* from *Configurator* class, and will do the real job. This class belongs to functional domain object society.

Next we will specify those depicted components, according to the semantics we have already described - input variables, output variables and actions. A specification of all three components involved in the coordination process in the example scenario is shown in figure 5.

So, what makes it different from contracts? The main difference is that involved components are specified separately and that they are interconnected through their interfaces. Therefore, it makes it possible for us to replace one component with another one whenever we may find it necessary. The only thing we should take into account is that the new brand component is compliant with the service requested and offers the same interface to the environment. Therefore, this will support cross platform development, as long as we can connect AIO components to any CIO (maybe CIOs for different platforms) which is able to offer the required functionality and interface to AIO component. For instance, in our window management example we could have components representing CIOs for XUL (Boswell, 2002) windows, Java^{tm} (Java, 2002) windows or Microsoft Windows^{tm}. Thus, our design process will boost portability and cross platform development with all the advantages it provides – above all reduced costs.

But this ability to switch between different components not only supports portability and cross platform design, it supports adaptive user interfaces specification to greatly improve overall user interface quality. AIO components are the traders between functional domain objects and CIO components, but we propose "intelligent" AIOs which are able to process the information to be presented, so they can choose between different CIOs to meet user preferences or device dependent features. For instance, a menu for an application could be presented in different ways depending on the number of options available for selection, as proposed in (Vanderdonckt, 1993). Thus, if there are just two options available a simple *checkbox* could do the work. If the available number of options is three, a set of grouped *radiobuttons* will be an interesting choice, while if there are more than three options available; a *listbox* could be used for this purpose.

Designing both CIO and AIO once we are able to achieve two great features: (1) we really boost portability and cross platform development, and (2) we generate automatically adaptive interfaces for all the applications using the designed AIO and CIO sets of components.

5 CONCLUSIONS

User interface generation has become a Software Engineering branch of increasing interest, probably due to the great amount of money, time and effort used to develop user interfaces and the increasing level of exigency of user requirements for usability (Nielsen, 1993) and accessibility (W3C, 2002) compliance interfaces. Besides the kind of users engaged in HCI is becoming more and more heterogeneous, and that is a fact we can not ignore.

In this paper we have proposed a first step towards user interface design and generation with some adaptive features by means of connectors in a model based user interface design methodology: *IDEAS*. We have shown how connectors can be used to introduce a high degree of portability and cross platform design, and how connectors can support adaptive user interfaces generation.

ACKNOWLEDGEMENTS

This work is supported in part by the Spanish CICYT TIC 2000-1673-C06-06 and CICYT TIC 2000-1106-C02-02 grants.

REFERENCES

Allen, R., Garlan, D. 1997. A Formal Basis for Architectural Connectors, *ACM TOSEM*, 6(3), pg. 213-249, July.

Andrade, L.F., Fiadeiro, J.L. 1999. Interconnecting Objects via Contracts. In: UML'99. *Proceedings of the International conference on the Unified Modeling Language.*

Boswell, D., King, B., Oeschger, I., Collins, P., Murphy, E. 2002. Creating Applications with Mozilla. O'Reilly. 0-596-00052-9.

Cockburn, A. 2001. *Writing Effective Use Cases*. Addison-Wesley.

Dourish, P. 2001.Where the Action Is: The Foundations of Embodied Interaction. Massachusetts Institute of Technology.

Elwert, T., Schlungbaum, E. 1995. Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. In: *Designing, Specification and Verification of Interactive Systems.* Wien: Springer, 193-208.

Java. Sun Microsystems. 2002. http://java.sun.com.

Letelier, P., Ramos, I., Sánchez, P., Pastor, O. 1998. OASIS version 3: A Formal Approach for Object Oriented Conceptual Modeling. SPUPV-98.4011. Edited by Universidad Politécnica de Valencia, Spain.

Lozano, M. 2001. Entorno Metodológico Orientado a Objetos para la Especificación y Desarrollo de Interfaces de Usuario. Ph.D. Thesis. Supervisors: Dr. Isidro Ramos / Dr. Pascual Gonzalez. UPV. Valencia, 2001.

Lozano, M., Ramos, I., González, P. 2001. User Interface Specification and Modelling in an Object Oriented Environment for Automatic Software Development. IEEE 34th International Conference on TOOLS, USA.

Myers, B. A., Rosson, M. B.. 1992. Survey on User Interface Programming. In Striking a Balance. *Proceedings CHI'92*. Monterey, May 1992, New York: ACM Press, 195-202..

Nielsen, J. 1993. Usability Engineering. Academic Press.

Paternò, F. 1999. Model-Based Design and Evaluation of Interactive Applications. Springer.

Puerta, A.R. 1997. A Model-Based Interface Development Environment. IEEE Software, pp. 40-47.

Vanderdonckt. J.; Bodart, F. 1993. Encapsulating Kwowledge for Intelligence Interaction Objects Selection. Proceedings of Inter-CHI'93. ACM Press, 424-429.

Vanderdonckt, J. 1993. A Corpus of Selection Rules for Choosing Interaction Objects, Technical Report TR 93/3, University of Namur.

Vanderdonckt, J. 1996. Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience. Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix. Namur, Belgica.

Vanderdonckt, J. 1996. Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience. Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix. Namur, Belgica.

W3C. 2002. WAI. http://www.w3.org/WAI/

Wegner, P. 1997. Why interaction is more powerful than algorithms. Communications of the ACM, Vol. 40, No. 5 (1997) 80-91.

Wermelinger, M., Lopes, A., Fiadeiro, J.L. 2000. Superposing connectors, *in Proc. 10h International Workshop on Software Specification and Design*, IEEE Computer Society Press, 87-94